INTERNATIONAL STANDARD

ISO/IEC 11889-4

Second edition
2015-12-15

Information technology — Trusted Platform Module Library —

Part 4: Supporting Routines

Technologies de l'information — Bibliothèque de module de plate-forme de confiance —
Partie 4: Routines de support

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

Reference number
ISO/IEC 11889-4:2015(E)

© ISO/IEC 2015

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2015, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

 © ISO/IEC 2015 – All rights reserved

CONTENTS

Foreword xvii
Introduction xviii
1 Scope 1
2 Normative references 2
3 Terms and definitions 2
4 Symbols and abbreviated terms 2
5 Automation 2
5.1 Introduction 2
5.2 Configuration Parser 2
5.3 Structure Parser 3
5.3.1 Introduction 3
5.3.2 Unmarshaling Code Prototype 3
5.3.3 Marshaling Code Function Prototypes 5
5.4 Command Parser 6
5.5 Portability 6
6 Header Files 7
6.1 Introduction 7
6.2 BaseTypes.h 7
6.3 bits.h 8
6.4 bool.h 9
6.5 Capabilities.h 9
6.6 TPMB.h 9
6.7 TpmError.h 10
6.8 Global.h 10
6.8.1 Description 10
6.8.2 Includes 11
6.8.3 Defines and Types 11
6.8.4 Loaded Object Structures 12
6.8.5 AUTH_DUP Types 14
6.8.6 Active Session Context 14
6.8.7 PCR 16
6.8.8 Startup 17
6.8.9 NV 18
6.8.10 COMMIT_INDEX_MASK 19
6.8.11 RAM Global Values 19
6.8.12 Persistent Global Values 21
6.8.13 Global Macro Definitions 26
6.9 Private data 26
6.10 Tpm.h 31
6.11 swap.h 31
6.12 InternalRoutines.h 32
6.13 TpmBuildSwitches.h 33
6.14 VendorString.h 34
7 Main 36
7.1 CommandDispatcher() 36
7.2 ExecCommand.c 36
7.2.1 Introduction 36
7.2.2 Includes 36
7.2.3 ExecuteCommand() 36

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

7.3 ParseHandleBuffer() 42
7.4 SessionProcess.c 43
7.4.1 Introduction 43
7.4.2 Includes and Data Definitions 43
7.4.3 Authorization Support Functions 43
7.4.4 Session Parsing Functions 50
7.4.5 Response Session Processing 68
8 Command Support Functions 78
8.1 Introduction 78
8.2 Attestation Command Support (Attest_spt.c) 78
8.2.1 Includes 78
8.2.2 Functions 78
8.3 Context Management Command Support (Context_spt.c) 81
8.3.1 Includes 81
8.3.2 Functions 81
8.4 Policy Command Support (Policy_spt.c) 83
8.4.1 Includes 83
8.4.2 Functions 83
8.5 NV Command Support (NV_spt.c) 85
8.5.1 Includes 85
8.5.2 Functions 86
8.6 Object Command Support (Object_spt.c) 88
8.6.1 Includes 88
8.6.2 Local Functions 88
8.6.3 Public Functions 93
9 Subsystem 113
9.1 CommandAudit.c 113
9.1.1 Introduction 113
9.1.2 Includes 113
9.1.3 Functions 113
9.2 DA.c 117
9.2.1 Introduction 117
9.2.2 Includes and Data Definitions 117
9.2.3 Functions 117
9.3 Hierarchy.c 120
9.3.1 Introduction 120
9.3.2 Includes 120
9.3.3 Functions 120
9.4 NV.c 124
9.4.1 Introduction 124
9.4.2 Includes, Defines and Data Definitions 124
9.4.3 NV Utility Functions 124
9.4.4 NV Index and Persistent Object Access Functions 126
9.4.5 RAM-based NV Index Data Access Functions 130
9.4.6 Utility Functions 132
9.4.7 NV Access Functions 140
9.5 Object.c 158

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.5.1 Introduction 158
9.5.2 Includes and Data Definitions 158
9.5.3 Functions 158
9.6 PCR.c 174
9.6.1 Introduction 174
9.6.2 Includes, Defines, and Data Definitions 174
9.6.3 Functions 174
9.7 PP.c 196
9.7.1 Introduction 196
9.7.2 Includes 196
9.7.3 Functions 196
9.8 Session.c 199
9.8.1 Introduction 199
9.8.2 Includes, Defines, and Local Variables 200
9.8.3 File Scope Function -- ContextIdSetOldest() 200
9.8.4 Startup Function -- SessionStartup() 201
9.8.5 Access Functions 202
9.8.6 Utility Functions 204
9.9 Time.c 216
9.9.1 Introduction 216
9.9.2 Includes 216
9.9.3 Functions 216
10 Support 221
10.1 AlgorithmCap.c 221
10.1.1 Description 221
10.1.2 Includes and Defines 221
10.1.3 AlgorithmCapGetImplemented() 222
10.2 Bits.c 224
10.2.1 Introduction 224
10.2.2 Includes 224
10.2.3 Functions 224
10.3 CommandAttributeData.c 225
10.4 CommandCodeAttributes.c 231
10.4.1 Introduction 231
10.4.2 Includes and Defines 231
10.4.3 Command Attribute Functions 231
10.5 DRTM.c 236
10.5.1 Description 236
10.5.2 Includes 236
10.5.3 Functions 236
10.6 Entity.c 237
10.6.1 Description 237
10.6.2 Includes 237
10.6.3 Functions 237
10.7 Global.c 243
10.7.1 Description 243
10.7.2 Includes and Defines 244
10.7.3 Global Data Values 244

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.7.4 Private Values 244
10.8 Handle.c 246
10.8.1 Description 246
10.8.2 Includes 246
10.8.3 Functions 246
10.9 Locality.c 248
10.9.1 Includes 248
10.9.2 LocalityGetAttributes() 248
10.10 Manufacture.c 248
10.10.1 Description 248
10.10.2 Includes and Data Definitions 248
10.10.3 Functions 249
10.11 Marshal.c 251
10.11.1 Introduction 251
10.11.2 Unmarshal and Marshal a Value 251
10.11.3 Unmarshal and Marshal a Union 252
10.11.4 Unmarshal and Marshal a Structure 254
10.11.5 Unmarshal and Marshal an Array 256
10.11.6 TPM2B Handling 258
10.12 MemoryLib.c 259
10.12.1 Description 259
10.12.2 Includes and Data Definitions 259
10.12.3 Functions on BYTE Arrays 259
10.13 Power.c 264
10.13.1 Description 264
10.13.2 Includes and Data Definitions 264
10.13.3 Functions 264
10.14 PropertyCap.c 265
10.14.1 Description 265
10.14.2 Includes 265
10.14.3 Functions 265
10.15 TpmFail.c 273
10.15.1 Includes, Defines, and Types 273
10.15.2 Typedefs 273
10.15.3 Local Functions 274
10.15.4 Public Functions 275
10.15.5 TpmFailureMode 275
11 Cryptographic Functions 279
11.1 Introduction 279
11.2 CryptUtil.c 279
11.2.1 Introduction 279
11.2.2 Includes 279
11.2.3 TranslateCryptErrors() 279
11.2.4 Random Number Generation Functions 280
11.2.5 Hash/HMAC Functions 281
11.2.6 RSA Functions 294
11.2.7 ECC Functions 302
11.2.8 Symmetric Functions 312
11.2.9 Initialization and shut down 316

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

11.2.10 Algorithm-Independent Functions 317
11.2.11 Math functions 339
11.2.12 Capability Support 341

11.3 Ticket.c 343

11.3.1 Introduction 343
11.3.2 Includes 343
11.3.3 Functions 343

11.4 CryptSelfTest.c 346

11.4.1 Introduction 346
11.4.2 Functions 347

Annex A (informative) Implementation Dependent 351

A.1 Introduction 351
A.2 Implementation.h 351

Annex B (informative) Cryptographic Library Interface 365

B.1 Introduction 365
B.2 Integer Format 365
B.3 CryptoEngine.h 365

B.3.1. Introduction 365
B.3.2. General Purpose Macros 366
B.3.3. Self-test 366
B.3.4. Hash-related Structures 366
B.3.5. Asymmetric Structures and Values 368
B.3.6. ECC-related Structures 368
B.3.7. RSA-related Structures 368

B.4 OsslCryptoEngine.h 370

B.4.1. Introduction 370
B.4.2. Defines 370

B.5 MathFunctions.c 371

B.5.1. Introduction 371
B.5.2. Externally Accessible Functions 371

B.6 CpriCryptPri.c 381

B.6.1. Introduction 381
B.6.2. Includes and Locals 381
B.6.3. Functions 381

B.7 CpriRNG.c 383

B.7.1. Introduction 383
B.7.2. Defines 383
B.7.3. Includes and Values 383
B.7.4. Functions 383

B.8 CpriHash.c 386

B.8.1. Description 386
B.8.2. Includes, Defines, and Types 386
B.8.3. Static Functions 386
B.8.4. Hash Functions 388
B.8.5. HMAC Functions 395
B.8.6. Mask and Key Generation Functions 397

B.9 CpriHashData.c 402
B.10 CpriMisc.c 403

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.10.1. Includes....403
B.10.2. Functions....403
B.11 CpriSym.c....405
B.11.1. Introduction....405
B.11.2. Includes, Defines, and Typedefs....405
B.11.3. Utility Functions....406
B.11.4. Symmetric Encryption....407
B.12 RSA Files....412
B.12.1. CpriRSA.c....412
B.12.2. Alternative RSA Key Generation....436
B.13 Elliptic Curve Files....467
B.13.1. CpriDataEcc.h....467
B.13.2. CpriDataEcc.c....468
B.13.3. CpriECC.c....471
Annex C (informative) Simulation Environment....509
C.1 Introduction....509
C.2 Cancel.c....509
C.2.1. Introduction....509
C.2.2. Includes, Typedefs, Structures, and Defines....509
C.2.3. Functions....509
C.3 Clock.c....511
C.3.1. Introduction....511
C.3.2. Includes and Data Definitions....511
C.3.3. Functions....511
C.4 Entropy.c....513
C.4.1. Includes and Defines....513
C.4.2. Local values....513
C.4.3. _plat__GetEntropy()....513
C.5 LocalityPlat.c....515
C.5.1. Includes....515
C.5.2. Functions....515
C.6 NVMem.c....516
C.6.1. Introduction....516
C.6.2. Includes....516
C.6.3. Functions....516
C.7 PowerPlat.c....521
C.7.1. Includes....521
C.7.2. Functions....521
C.8 Platform.h....523
C.8.1. Includes and Defines....523
C.8.2. Power Functions....523
C.8.3. Physical Presence Functions....523
C.8.4. Command Canceling Functions....524
C.8.5. NV memory functions....525
C.8.6. Locality Functions....527
C.8.7. Clock Constants and Functions....527
C.8.8. Entropy Function _plat__GetEntropy()....529

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

C.9 PlatformData.h 530
C.10 PlatformData.c 531
C.10.1. Description 531
C.10.2. Includes 531
C.11 PPPlat.c 532
C.11.1. Description 532
C.11.2. Includes 532
C.11.3. Functions 532
C.12 Unique.c 534
C.12.1. Introduction 534
C.12.2. Includes 534
C.12.3. _plat__GetUnique() 534
Annex D (informative) Remote Procedure Interface 535
D.1 Introduction 535
D.2 TpmTcpProtocol.h 536
D.2.1. Introduction 536
D.2.2. Defines 536
D.2.3. Typedefs 536
D.3 TcpServer.c 538
D.3.1. Description 538
D.3.2. Includes, Locals, Defines and Function Prototypes 538
D.3.3. Functions 538
D.4 TPMCmdp.c 548
D.4.1. Description 548
D.4.2. Includes and Data Definitions 548
D.4.3. Functions 548
D.5 TPMCmds.c 554
D.5.1. Description 554
D.5.2. Includes, Defines, Data Definitions, and Function Prototypes 554
D.5.3. Functions 554
Bibliography 556

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Tables

Table 1 43
Table 2 44
Table 3 46
Table 4 46
Table 5 47
Table 6 49
Table 7 52
Table 8 54
Table 9 55
Table 10 58
Table 11 61
Table 12 62
Table 13 64
Table 14 64
Table 15 67
Table 16 78
Table 17 80
Table 18 86
Table 19 87
Table 20 88
Table 21 92
Table 22 93
Table 23 94
Table 24 97
Table 25 102
Table 26 104
Table 27 108
Table 28 111
Table 29 114
Table 30 115
Table 31 115
Table 32 116
Table 33 123
Table 34 125
Table 35 128
Table 36 130

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 37 136
Table 38 136
Table 39 138
Table 40 140
Table 41 141
Table 42 142
Table 43 143
Table 44 146
Table 45 149
Table 46 150
Table 47 154
Table 48 155
Table 49 159
Table 50 160
Table 51 162
Table 52 163
Table 53 166
Table 54 166
Table 55 167
Table 56 168
Table 57 169
Table 58 172
Table 59 172
Table 60 172
Table 61 174
Table 62 175
Table 63 176
Table 64 176
Table 65 178
Table 66 179
Table 67 180
Table 68 181
Table 69 184
Table 70 185
Table 71 186
Table 72 189
Table 73 192
Table 74 193
Table 75 195

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 76 195
Table 77 198
Table 78 198
Table 79 202
Table 80 203
Table 81 203
Table 82 204
Table 83 205
Table 84 208
Table 85 209
Table 86 213
Table 87 214
Table 88 219
Table 89 223
Table 90 224
Table 91 231
Table 92 232
Table 93 232
Table 94 233
Table 95 233
Table 96 235
Table 97 237
Table 98 247
Table 99 249
Table 100 250
Table 101— Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type from ISO/IEC 11889-2....251
Table 102 — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S> from ISO/IEC 11889-2....252
Table 103 — Definition of TPMT_PUBLIC Structure from ISO/IEC 11889-2....254
Table 104 — Definition of TPML_DIGEST Structure from ISO/IEC 11889-2....256
Table 105 — Definition of TPM2B_EVENT Structure from ISO/IEC 11889-2....258
Table 106....260
Table 107 262
Table 108 265
Table 109 265
Table 110 272
Table 111 279
Table 112 281
Table 113 282
Table 114 283

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 115 283
Table 116 284
Table 117 284
Table 118 286
Table 119 287
Table 120 287
Table 121 288
Table 122 288
Table 123 289
Table 124 289
Table 125 290
Table 126 290
Table 127 291
Table 128 291
Table 129 294
Table 130 295
Table 131 296
Table 132 297
Table 133 299
Table 134 300
Table 135 301
Table 136 303
Table 137 304
Table 138 305
Table 139 306
Table 140 307
Table 141 307
Table 142 309
Table 143 310
Table 144 317
Table 145 320
Table 146 322
Table 147 326
Table 148 328
Table 149 330
Table 150 331
Table 151 335
Table 152 337
Table 153 338

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 154 339
Table 155 340
Table 156 340
Table 157 341
Table 158 342
Table 159 343
Table 160 347
Table 161 348
Table 162 349
Table B.1 368
Table B.2 371
Table B.3 372
Table B.4 372
Table B.5 373
Table B.6 374
Table B.7 375
Table B.8 377
Table B.9 377
Table B.10 378
Table B.11 380
Table B.12 387
Table B.13 389
Table B.14 389
Table B.15 390
Table B.16 391
Table B.17 392
Table B.18 394
Table B.19 395
Table B.20 396
Table B.21 397
Table B.22 398
Table B.23 400
Table B.24 403
Table B.25 406
Table B.26 409
Table B.27 412
Table B.28 414
Table B.29 416
Table B.30 417

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table B.31 418
Table B.32 419
Table B.33 421
Table B.34 422
Table B.35 422
Table B.36 424
Table B.37 426
Table B.38 427
Table B.39 428
Table B.40 429
Table B.41 430
Table B.42 431
Table B.43 432
Table B.44 439
Table B.45 442
Table B.46 445
Table B.47 451
Table B.48 472
Table B.49 473
Table B.50 475
Table B.51 477
Table B.52 479
Table B.53 481
Table B.54 482
Table B.55 487
Table B.56 489
Table B.57 492
Table B.58 494
Table B.59 495
Table B.60 497
Table B.61 498
Table B.62 500
Table B.63 502
Table B.64 504
Table B.65 507
Table C.1 509
Table C.2 513
Table C.3 516
Table C.4 518

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table C.5 519
Table C.6 520
Table C.7 524
Table C.8 524
Table C.9 525
Table C.10 525
Table C.11 526
Table C.12 526
Table C.13 529
Table C.14 532

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT), see the following URL: Foreword — Supplementary information.

ISO/IEC 11889-4 was prepared by the Trusted Computing Group (TCG) and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

This second edition cancels and replaces the first edition (ISO/IEC 11889-4:2009), which has been technically revised.

ISO/IEC 11889 consists of the following parts, under the general title *Information technology — Trusted Platform Module Library:*

– *Part 1: Architecture*

– *Part 2: Structures*

– *Part 3: Commands*

– *Part 4: Supporting routines*

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Introduction

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured the ISO and IEC that he/she is willing to negotiate licences either free of charge or under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from:

Fujitsu Limited **1-1, Kamikodanaka 4-chrome, Nakahara-ku, Kawasaki-shi, Kanagawa, 211-8588 Japan**
Microsoft Corporation **One Microsoft Way, Redmond, WA 98052**
Enterasys Networks, Inc **50 Minuteman Road, US-Andover, MA 01810**
Lenovo **1009 Think Place, US-Morrisville, NC 27560-8496**
Advanced Micro devices, Inc. - AMD **7171 Southwest Parkway, Mailstop B100.3, US-Austin, Texas 78735**
Hewlett-Packard Company **P.O. Box 10490, US-Palo Alto, CA 94303-0969**
Infineon Technologies AG - Neubiberg **Am Campeon 1-12, DE-85579 Neubiberg**
Sun Microsystems Inc. - Menlo Park, CA **10 Network Circle, UMPK10-146, US-Menlo Park, CA 94025**
IBM Corporation **North Castle Drive, US-Armonk, N.Y. 10504**
Intel Corporation **5200 Elam Young Parkway, US-Hillsboro, OR 97123**

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO (www.iso.org/patents) and IEC (http://patents.iec.ch) maintain on-line databases of patents relevant to their standards. Users are encouraged to consult the databases for the most up to date information concerning patents.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Information technology — Trusted Platform Module Library —

Part 4: Supporting routines

1 Scope

This part of ISO/IEC 11889 contains C code that describes the algorithms and methods used by the command code in ISO/IEC 11889-3. The code in this part of ISO/IEC 11889 augments ISO/IEC 11889-2 and ISO/IEC 11889-3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

Any code in this part of ISO/IEC 11889 may be replaced by code that provides similar results when interfacing to the action code in ISO/IEC 11889-3. The behavior of code in this part of ISO/IEC 11889 that is not included in an annex is *normative*, as observed at the interfaces with ISO/IEC 11889-3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of ISO/IEC 11889 from the provided code.

The code in ISO/IEC 11889-3 and this part of ISO/IEC 11889 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 ISO/IEC 11889-3 would be compliant.

The code in ISO/IEC 11889-3 and this part of ISO/IEC 11889 is not written to meet any particular level of conformance nor does ISO/IEC 11889 require that a TPM meet any particular level of conformance.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 9797-2, *Information technology -- Security techniques -- Message Authentication Codes (MACs) -- Part 2: Mechanisms using a dedicated hash-function*
- ISO/IEC 10116:2006, *Information technology — Security techniques — Modes of operation for an n-bit block cipher*
- ISO/IEC 11889-1, *Information technology — Trusted Platform Module Library — Part 1: Architecture*
- ISO/IEC 11889-2, *Information technology — Trusted Platform Module Library — Part 2: Structures*
- ISO/IEC 11889-3, *Information technology — Trusted Platform Module Library — Part 3: Commands*

3 Terms and definitions

For the purposes of this part of ISO/IEC 11889, the terms and definitions given in ISO/IEC 11889-1 apply.

4 Symbols and abbreviated terms

For the purposes of this part of ISO/IEC 11889, the symbols and abbreviated terms given in ISO/IEC 11889-1 apply.

5 Automation

5.1 Introduction

ISO/IEC 11889-2 and ISO/IEC 11889-3 are constructed so that they can be processed by an automated parser.

EXAMPLE 1 ISO/IEC 11889-2 can be processed to generate header file contents such as structures, typedefs, and enums.

EXAMPLE 2 ISO/IEC 11889-3 can be processed to generate command and response marshaling and unmarshaling code.

5.2 Configuration Parser

The tables in the ISO/IEC 11889-2 Annexes are constructed so that they can be processed by a program. The program that processes these tables in the ISO/IEC 11889-2 Annexes is called "The ISO/IEC 11889-2 Configuration Parser."

The tables in the ISO/IEC 11889-2 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 ISO/IEC 11889-2 Configuration Parser produces a set of structures and definitions that are used by the ISO/IEC 11889-2 Structure Parser.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

5.3 Structure Parser

5.3.1 Introduction

The program that processes the tables in ISO/IEC 11889-2 (other than the table in the annexes) is called "The ISO/IEC 11889-2 Structure Parser."

NOTE A Perl script was used to parse the tables in ISO/IEC 11889-2 to produce the header files and unmarshaling code in for the reference implementation.

The ISO/IEC 11889-2 Structure Parser takes as input the files produced by the ISO/IEC 11889-2 Configuration Parser and ISO/IEC 11889-2. The ISO/IEC 11889-2 Structure Parser will generate all of the C structure constant definitions that are required by the TPM interface. Additionally, the parser will generate unmarshaling code for all structures passed to the TPM, and marshaling code for structures passed from the TPM.

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

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

The clauses below describe the function prototypes for the marshaling and unmarshaling code that is automatically generated by the ISO/IEC 11889-2 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.

5.3.2 Unmarshaling Code Prototype

5.3.2.1 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 ISO/IEC 11889-2).

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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

5.3.2.2 Union Types

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

The function prototype for a union has the form:

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

where:

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

5.3.2.3 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.

EXAMPLE The TPMI_ALG_HASH data type is used in many places.

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

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

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

5.3.2.4 Arrays

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

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

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

5.3.3 Marshaling Code Function Prototypes

5.3.3.1 Simple Types and Structures

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

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

Where:

TYPE	name of the data type or structure
***source**	location in the TPM memory containing the value that is to be marshaled in to the designated buffer
****buffer**	location in the output buffer where the first octet of the **TYPE** is to be placed
***size**	number of octets remaining in ****buffer**. If **size** is a NULL pointer, then no data is marshaled and the routine will compute the size of the memory required to marshal the indicated type

When the data is successfully marshaled, the called routine will return the number of octets marshaled into ****buffer.**

If the data is successfully marshaled, ***buffer** is advanced point to the first octet of the next location in the output buffer and ***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.

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

5.3.3.2 Union Types

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

The function prototype for a union has the form:

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

The parameters have a similar meaning as those in 5.3.2.2 but the data movement is from **source** to **buffer**.

5.3.3.3 Arrays

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

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

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

5.4 Command Parser

The program that processes the tables in ISO/IEC 11889-3 is called "The ISO/IEC 11889-3 Command Parser."

The ISO/IEC 11889-3 Command Parser takes as input ISO/IEC 11889-3 and some configuration files produced by the ISO/IEC 11889-2 Configuration Parser. This parser uses the contents of the command and response tables in ISO/IEC 11889-3 to produce unmarshaling code for the command and the marshaling code for the response. Additionally, this parser produces support routines that are used to check that the proper number of authorization values of the proper type have been provided. These support routines are called by the functions in this part of ISO/IEC 11889.

5.5 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.

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 0^{th} 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 0^{th} octet always contains the 0^{th} bit of the structure no matter how large the structure. However, for a big endian machine, the 0^{th} bit will be in the highest numbered octet. When unmarshaling a TPMA_SESSION, the current unmarshaling code will place the input octet at the 0^{th} octet of the TPMA_SESSION. Since the 0^{th} 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).

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

6 Header Files

6.1 Introduction

The files in clause 6 are used to define values that are used in ISO/IEC 11889-3 and this part of ISO/IEC 11889 and are not confined to a single module.

6.2 BaseTypes.h

```
#ifndef _BASETYPES_H
#define _BASETYPES_H
#include "stdint.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;
typedef struct {
    UINT16        size;
    BYTE          buffer[1];
} TPM2B;
#endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

6.3 bits.h

```
#ifndef     _BITS_H
#define     _BITS_H
#define CLEAR_BIT(bit, vector)  BitClear((bit), (BYTE *)&(vector), sizeof(vector))
#define SET_BIT(bit, vector)    BitSet((bit), (BYTE *)&(vector), sizeof(vector))
                                                                \
#define TEST_BIT(bit, vector)   BitIsSet((bit), (BYTE *)&(vector), sizeof(vector))
#endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

6.4 bool.h

```
#ifndef     _BOOL_H
#define     _BOOL_H
#if defined(TRUE)
#undef TRUE
#endif
#if defined FALSE
#undef FALSE
#endif
typedef int BOOL;
#define FALSE   ((BOOL)0)
#define TRUE    ((BOOL)1)
#endif
```

6.5 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().

```
#ifndef     _CAPABILITIES_H
#define     _CAPABILITIES_H
#define    MAX_CAP_DATA          (MAX_CAP_BUFFER-sizeof(TPM_CAP)-sizeof(UINT32))
#define    MAX_CAP_ALGS          (ALG_LAST_VALUE - ALG_FIRST_VALUE + 1)
#define    MAX_CAP_HANDLES       (MAX_CAP_DATA/sizeof(TPM_HANDLE))
#define    MAX_CAP_CC            ((TPM_CC_LAST - TPM_CC_FIRST) + 1)
#define    MAX_TPM_PROPERTIES    (MAX_CAP_DATA/sizeof(TPMS_TAGGED_PROPERTY))
#define    MAX_PCR_PROPERTIES    (MAX_CAP_DATA/sizeof(TPMS_TAGGED_PCR_SELECT))
#define    MAX_ECC_CURVES        (MAX_CAP_DATA/sizeof(TPM_ECC_CURVE))
#endif
```

6.6 TPMB.h

This file contains extra TPM2B structures

```
#ifndef _TPMB_H
#define _TPMB_H
#include "TPM_Types.h"
```

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
```

Macro to instance and initialize a TPM2B value

```
#define TPM2B_INIT(TYPE, name)  \
    TPM2B_##TYPE    name = {sizeof(name.t.buffer), {0}}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

A 2B structure for a seed

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

A 2B hash block

```
15    TPM2B_TYPE(HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
16    TPM2B_TYPE(RSA_PRIME, MAX_RSA_KEY_BYTES/2);
17    TPM2B_TYPE(1_BYTE_VALUE, 1);
18    TPM2B_TYPE(2_BYTE_VALUE, 2);
19    TPM2B_TYPE(4_BYTE_VALUE, 4);
20    TPM2B_TYPE(20_BYTE_VALUE, 20);
21    TPM2B_TYPE(32_BYTE_VALUE, 32);
22    TPM2B_TYPE(48_BYTE_VALUE, 48);
23    TPM2B_TYPE(64_BYTE_VALUE, 64);
24    TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
25    #endif
```

6.7 TpmError.h

```
 1    #ifndef _TPM_ERROR_H
 2    #define _TPM_ERROR_H
 3    #include "TpmBuildSwitches.h"
 4    #define     FATAL_ERROR_ALLOCATION          (1)
 5    #define     FATAL_ERROR_DIVIDE_ZERO         (2)
 6    #define     FATAL_ERROR_INTERNAL            (3)
 7    #define     FATAL_ERROR_PARAMETER           (4)
 8    #define     FATAL_ERROR_ENTROPY             (5)
 9    #define     FATAL_ERROR_SELF_TEST           (6)
10    #define     FATAL_ERROR_CRYPTO              (7)
11    #define     FATAL_ERROR_NV_UNRECOVERABLE    (8)
12    #define     FATAL_ERROR_REMANUFACTURED      (9) // indicates that the TPM has
13                                                    // been re-manufactured after an
14                                                    // unrecoverable NV error
15    #define     FATAL_ERROR_DRBG                (10)
16    #define     FATAL_ERROR_FORCED              (666)
```

These are the crypto assertion routines. When a function returns an unexpected and unrecoverable result, the assertion fails and the TpmFail() is called

```
17    void
18    TpmFail(const char *function, int line, int code);
19    typedef void    (*FAIL_FUNCTION)(const char *, int, int);
20    #define FAIL(a) (TpmFail(__FUNCTION__, __LINE__, a))
21    #if defined(EMPTY_ASSERT)
22    #   define pAssert(a)  ((void)0)
23    #else
24    #   define pAssert(a) (!!(a) ? 1 : (FAIL(FATAL_ERROR_PARAMETER), 0))
25    #endif
26    #endif // _TPM_ERROR_H
```

6.8 Global.h

6.8.1 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 clause, is a clause that defines global values that are only present in RAM. The

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

next three clauses define the structures for the NV data areas: persistent, orderly, and state save. Additional clauses 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.

6.8.2 Includes

```
#ifndef         GLOBAL_H
#define         GLOBAL_H
//#define SELF_TEST
#include        "TpmBuildSwitches.h"
#include        "Tpm.h"
#include        "TPMB.h"
#include        "CryptoEngine.h"
#include        <setjmp.h>
```

6.8.3 Defines and Types

6.8.3.1 Unreferenced Parameter

This define is used to eliminate the compiler warning about an unreferenced parameter. Basically, it tells the compiler that it is not an accident that the parameter is unreferenced.

```
#ifndef UNREFERENCED_PARAMETER
#   define UNREFERENCED_PARAMETER(a)   (a)
#endif
#include    "bits.h"
```

6.8.3.2 Crypto Self-Test Values

Define these values here if the AlgorithmTests() project is not used

```
#ifndef SELF_TEST
extern ALGORITHM_VECTOR     g_implementedAlgorithms;
extern ALGORITHM_VECTOR     g_toTest;
#else
LIB_IMPORT extern ALGORITHM_VECTOR     g_implementedAlgorithms;
LIB_IMPORT extern ALGORITHM_VECTOR     g_toTest;
#endif
```

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

```
#define     TEST(alg) if(TEST_BIT(alg, g_toTest)) CryptTestAlgorithm(alg, NULL)
```

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

```
#define     TEST_HASH(alg)                                          \
            if(     TEST_BIT(alg, g_toTest)                         \
                &&  (alg != ALG_NULL_VALUE))                        \
                CryptTestAlgorithm(alg, NULL)
```

6.8.3.3 Hash and HMAC State Structures

These definitions are for the types that can be in a hash state structure. These types are used in the crypto utilities

```
typedef BYTE    HASH_STATE_TYPE;
#define HASH_STATE_EMPTY        ((HASH_STATE_TYPE) 0)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
#define HASH_STATE_HASH         ((HASH_STATE_TYPE) 1)
#define HASH_STATE_HMAC         ((HASH_STATE_TYPE) 2)
```

A HASH_STATE structure contains an opaque hash stack state. A caller would use this structure when performing incremental hash operations. The state is updated on each call. If *type* is an HMAC_STATE, or HMAC_STATE_SEQUENCE then state is followed by the HMAC key in *oPad* format.

```
typedef struct
{
    CPRI_HASH_STATE     state;              // hash state
    HASH_STATE_TYPE     type;               // type of the context
} HASH_STATE;
```

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
{
    HASH_STATE          hashState;          // the hash state
    TPM2B_HASH_BLOCK    hmacKey;            // the HMAC key
} HMAC_STATE;
```

6.8.3.4 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)];
```

6.8.4 Loaded Object Structures

6.8.4.1 Description

The structures in clause 6.8.4 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.

6.8.4.2 OBJECT_ATTRIBUTES

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

```
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
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
48      unsigned            ppsHierarchy : 1;   //2) SET if the object belongs to PPS
49                                              //   Hierarchy
50      unsigned            spsHierarchy : 1;   //3) SET f the object belongs to SPS
51                                              //   Hierarchy
52      unsigned            evict        : 1;   //4) SET if the object is a platform or
53                                              //   owner evict object.  Platform-
54                                              //   evict object belongs to PPS
55                                              //   hierarchy, owner-evict object
56                                              //   belongs to SPS or EPS hierarchy.
57                                              //   This bit is also used to mark a
58                                              //   completed sequence object so it
59                                              //   will be flush when the
60                                              //   SequenceComplete command succeeds.
61      unsigned            primary     : 1;    //5) SET for a primary object
62      unsigned            temporary   : 1;    //6) SET for a temporary object
63      unsigned            stClear     : 1;    //7) SET for an stClear object
64      unsigned            hmacSeq     : 1;    //8) SET for an HMAC sequence object
65      unsigned            hashSeq     : 1;    //9) SET for a hash sequence object
66      unsigned            eventSeq    : 1;    //10) SET for an event sequence object
67      unsigned            ticketSafe  : 1;    //11) SET if a ticket is safe to create
68                                              //    for hash sequence object
69      unsigned            firstBlock  : 1;    //12) SET if the first block of hash
70                                              //    data has been received.  It
71                                              //    works with ticketSafe bit
72      unsigned            isParent    : 1;    //13) SET if the key has the proper
73                                              //    attributes to be a parent key
74      unsigned            privateExp  : 1;    //14) SET when the private exponent
75                                              //    of an RSA key has been validated.
76      unsigned        reserved    : 1;    //15) reserved bits. unused.
77  } OBJECT_ATTRIBUTES;
```

6.8.4.3 OBJECT Structure

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

```
78  typedef struct
79  {
80      // The attributes field is required to be first followed by the publicArea.
81      // This allows the overlay of the object structure and a sequence structure
82      OBJECT_ATTRIBUTES   attributes;         // object attributes
83      TPMT_PUBLIC         publicArea;         // public area of an object
84      TPMT_SENSITIVE      sensitive;          // sensitive area of an object
85
86  #ifdef  TPM_ALG_RSA
87      TPM2B_PUBLIC_KEY_RSA privateExponent;   // Additional field for the private
88                                              // exponent of an RSA key.
89  #endif
90      TPM2B_NAME          qualifiedName;      // object qualified name
91      TPMI_DH_OBJECT      evictHandle;        // if the object is an evict object,
92                                              // the original handle is kept here.
93                                              // The 'working' handle will be the
94                                              // handle of an object slot.
95
96      TPM2B_NAME          name;               // Name of the object name. Kept here
97                                              // to avoid repeatedly computing it.
98  } OBJECT;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

6.8.4.4 HASH_OBJECT Structure

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

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

```
typedef struct
{
    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;               // auth for use of sequence
    union
    {
        HASH_STATE      hashState[HASH_COUNT];
        HMAC_STATE      hmacState;
    }                   state;
} HASH_OBJECT;
```

6.8.4.5 ANY_OBJECT

This is the union for holding either a sequence object or a regular object.

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

6.8.5 AUTH_DUP Types

These values are used in the authorization processing.

```
typedef UINT32          AUTH_ROLE;
#define AUTH_NONE       ((AUTH_ROLE)(0))
#define AUTH_USER       ((AUTH_ROLE)(1))
#define AUTH_ADMIN      ((AUTH_ROLE)(2))
#define AUTH_DUP        ((AUTH_ROLE)(3))
```

6.8.6 Active Session Context

6.8.6.1 Description

The structures in clause 6.8.6 define the internal structure of a session context.

6.8.6.2 SESSION_ATTRIBUTES

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

```
typedef struct
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
{
    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
    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;//4) SET if the cpHash has been defined
                                            //   This attribute is not SET unless
                                            //   'isPolicy' is SET.
    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
                                            //   auth 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            requestWasBound : 1;//11) SET if the session is being used
                                            //    with the bind entity. If SET
                                            //    the authValue will not be use
                                            //    in the response HMAC computation.
    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.
} SESSION_ATTRIBUTES;
```

6.8.6.3 SESSION Structure

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

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

```
typedef struct
{
    TPM_ALG_ID          authHashAlg;        // session hash algorithm
    TPM2B_NONCE         nonceTPM;           // last TPM-generated nonce for
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
                                            // this session

    TPMT_SYM_DEF        symmetric;          // session symmetric algorithm (if any)
    TPM2B_AUTH          sessionKey;         // session secret value used for
                                            // generating HMAC and encryption keys

    SESSION_ATTRIBUTES  attributes;         // session attributes
    TPM_CC              commandCode;        // command code (policy session)
    TPMA_LOCALITY       commandLocality;    // command locality (policy session)
    UINT32              pcrCounter;         // PCR counter value when PCR is
                                            // included (policy session)
                                            // If no PCR is included, this
                                            // value is 0.

    UINT64              startTime;          // value of TPMS_CLOCK_INFO.clock when
                                            // the session was started (policy
                                            // session)

    UINT64              timeOut;            // timeout relative to
                                            // TPMS_CLOCK_INFO.clock
                                            // There is no timeout if this value
                                            // is 0.
    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

    } u1;                                    // 'boundEntity' and 'cpHash' may
                                             // share the same space to save memory

    union
    {
        TPM2B_DIGEST    auditDigest;        // audit session digest
        TPM2B_DIGEST    policyDigest;         // policyHash

    } u2;                                   // audit log and policyHash may
                                            // share space to save memory
} SESSION;
```

6.8.7 PCR

6.8.7.1 PCR_SAVE Structure

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

```
typedef struct
{
#ifdef TPM_ALG_SHA1
    BYTE                sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA256
    BYTE                sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA384
    BYTE                sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA512
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
235         BYTE                sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
236     #endif
237     #ifdef TPM_ALG_SM3_256
238         BYTE                sm3_256[NUM_STATIC_PCR][SM3_256_DIGEST_SIZE];
239     #endif
240
241         // This counter increments whenever the PCR are updated.
242         // NOTE: A platform-specific specification may designate
243         //       certain PCR changes as not causing this counter
244         //       to increment.
245         UINT32              pcrCounter;
246
247     } PCR_SAVE;
```

6.8.7.2 PCR_POLICY

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

```
248     typedef struct
249     {
250         TPMI_ALG_HASH       hashAlg[NUM_POLICY_PCR_GROUP];
251         TPM2B_DIGEST        a;
252         TPM2B_DIGEST        policy[NUM_POLICY_PCR_GROUP];
253     } PCR_POLICY;
```

6.8.7.3 PCR_AUTHVALUE

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

```
254     typedef struct
255     {
256         TPM2B_DIGEST        auth[NUM_AUTHVALUE_PCR_GROUP];
257     } PCR_AUTHVALUE;
```

6.8.8 Startup

6.8.8.1 SHUTDOWN_NONE

ISO/IEC 11889-2 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.

```
258     #define SHUTDOWN_NONE   (TPM_SU)(0xFFFF)
```

6.8.8.2 STARTUP_TYPE

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

```
259     typedef enum
260     {
261         SU_RESET,
262         SU_RESTART,
263         SU_RESUME
264     } STARTUP_TYPE;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

6.8.9 NV

6.8.9.1 NV_RESERVE

This enumeration defines the master list of the elements of a reserved portion of NV. This list includes all the pre-defined data that takes space in NV, either as persistent data or as state save data. The enumerations are used as indexes into an array of offset values. The offset values then are used to index into NV. This is method provides an imperfect analog to an actual NV implementation.

```
typedef enum
{
// Entries below mirror the PERSISTENT_DATA structure. These values are written
// to NV as individual items.
    // hierarchy
    NV_DISABLE_CLEAR,
    NV_OWNER_ALG,
    NV_ENDORSEMENT_ALG,
    NV_LOCKOUT_ALG,
    NV_OWNER_POLICY,
    NV_ENDORSEMENT_POLICY,
    NV_LOCKOUT_POLICY,
    NV_OWNER_AUTH,
    NV_ENDORSEMENT_AUTH,
    NV_LOCKOUT_AUTH,

    NV_EP_SEED,
    NV_SP_SEED,
    NV_PP_SEED,

    NV_PH_PROOF,
    NV_SH_PROOF,
    NV_EH_PROOF,

    // Time
    NV_TOTAL_RESET_COUNT,
    NV_RESET_COUNT,

    // PCR
    NV_PCR_POLICIES,
    NV_PCR_ALLOCATED,

    // Physical Presence
    NV_PP_LIST,

    // Dictionary Attack
    NV_FAILED_TRIES,
    NV_MAX_TRIES,
    NV_RECOVERY_TIME,
    NV_LOCKOUT_RECOVERY,
    NV_LOCKOUT_AUTH_ENABLED,

    // Orderly State flag
    NV_ORDERLY,

    // Command Audit
    NV_AUDIT_COMMANDS,
    NV_AUDIT_HASH_ALG,
    NV_AUDIT_COUNTER,

    // Algorithm Set
    NV_ALGORITHM_SET,

    NV_FIRMWARE_V1,
    NV_FIRMWARE_V2,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
320
321     // The entries above are in PERSISTENT_DATA. The entries below represent
322     // structures that are read and written as a unit.
323
324     // ORDERLY_DATA data structure written on each orderly shutdown
325         NV_ORDERLY_DATA,
326
327     // STATE_CLEAR_DATA structure written on each Shutdown(STATE)
328         NV_STATE_CLEAR,
329
330     // STATE_RESET_DATA structure written on each Shutdown(STATE)
331         NV_STATE_RESET,
332
333         NV_RESERVE_LAST             // end of NV reserved data list
334     } NV_RESERVE;
```

6.8.9.2 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.

```
335     typedef struct
336     {
337         TPMS_NV_PUBLIC      publicArea;
338         TPM2B_AUTH          authValue;
339     } NV_INDEX;
```

6.8.10 COMMIT_INDEX_MASK

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

```
340     #ifdef TPM_ALG_ECC
341     #define COMMIT_INDEX_MASK ((UINT16)((sizeof(gr.commitArray)*8)-1))
342     #endif
```

6.8.11 RAM Global Values

6.8.11.1 Description

The values in clause 6.8.11 are only extant in RAM. They are defined here and instanced in Global.c.

6.8.11.2 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.

```
343     extern const UINT16     g_rcIndex[15];
```

6.8.11.3 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.

```
344     extern TPM_HANDLE       g_exclusiveAuditSession;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

6.8.11.4 g_time

This value is the count of milliseconds since the TPM was powered up. This value is initialized at _TPM_Init().

```
345  extern  UINT64          g_time;
```

6.8.11.5 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.

```
346  extern BOOL             g_phEnable;
```

6.8.11.6 g_pceReConfig

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).

```
347  extern BOOL             g_pcrReConfig;
```

6.8.11.7 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().

```
348  extern TPMI_DH_OBJECT   g_DRTMHandle;
```

6.8.11.8 g_DrtmPreStartup

This value indicates that an H-CRTM occurred after _TPM_Init() but before TPM2_Startup(). The define is used to add the *g_DrtmPreStartup* value to *gp_orderlyState* at shutdown. This is a bit of a hack that was done to avoid adding another NV variable just to have a bit

```
349  extern  BOOL            g_DrtmPreStartup;
350  #define PRE_STARTUP_FLAG    0x8000
```

6.8.11.9 g_updateNV

This flag indicates if NV should be updated at the end of a command. This flag is set to FALSE at the beginning of each command in ExecuteCommand(). This flag is checked in ExecuteCommand() after the detailed actions of a command complete. If the command execution was successful and this flag is SET, any pending NV writes will be committed to NV.

```
351  extern BOOL             g_updateNV;
```

6.8.11.10 g_clearOrderly

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
352  extern BOOL             g_clearOrderly;
```

6.8.11.11 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.

```
353  extern TPM_SU           g_prevOrderlyState;
```

6.8.11.12 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.

```
354  extern BOOL             g_nvOk;
```

6.8.11.13 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 TMP vendor should not be able to know the first value but they are expected to know the second.

```
355  extern TPM2B_AUTH       g_platformUniqueAuthorities; // Reserved for RNG
356  extern TPM2B_AUTH       g_platformUniqueDetails;   // referenced by VENDOR_PERMANENT
```

6.8.12 Persistent Global Values

6.8.12.1 Description

The values in clause 6.8.12 are global values that are persistent across power events. The lifetime of the values determines the structure in which the value is placed.

6.8.12.2 PERSISTENT_DATA

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

```
357  typedef struct
358  {
359  //*****************************************************************************
360  //          Hierarchy
361  //*****************************************************************************
362  // The values in this clause are related to the hierarchies.
363
364      BOOL                disableClear;       // TRUE if TPM2_Clear() using
365                                              // lockoutAuth is disabled
366
367      // Hierarchy authPolicies
368      TPMI_ALG_HASH       ownerAlg;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
369         TPMI_ALG_HASH       endorsementAlg;
370         TPMI_ALG_HASH       lockoutAlg;
371         TPM2B_DIGEST        ownerPolicy;
372         TPM2B_DIGEST        endorsementPolicy;
373         TPM2B_DIGEST        lockoutPolicy;
374
375         // Hierarchy authValues
376         TPM2B_AUTH          ownerAuth;
377         TPM2B_AUTH          endorsementAuth;
378         TPM2B_AUTH          lockoutAuth;
379
380         // Primary Seeds
381         TPM2B_SEED          EPSeed;
382         TPM2B_SEED          SPSeed;
383         TPM2B_SEED          PPSeed;
384         // Note there is a nullSeed in the state_reset memory.
385
386         // Hierarchy proofs
387         TPM2B_AUTH          phProof;
388         TPM2B_AUTH          shProof;
389         TPM2B_AUTH          ehProof;
390         // Note there is a nullProof in the state_reset memory.
391
392     //*****************************************************************************
393     //          Reset Events
394     //*****************************************************************************
395     // A count that increments at each TPM reset and never get reset during the life
396     // time of TPM.  The value of this counter is initialized to 1 during TPM
397     // manufacture process.
398         UINT64              totalResetCount;
399
400     // This counter increments on each TPM Reset. The counter is reset by
401     // TPM2_Clear().
402         UINT32              resetCount;
403
404
405     //*****************************************************************************
406     //          PCR
407     //*****************************************************************************
408     // This structure hold the policies for those PCR that have an update policy.
409     // This implementation only supports a single group of PCR controlled by
410     // policy. If more are required, then this structure would be changed to
411     // an array.
412         PCR_POLICY          pcrPolicies;
413
414     // This structure indicates the allocation of PCR. The structure contains a
415     // list of PCR allocations for each implemented algorithm. If no PCR are
416     // allocated for an algorithm, a list entry still exists but the bit map
417     // will contain no SET bits.
418         TPML_PCR_SELECTION  pcrAllocated;
419
420     //*****************************************************************************
421     //          Physical Presence
422     //*****************************************************************************
423     // The PP_LIST type contains a bit map of the commands that require physical
424     // to be asserted when the authorization is evaluated. Physical presence will be
425     // checked if the corresponding bit in the array is SET and if the authorization
426     // handle is TPM_RH_PLATFORM.
427     //
428     // These bits may be changed with TPM2_PP_Commands().
429         BYTE                ppList[((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7)/8];
430
431     //*****************************************************************************
432     //          Dictionary attack values
433     //*****************************************************************************
434     // These values are used for dictionary attack tracking and control.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
435         UINT32              failedTries;        // the current count of unexpired
436                                                 // authorization failures
437
438         UINT32              maxTries;           // number of unexpired authorization
439                                                 // failures before the TPM is in
440                                                 // lockout
441
442         UINT32              recoveryTime;       // time between authorization failures
443                                                 // before failedTries is decremented
444
445         UINT32              lockoutRecovery;    // time that must expire between
446                                                 // authorization failures associated
447                                                 // with lockoutAuth
448
449         BOOL                lockOutAuthEnabled; // TRUE if use of lockoutAuth is
450                                                 // allowed
451
452     //*****************************************************************************
453     //           Orderly State
454     //*****************************************************************************
455     // The orderly state for current cycle
456         TPM_SU              orderlyState;
457
458     //*****************************************************************************
459     //           Command audit values.
460     //*****************************************************************************
461         BYTE                auditComands[((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8];
462         TPMI_ALG_HASH       auditHashAlg;
463         UINT64              auditCounter;
464
465     //*****************************************************************************
466     //           Algorithm selection
467     //*****************************************************************************
468     //
469     // The 'algorithmSet' value indicates the collection of algorithms that are
470     // currently in used on the TPM.  The interpretation of value is vendor dependent.
471         UINT32              algorithmSet;
472
473     //*****************************************************************************
474     //           Firmware version
475     //*****************************************************************************
476     // The firmwareV1 and firmwareV2 values are instanced in TimeStamp.c. This is
477     // a scheme used in development to allow determination of the linker build time
478     // of the TPM. An actual implementation would implement these values in a way that
479     // is consistent with vendor needs. The values are maintained in RAM for simplified
480     // access with a master version in NV.  These values are modified in a
481     // vendor-specific way.
482
483     // g_firmwareV1 contains the more significant 32-bits of the vendor version number.
484     // In the reference implementation, if this value is printed as a hex
485     // value, it will have the format of yyyymmdd
486         UINT32              firmwareV1;
487
488     // g_firmwareV1 contains the less significant 32-bits of the vendor version number.
489     // In the reference implementation, if this value is printed as a hex
490     // value, it will have the format of 00 hh mm ss
491         UINT32              firmwareV2;
492
493     } PERSISTENT_DATA;
494     extern PERSISTENT_DATA  gp;
```

6.8.12.3 ORDERLY_DATA

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
typedef struct orderly_data
{

//*****************************************************************************
//          TIME
//*****************************************************************************

// Clock has two parts. One is the state save part and one is the NV part. The
// state save version is updated on each command. When the clock rolls over, the
// NV version is updated. When the TPM starts up, if the TPM was shutdown in and
// orderly way, then the sClock value is used to initialize the clock. If the
// TPM shutdown was not orderly, then the persistent value is used and the safe
// attribute is clear.

    UINT64              clock;              // The orderly version of clock
    TPMI_YES_NO         clockSafe;          // Indicates if the clock value is
                                            // safe.
//*****************************************************************************
//          DRBG
//*****************************************************************************
#ifdef _DRBG_STATE_SAVE
    // This is DRBG state data. This is saved each time the value of clock is
    // updated.
    DRBG_STATE          drbgState;
#endif

} ORDERLY_DATA;
extern ORDERLY_DATA     go;
```

6.8.12.4 STATE_CLEAR_DATA

This structure contains the data that is saved on Shutdown(STATE). and restored on Startup(STATE). The values are set to their default settings on any Startup(Clear). In other words the data is only persistent across TPM Resume.

If the comments associated with a parameter indicate a default reset value, the value is applied on each Startup(CLEAR).

```
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
//*****************************************************************************
// The set of PCR to be saved on Shutdown(STATE)
    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;

```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
} STATE_CLEAR_DATA;
extern STATE_CLEAR_DATA gc;
```

6.8.12.5 State Reset Data

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

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

```
typedef struct state_reset_data
{
//*****************************************************************************
//          Hierarchy Control
//*****************************************************************************
    TPM2B_AUTH          nullProof;          // The proof value associated with
                                            // the TPM_RH_NULL hierarchy. The
                                            // default reset value is from the RNG.

    TPM2B_SEED          nullSeed;           // The seed value for the TPM_RN_NULL
                                            // hierarchy. The default reset value
                                            // is from the RNG.

//*****************************************************************************
//          Context
//*****************************************************************************
// The 'clearCount' counter is incremented each time the TPM successfully executes
// a TPM Resume. The counter is included in each saved context that has 'stClear'
// SET (including descendants of keys that have 'stClear' SET). This prevents these
// objects from being loaded after a TPM Resume.
// If 'clearCount' at its maximum value when the TPM receives a Shutdown(STATE),
// the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
    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.

    CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];
                                            // This is the value from which the
                                            // 'contextID' is derived. The
                                            // default reset value is {0}.


    CONTEXT_COUNTER     contextCounter;     // This array contains contains the
                                            // values used to track the version
                                            // numbers of saved contexts (see
                                            // Session.c in for details). 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
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
606  //*****************************************************************************
607
608      UINT32              restartCount;       // This counter counts TPM Restarts.
609                                              // The default reset value is 0.
610
611  //*****************************************************************************
612  //            PCR
613  //*****************************************************************************
614  // This counter increments whenever the PCR are updated. This counter is preserved
615  // across TPM Resume even though the PCR are not preserved. This is because
616  // sessions remain active across TPM Restart and the count value in the session
617  // is compared to this counter so this counter must have values that are unique
618  // as long as the sessions are active.
619  // NOTE: A platform-specific specification may designate that certain PCR changes
620  //       do not increment this counter to increment.
621      UINT32              pcrCounter;         // The default reset value is 0.
622
623  #ifdef TPM_ALG_ECC
624
625  //*****************************************************************************
626  //         ECDAA
627  //*****************************************************************************
628      UINT64              commitCounter;      // This counter increments each time
629                                              // TPM2_Commit() returns
630                                              // TPM_RC_SUCCESS. The default reset
631                                              // value is 0.
632
633
634      TPM2B_NONCE         commitNonce;        // This random value is used to compute
635                                              // the commit values. The default reset
636                                              // value is from the RNG.
637
638  // This implementation relies on the number of bits in g_commitArray being a
639  // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
640      BYTE                commitArray[16];    // The default reset value is {0}.
641
642  #endif //TPM_ALG_ECC
643
644  } STATE_RESET_DATA;
645  extern STATE_RESET_DATA gr;
```

6.8.13 Global Macro Definitions

This macro is used to ensure that a handle, session, or parameter number is only added if the response code is FMT1.

```
646  #define RcSafeAddToResult(r, v) \
647      ((r) + (((r) & RC_FMT1) ? (v) : 0))
```

This macro is used when a parameter is not otherwise referenced in a function. This macro is normally not used by itself but is paired with a pAssert() within a #ifdef *pAssert*. If *pAssert* is not defined, then a parameter might not otherwise be referenced. This macro **uses** the parameter from the perspective of the compiler so it doesn't complain.

```
648  #define UNREFERENCED(a) ((void)(a))
```

6.9 Private data

```
649  #if defined SESSION_PROCESS_C || defined GLOBAL_C || defined MANUFACTURE_C
```

From SessionProcess.c

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

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

```
650   extern TPM_HANDLE       s_sessionHandles[MAX_SESSION_NUM];
```

Array of authorization session attributes

```
651   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

```
652   extern TPM_HANDLE       s_associatedHandles[MAX_SESSION_NUM];
```

Array of nonces provided by the caller for the corresponding sessions

```
653   extern TPM2B_NONCE      s_nonceCaller[MAX_SESSION_NUM];
```

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

```
654   extern TPM2B_AUTH       s_inputAuthValues[MAX_SESSION_NUM];
```

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter

```
656   extern UINT32           s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
657   extern UINT32           s_decryptSessionIndex;
```

Index of a session used for audit

```
658   extern UINT32           s_auditSessionIndex;
```

The *cpHash* for an audit session

```
659   extern TPM2B_DIGEST     s_cpHashForAudit;
```

The *cpHash* for command audit

```
660   #ifdef  TPM_CC_GetCommandAuditDigest
661   extern TPM2B_DIGEST    s_cpHashForCommandAudit;
662   #endif
```

Number of authorization sessions present in the command

```
663   extern UINT32           s_sessionNum;
```

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

```
664   extern BOOL             s_DAPendingOnNV;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
665  #endif // SESSION_PROCESS_C
666  #if defined DA_C || defined GLOBAL_C || defined MANUFACTURE_C
```

From DA.c

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

```
667  extern UINT64       s_selfHealTimer;
```

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

```
668  extern UINT64       s_lockoutTimer;
669  #endif // DA_C
670  #if defined NV_C || defined GLOBAL_C
```

From NV.c

List of pre-defined address of reserved data

```
671  extern UINT32       s_reservedAddr[NV_RESERVE_LAST];
```

List of pre-defined reserved data size in byte

```
672  extern UINT32       s_reservedSize[NV_RESERVE_LAST];
```

Size of data in RAM index buffer

```
673  extern UINT32       s_ramIndexSize;
```

Reserved RAM space for frequently updated NV Index. The data layout in ram buffer is {NV_handle(), size of data, data} for each NV index data stored in RAM

```
674  extern BYTE      s_ramIndex[RAM_INDEX_SPACE];
```

Address of size of RAM index space in NV

```
675  extern UINT32   s_ramIndexSizeAddr;
```

Address of NV copy of RAM index space

```
676  extern UINT32   s_ramIndexAddr;
```

Address of maximum counter value; an auxiliary variable to implement NV counters

```
677  extern UINT32   s_maxCountAddr;
```

Beginning of NV dynamic area; starts right after the *s_maxCountAddr* and *s_evictHandleMapAddr* variables

```
678  extern UINT32   s_evictNvStart;
```

Beginning of NV dynamic area; also the beginning of the predefined reserved data area.

```
679  extern UINT32   s_evictNvEnd;
```

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
680   extern TPM_RC   s_NvStatus;
681   #endif
682   #if defined OBJECT_C || defined GLOBAL_C
```

From Object.c

This type is the container for an object.

```
683   typedef struct
684   {
685       BOOL        occupied;
686       ANY_OBJECT      object;
687   } OBJECT_SLOT;
```

This is the memory that holds the loaded objects.

```
688   extern OBJECT_SLOT     s_objects[MAX_LOADED_OBJECTS];
689   #endif // OBJECT_C
690   #if defined PCR_C || defined GLOBAL_C
```

From PCR.c

```
691   typedef struct
692   {
693   #ifdef TPM_ALG_SHA1
694       // SHA1 PCR
695       BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
696   #endif
697   #ifdef TPM_ALG_SHA256
698       // SHA256 PCR
699       BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
700   #endif
701   #ifdef TPM_ALG_SHA384
702       // SHA384 PCR
703       BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
704   #endif
705   #ifdef TPM_ALG_SHA512
706       // SHA512 PCR
707       BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
708   #endif
709   #ifdef TPM_ALG_SM3_256
710       // SHA256 PCR
711       BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
712   #endif
713   } PCR;
714   typedef struct
715   {
716       unsigned int    stateSave : 1;              // if the PCR value should be
717                                                   // saved in state save
718       unsigned int    resetLocality : 5;          // The locality that the PCR
719                                                   // can be reset
720       unsigned int    extendLocality : 5;         // The locality that the PCR
721                                                   // can be extend
722   } PCR_Attributes;
723   extern PCR          s_pcrs[IMPLEMENTATION_PCR];
724   #endif // PCR_C
725   #if defined SESSION_C || defined GLOBAL_C
```

From Session.c

Container for HMAC or policy session tracking information

```
726   typedef struct
727   {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
728         BOOL                occupied;
729         SESSION             session;        // session structure
730     } SESSION_SLOT;
731     extern SESSION_SLOT     s_sessions[MAX_LOADED_SESSIONS];
```

The index in *conextArray* 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.

```
732     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)

```
733     extern int              s_freeSessionSlots;
734     #endif // SESSION_C
```

From Manufacture.c

```
735     extern BOOL             g_manufactured;
736     #if defined POWER_C || defined GLOBAL_C
```

From Power.c

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.

```
737     extern BOOL             s_initialized;
738     #endif // POWER_C
739     #if defined MEMORY_LIB_C || defined GLOBAL_C
```

The *s_actionOutputBuffer* should not be modifiable by the host system until the TPM has returned a response code. The *s_actionOutputBuffer* should not be accessible until response parameter encryption, if any, is complete.

```
740     extern UINT32   s_actionInputBuffer[1024];          // action input buffer
741     extern UINT32   s_actionOutputBuffer[1024];         // action output buffer
742     extern BYTE     s_responseBuffer[MAX_RESPONSE_SIZE];// response buffer
743     #endif // MEMORY_LIB_C
```

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.

```
744     extern jmp_buf    g_jumpBuffer;         // the jump buffer
745     extern BOOL       g_inFailureMode;      // Indicates that the TPM is in failure mode
746     extern BOOL       g_forceFailureMode;   // flag to force failure mode during test
747     #if defined TPM_FAIL_C || defined GLOBAL_C || 1
748     extern UINT32     s_failFunction;
749     extern UINT32     s_failLine;           // the line in the file at which
750                                             // the error was signaled
751     extern UINT32     s_failCode;           // the error code used
752     #endif // TPM_FAIL_C
753     #endif // GLOBAL_H
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

6.10 Tpm.h

Root header file for building any TPM. lib code

```
#ifndef     _TPM_H
#define     _TPM_H
#include    "bool.h"
#include    "Implementation.h"
#include    "TPM_Types.h"
#include    "swap.h"
#endif
```

6.11 swap.h

```
#ifndef _SWAP_H
#define _SWAP_H
#include "Implementation.h"
#if    NO_AUTO_ALIGN == YES || LITTLE_ENDIAN_TPM == YES
```

The aggregation macros for machines that do not allow unaligned access or for little-endian machines. Aggregate bytes into an UINT

```
#define BYTE_ARRAY_TO_UINT8(b)   (UINT8)((b)[0])
#define BYTE_ARRAY_TO_UINT16(b)  (UINT16)(  ((b)[0] <<  8) \
                                          +  (b)[1])
#define BYTE_ARRAY_TO_UINT32(b)  (UINT32)(  ((b)[0] << 24) \
                                          + ((b)[1] << 16) \
                                          + ((b)[2] << 8 ) \
                                          +  (b)[3])
#define BYTE_ARRAY_TO_UINT64(b)  (UINT64)(  ((UINT64)(b)[0] << 56) \
                                          + ((UINT64)(b)[1] << 48) \
                                          + ((UINT64)(b)[2] << 40) \
                                          + ((UINT64)(b)[3] << 32) \
                                          + ((UINT64)(b)[4] << 24) \
                                          + ((UINT64)(b)[5] << 16) \
                                          + ((UINT64)(b)[6] <<  8) \
                                          +  (UINT64)(b)[7])
```

Disaggregate a UINT into a byte array

```
#define UINT8_TO_BYTE_ARRAY(i, b)     ((b)[0] = (BYTE)(i), i)
#define UINT16_TO_BYTE_ARRAY(i, b)    ((b)[0] = (BYTE)((i) >>  8), \
                                       (b)[1] = (BYTE) (i),       \
                                       (i))
#define UINT32_TO_BYTE_ARRAY(i, b)    ((b)[0] = (BYTE)((i) >> 24), \
                                       (b)[1] = (BYTE)((i) >> 16), \
                                       (b)[2] = (BYTE)((i) >>  8), \
                                       (b)[3] = (BYTE) (i),       \
                                       (i))
#define UINT64_TO_BYTE_ARRAY(i, b)    ((b)[0] = (BYTE)((i) >> 56), \
                                       (b)[1] = (BYTE)((i) >> 48), \
                                       (b)[2] = (BYTE)((i) >> 40), \
                                       (b)[3] = (BYTE)((i) >> 32), \
                                       (b)[4] = (BYTE)((i) >> 24), \
                                       (b)[5] = (BYTE)((i) >> 16), \
                                       (b)[6] = (BYTE)((i) >>  8), \
                                       (b)[7] = (BYTE) (i),       \
                                       (i))
#else
```

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
39  #define BYTE_ARRAY_TO_UINT8(b)        *((UINT8  *)(b))
40  #define BYTE_ARRAY_TO_UINT16(b)       *((UINT16 *)(b))
41  #define BYTE_ARRAY_TO_UINT32(b)       *((UINT32 *)(b))
42  #define BYTE_ARRAY_TO_UINT64(b)       *((UINT64 *)(b))
```

Disaggregate a UINT into a byte array

```
43  #define UINT8_TO_BYTE_ARRAY(i, b)   (*((UINT8  *)(b)) = (i))
44  #define UINT16_TO_BYTE_ARRAY(i, b)  (*((UINT16 *)(b)) = (i))
45  #define UINT32_TO_BYTE_ARRAY(i, b)  (*((UINT32 *)(b)) = (i))
46  #define UINT64_TO_BYTE_ARRAY(i, b)  (*((UINT64 *)(b)) = (i))
47  #endif  // NO_AUTO_ALIGN == YES
48  #endif  // _SWAP_H
```

6.12 InternalRoutines.h

```
1   #ifndef     INTERNAL_ROUTINES_H
2   #define     INTERNAL_ROUTINES_H
```

NULL definition

```
3   #ifndef         NULL
4   #define         NULL        (0)
5   #endif
```

UNUSED_PARAMETER

```
6   #ifndef         UNUSED_PARAMETER
7   #define         UNUSED_PARAMETER(param)     (void)(param);
8   #endif
```

Internal data definition

```
9   #include "Global.h"
10  #include "VendorString.h"
```

Error Reporting

```
11  #include "TpmError.h"
```

DRTM functions

```
12  #include "_TPM_Hash_Start_fp.h"
13  #include "_TPM_Hash_Data_fp.h"
14  #include "_TPM_Hash_End_fp.h"
```

Internal subsystem functions

```
15  #include "Object_fp.h"
16  #include "Entity_fp.h"
17  #include "Session_fp.h"
18  #include "Hierarchy_fp.h"
19  #include "NV_fp.h"
20  #include "PCR_fp.h"
21  #include "DA_fp.h"
22  #include "TpmFail_fp.h"
```

Internal support functions

```
23  #include "CommandCodeAttributes_fp.h"
24  #include "MemoryLib_fp.h"
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
#include "marshal_fp.h"
#include "Time_fp.h"
#include "Locality_fp.h"
#include "PP_fp.h"
#include "CommandAudit_fp.h"
#include "Manufacture_fp.h"
#include "Power_fp.h"
#include "Handle_fp.h"
#include "Commands_fp.h"
#include "AlgorithmCap_fp.h"
#include "PropertyCap_fp.h"
#include "Bits_fp.h"
```

Internal crypto functions

```
#include "Ticket_fp.h"
#include "CryptUtil_fp.h"
#include "CryptSelfTest_fp.h"
#endif
```

6.13 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.

```
#ifndef _TPM_BUILD_SWITCHES_H
#define _TPM_BUILD_SWITCHES_H
#define SIMULATION
#define FIPS_COMPLIANT
```

Define the alignment macro appropriate for the build environment For MS C compiler

```
#define ALIGN_TO(boundary)  __declspec(align(boundary))
```

For ISO 9899:2011

```
// #define ALIGN_TO(boundary)  _Alignas(boundary)
```

This switch enables the RNG state save and restore

```
#undef  _DRBG_STATE_SAVE
#define _DRBG_STATE_SAVE        // Comment this out if no state save is wanted
```

Set the alignment size for the crypto. It would be nice to set this according to macros automatically defined by the build environment, but that doesn't seem possible because there isn't any simple set for that. So, this is just a plugged value. Your compiler should complain if this alignment isn't possible.

NOTE This value can be set at the command line or just plugged in here.

```
#ifdef CRYPTO_ALIGN_16
#   define CRYPTO_ALIGNMENT     16
#elif defined CRYPTO_ALIGN_8
#   define CRYPTO_ALIGNMENT     8
#eliF defined CRYPTO_ALIGN_2
#   define  CRYPTO_ALIGNMENT    2
#elif defined CRTYPO_ALIGN_1
#   define  CRYPTO_ALIGNMENT    1
#else
#   define CRYPTO_ALIGNMENT     4    // For 32-bit builds
#endif
#define CRYPTO_ALIGNED  ALIGN_TO(CRYPTO_ALIGNMENT)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

This macro is used to handle LIB_EXPORT of function and variable names in lieu of a . def file

```
#define LIB_EXPORT __declspec(dllexport)
// #define LIB_EXPORT
```

For import of a variable

```
#define LIB_IMPORT __declspec(dllimport)
//#define LIB_IMPORT
```

This is defined to indicate a function that does not return. This is used in static code anlaysis.

```
#define __declspec(noreturn)
//#define #ifdef SELF_TEST
#pragma comment(lib, "algorithmtests.lib")
#endif
```

The switches in this group can only be enabled when running a simulation

```
#ifdef SIMULATION
#   define RSA_KEY_CACHE
#   define TPM_RNG_FOR_DEBUG
#else
#   undef RSA_KEY_CACHE
#   undef TPM_RNG_FOR_DEBUG
#endif  // SIMULATION
#define INLINE  __inline
#endif // _TPM_BUILD_SWITCHES_H
```

6.14 VendorString.h

```
#ifndef     _VENDOR_STRING_H
#define     _VENDOR_STRING_H
```

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

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

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

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

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

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

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

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

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

```
#define   FIRMWARE_V1         (0x20130315)
```

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

```
#define   FIRMWARE_V2         (0x00120000)
```

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

```
#ifndef FIRMWARE_V1
#error  FIRMWARE_V1 is not provided. \
Please modify include\VendorString.h to provide a vendor specific firmware \
version
#endif
#endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

7 Main

7.1 CommandDispatcher()

In the reference implementation, a program that uses ISO/IEC 11889-3 as input automatically generates the command dispatch code. The function prototype header file (CommandDispatcher_fp.h) is shown here.

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;
- 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.

```
#ifndef    COMMANDDISPATCHER_FP_H
#define    COMMANDDISPATCHER_FP_H
TPM_RC
CommandDispatcher(
    TPMI_ST_COMMAND_TAG     tag,    // IN: Input command tag
    TPM_CC      command_code,       // IN: Command code
    INT32       *parm_buffer_size,  // IN: size of parameter buffer
    BYTE        *parm_buffer_start, // IN: pointer to start of parameter buffer
    TPM_HANDLE  handles[],          // IN: handle array
    UINT32      *res_handle_size,   // OUT: size of handle buffer in response
    UINT32      *res_parm_size      // OUT: size of parameter buffer in response
    );
#endif
```

7.2 ExecCommand.c

7.2.1 Introduction

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

7.2.2 Includes

```
#include "InternalRoutines.h"
#include "HandleProcess_fp.h"
#include "SessionProcess_fp.h"
#include "CommandDispatcher_fp.h"
```

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

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

7.2.3 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

d) Calls ParseSessionBuffer() () to:

1) unmarshal and parse the session area;

2) check the authorizations; and

3) when necessary, decrypt a parameter.

e) Calls CommandDispatcher() to:

1) unmarshal the command parameters from the command buffer;

2) call the routine that performs the command actions; and

3) marshal the responses into the response buffer.

f) If any error occurs in any of the steps above create the error response and return.

g) Calls BuildResponseSession() to:

1) when necessary, encrypt a parameter

2) build the response authorization sessions

3) update the audit sessions and nonces

h) Assembles handle, parameter and session buffers for response and return.

```
LIB_EXPORT void
ExecuteCommand(
    unsigned int     requestSize,   // IN: command buffer size
    unsigned char   *request,       // IN: command buffer
    unsigned int    *responseSize,  // OUT: response buffer size
    unsigned char   **response      // OUT: response buffer
    )
{
    // Command local variables
    TPM_ST               tag;               // these first three variables are the
    UINT32               commandSize;
    TPM_CC               commandCode = 0;

    BYTE                *parmBufferStart;   // pointer to the first byte of an
                                            // optional parameter buffer

    UINT32               parmBufferSize = 0;// number of bytes in parameter area

    UINT32               handleNum = 0;     // number of handles unmarshaled into
                                            // the handles array

    TPM_HANDLE           handles[MAX_HANDLE_NUM];// array to hold handles in the
                                            // command. Only handles in the handle
                                            // area are stored here, not handles
                                            // passed as parameters.

    // Response local variables
    TPM_RC               result;            // return code for the command

    TPM_ST               resTag;            // tag for the response

    UINT32               resHandleSize = 0; // size of the handle area in the
                                            // response. This is needed so that the
                                            // handle area can be skipped when
                                            // generating the rpHash.

    UINT32               resParmSize = 0;   // the size of the response parameters
                                            // These values go in the rpHash.

    UINT32               resAuthSize = 0;   // size of authorization area in the
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
                                          // response

    INT32               size;             // remaining data to be unmarshaled
                                          // or remaining space in the marshaling
                                          // buffer

    BYTE               *buffer;           // pointer into the buffer being used
                                          // for marshaling or unmarshaling

    UINT32              i;                 // local temp

// 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, commandCode, 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 NB would be written on each call.
    g_updateNV = FALSE;
    g_clearOrderly = FALSE;


    // As of Sept 25, 2013, the failure mode handling has been incorporated in the
    // reference code. This implementation requires that the system support
    // setjmp/longjmp. This code is put here because of the complexity being
    // added to the platform and simulator code to deal with all the variations
    // of errors.
    if(g_inFailureMode)
    {
        // Do failure mode processing
        TpmFailureMode (requestSize, request, responseSize, response);
        return;
    }
    if(setjmp(g_jumpBuffer) != 0)
    {
        // Get here if we got a longjump putting us into failure mode
        g_inFailureMode = TRUE;
        result = TPM_RC_FAILURE;
        goto Fail;
    }

    // Assume that everything is going to work.
    result = TPM_RC_SUCCESS;


    // 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.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
112         if(g_DRTMHandle != TPM_RH_UNASSIGNED)
113             ObjectTerminateEvent();
114 
115         // Get command buffer size and command buffer.
116         size = requestSize;
117         buffer = request;
118 
119         // Parse command header: tag, commandSize and commandCode.
120         // First parse the tag. The unmarshaling routine will validate
121         // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
122         result = TPMI_ST_COMMAND_TAG_Unmarshal(&tag, &buffer, &size);
123         if(result != TPM_RC_SUCCESS)
124             goto Cleanup;
125 
126         // Unmarshal the commandSize indicator.
127         result = UINT32_Unmarshal(&commandSize, &buffer, &size);
128         if(result != TPM_RC_SUCCESS)
129             goto Cleanup;
130 
131         // On a TPM that receives bytes on a port, the number of bytes that were
132         // received on that port is requestSize it must be identical to commandSize.
133         // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
134         // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
135         // as the input processing (the function that receives the command bytes and
136         // places them in the input buffer) would likely have the input truncated when
137         // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
138         if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
139         {
140             result = TPM_RC_COMMAND_SIZE;
141             goto Cleanup;
142         }
143 
144         // Unmarshal the command code.
145         result = TPM_CC_Unmarshal(&commandCode, &buffer, &size);
146         if(result != TPM_RC_SUCCESS)
147             goto Cleanup;
148 
149         // Check to see if the command is implemented.
150         if(!CommandIsImplemented(commandCode))
151         {
152             result = TPM_RC_COMMAND_CODE;
153             goto Cleanup;
154         }
155 
156     #if  FIELD_UPGRADE_IMPLEMENTED  == YES
157         // If the TPM is in FUM, then the only allowed command is
158         // TPM_CC_FieldUpgradeData.
159         if(IsFieldUpgradeMode() && (commandCode != TPM_CC_FieldUpgradeData))
160         {
161             result = TPM_RC_UPGRADE;
162             goto Cleanup;
163         }
164         else
165     #endif
166             // Excepting FUM, the TPM only accepts TPM2_Startup() after
167             // _TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
168             // is no longer allowed.
169             if((   !TPMIsStarted() && commandCode != TPM_CC_Startup)
170                 || (TPMIsStarted() && commandCode == TPM_CC_Startup))
171             {
172                 result = TPM_RC_INITIALIZE;
173                 goto Cleanup;
174             }
175 
176         // Start regular command process.
177         // Parse Handle buffer.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
178         result = ParseHandleBuffer(commandCode, &buffer, &size, handles, &handleNum);
179         if(result != TPM_RC_SUCCESS)
180             goto Cleanup;
181
182         // Number of handles retrieved from handle area should be less than
183         // MAX_HANDLE_NUM.
184         pAssert(handleNum <= MAX_HANDLE_NUM);
185
186         // All handles in the handle area are required to reference TPM-resident
187         // entities.
188         for(i = 0; i < handleNum; i++)
189         {
190             result = EntityGetLoadStatus(&handles[i], commandCode);
191             if(result != TPM_RC_SUCCESS)
192             {
193                 if(result == TPM_RC_REFERENCE_H0)
194                     result = result + i;
195                 else
196                     result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
197                 goto Cleanup;
198             }
199         }
200
201         // Authorization session handling for the command.
202         if(tag == TPM_ST_SESSIONS)
203         {
204             BYTE        *sessionBufferStart;// address of the session area first byte
205                                             // in the input buffer
206
207             UINT32      authorizationSize;  // number of bytes in the session area
208
209             // Find out session buffer size.
210             result = UINT32_Unmarshal(&authorizationSize, &buffer, &size);
211             if(result != TPM_RC_SUCCESS)
212                 goto Cleanup;
213
214             // Perform sanity check on the unmarshaled value. If it is smaller than
215             // the smallest possible session or larger than the remaining size of
216             // the command, then it is an error. NOTE: This check could pass but the
217             // session size could still be wrong. That will be determined after the
218             // sessions are unmarshaled.
219             if(   authorizationSize < 9
220                || authorizationSize > (UINT32) size)
221             {
222                 result = TPM_RC_SIZE;
223                 goto Cleanup;
224             }
225
226             // The sessions, if any, follows authorizationSize.
227             sessionBufferStart = buffer;
228
229             // The parameters follow the session area.
230             parmBufferStart = sessionBufferStart + authorizationSize;
231
232             // Any data left over after removing the authorization sessions is
233             // parameter data. If the command does not have parameters, then an
234             // error will be returned if the remaining size is not zero. This is
235             // checked later.
236             parmBufferSize = size - authorizationSize;
237
238             // The actions of ParseSessionBuffer() are described in the introduction.
239             result = ParseSessionBuffer(commandCode,
240                                         handleNum,
241                                         handles,
242                                         sessionBufferStart,
243                                         authorizationSize,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
                                    parmBufferStart,
                                    parmBufferSize);
        if(result != TPM_RC_SUCCESS)
            goto Cleanup;
    }
    else
    {
        // Whatever remains in the input buffer is used for the parameters of the
        // command.
        parmBufferStart = buffer;
        parmBufferSize = size;

        // The command has no authorization sessions.
        // If the command requires authorizations, then CheckAuthNoSession() will
        // return an error.
        result = CheckAuthNoSession(commandCode, handleNum, handles,
                                    parmBufferStart, parmBufferSize);
        if(result != TPM_RC_SUCCESS)
            goto Cleanup;
    }

    // CommandDispatcher returns a response handle buffer and a response parameter
    // buffer if it succeeds. It will also set the parameterSize field in the
    // buffer if the tag is TPM_RC_SESSIONS.
    result = CommandDispatcher(tag,
                               commandCode,
                               (INT32 *) &parmBufferSize,
                               parmBufferStart,
                               handles,
                               &resHandleSize,
                               &resParmSize);
    if(result != TPM_RC_SUCCESS)
        goto Cleanup;

    // Build the session area at the end of the parameter area.
    BuildResponseSession(tag,
                         commandCode,
                         resHandleSize,
                         resParmSize,
                         &resAuthSize);

Cleanup:
    // This implementation loads an "evict" object to a transient object slot in
    // RAM whenever an "evict" object handle is used in a command so that the
    // access to any object is the same. These temporary objects need to be
    // cleared from RAM whether the command succeeds or fails.
    ObjectCleanupEvict();

Fail:
    // The response will contain at least a response header.
    *responseSize = sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC);

    // If the command completed successfully, then build the rest of the response.
    if(result == TPM_RC_SUCCESS)
    {
        // Outgoing tag will be the same as the incoming tag.
        resTag = tag;
        // The overall response will include the handles, parameters,
        // and authorizations.
        *responseSize += resHandleSize + resParmSize + resAuthSize;

        // Adding parameter size field.
        if(tag == TPM_ST_SESSIONS)
            *responseSize += sizeof(UINT32);

        if(   g_clearOrderly == TRUE
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
310             && gp.orderlyState != SHUTDOWN_NONE)
311         {
312             gp.orderlyState = SHUTDOWN_NONE;
313             NvWriteReserved(NV_ORDERLY, &gp.orderlyState);
314             g_updateNV = TRUE;
315         }
316     }
317     else
318     {
319         // The command failed.
320         // If this was a failure due to a bad command tag, then need to return
321         // an ISO/IEC 11889 (first edition) compatible response
322         if(result == TPM_RC_BAD_TAG)
323             resTag = TPM_ST_RSP_COMMAND;
324         else
325             // return ISO/IEC 11889 compatible response
326             resTag = TPM_ST_NO_SESSIONS;
327     }
328     // Try to commit all the writes to NV if any NV write happened during this
329     // command execution. This check should be made for both succeeded and failed
330     // commands, because a failed one may trigger a NV write in DA logic as well.
331     // This is the only place in the command execution path that may call the NV
332     // commit. If the NV commit fails, the TPM should be put in failure mode.
333     if(g_updateNV && !g_inFailureMode)
334     {
335         g_updateNV = FALSE;
336         if(!NvCommit())
337             FAIL(FATAL_ERROR_INTERNAL);
338     }
339
340     // Marshal the response header.
341     buffer = MemoryGetResponseBuffer(commandCode);
342     TPM_ST_Marshal(&resTag, &buffer, NULL);
343     UINT32_Marshal((UINT32 *)responseSize, &buffer, NULL);
344     pAssert(*responseSize <= MAX_RESPONSE_SIZE);
345     TPM_RC_Marshal(&result, &buffer, NULL);
346
347     *response = MemoryGetResponseBuffer(commandCode);
348
349     // Clear unused bit in response buffer.
350     MemorySet(*response + *responseSize, 0, MAX_RESPONSE_SIZE - *responseSize);
351
352     return;
353 }
```

7.3 ParseHandleBuffer()

In the reference implementation, the routine for unmarshaling the command handles is automatically generated from ISO/IEC 11889-3 command tables. The prototype header file (HandleProcess_fp.h) is shown here.

```
1  #ifndef    HANDLEPROCESS_FP_H
2  #define    HANDLEPROCESS_FP_H
3  TPM_RC
4  ParseHandleBuffer(
5      TPM_CC      command_code,
6      BYTE        **handle_buffer_start,
7      INT32       *buffer_remain_size,
8      TPM_HANDLE  handles[],
9      UINT32      *handle_num
10     );
11 #endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

7.4 SessionProcess.c

7.4.1 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.

7.4.2 Includes and Data Definitions

```
#define SESSION_PROCESS_C
#include "InternalRoutines.h"
#include "SessionProcess_fp.h"
#include "Platform.h"
```

7.4.3 Authorization Support Functions

7.4.3.1 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 1

Return Value	Meaning
TRUE	handle is exempted from DA logic
FALSE	handle is not exempted from DA logic

```
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:
        {
            OBJECT      *object;
            object = ObjectGet(handle);
            result = (object->publicArea.objectAttributes.noDA == SET);
            break;
        }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
29          case TPM_HT_NV_INDEX:
30          {
31              NV_INDEX        nvIndex;
32              NvGetIndexInfo(handle, &nvIndex);
33              result = (nvIndex.publicArea.attributes.TPMA_NV_NO_DA == SET);
34              break;
35          }
36          case TPM_HT_PCR:
37              // PCRs are always exempted from DA.
38              result = TRUE;
39              break;
40          default:
41              break;
42      }
43      return result;
44  }
```

7.4.3.2 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 2

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

```
45  static TPM_RC
46  IncrementLockout(
47      UINT32          sessionIndex
48      )
49  {
50      TPM_HANDLE      handle = s_associatedHandles[sessionIndex];
51      TPM_HANDLE      sessionHandle = s_sessionHandles[sessionIndex];
52      TPM_RC          result;
53      SESSION        *session = NULL;
54
55
56      // Don't increment lockout unless the handle associated with the session
57      // is DA protected or the session is bound to a DA protected entity.
58      if(sessionHandle == TPM_RS_PW)
59      {
60          if(IsDAExempted(handle))
61              return TPM_RC_BAD_AUTH;
62
63      }
64      else
65      {
66          session = SessionGet(sessionHandle);
67          // If the session is bound to lockout, then use that as the relevant
68          // handle. This means that an auth failure with a bound session
69          // bound to lockoutAuth will take precedence over any other
70          // lockout check
71          if(session->attributes.isLockoutBound == SET)
72              handle = TPM_RH_LOCKOUT;
73
74          if(   session->attributes.isDaBound == CLEAR
75             && IsDAExempted(handle)
76            )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
77              // If the handle was changed to TPM_RH_LOCKOUT, this will not return
78              // TPM_RC_BAD_AUTH
79               return TPM_RC_BAD_AUTH;
80
81      }
82
83      if(handle == TPM_RH_LOCKOUT)
84      {
85          pAssert(gp.lockOutAuthEnabled);
86          gp.lockOutAuthEnabled = FALSE;
87          // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
88          // the lockout auth will be reset at startup.
89          if(gp.lockoutRecovery != 0)
90          {
91              result = NvIsAvailable();
92              if(result != TPM_RC_SUCCESS)
93              {
94                  // No NV access for now. Put the TPM in pending mode.
95                  s_DAPendingOnNV = TRUE;
96              }
97              else
98              {
99                  // Update NV.
100                 NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
101                 g_updateNV = TRUE;
102             }
103         }
104     }
105     else
106     {
107         if(gp.recoveryTime != 0)
108         {
109             gp.failedTries++;
110             result = NvIsAvailable();
111             if(result != TPM_RC_SUCCESS)
112             {
113                 // No NV access for now.  Put the TPM in pending mode.
114                 s_DAPendingOnNV = TRUE;
115             }
116             else
117             {
118                 // Record changes to NV.
119                 NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
120                 g_updateNV = TRUE;
121             }
122         }
123     }
124
125     // Register a DA failure and reset the timers.
126     DARegisterFailure(handle);
127
128     return TPM_RC_AUTH_FAIL;
129 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

7.4.3.3 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.

© ISO/IEC 2015 – All rights reserved

Table 3

Return Value	Meaning
TRUE	handle points to the session start entity
FALSE	handle does not point to the session start entity

```
130  static BOOL
131  IsSessionBindEntity(
132      TPM_HANDLE       associatedHandle,  // IN: handle to be authorized
133      SESSION         *session            // IN: associated session
134      )
135  {
136      TPM2B_NAME     entity;             // The bind value for the entity
137
138      // If the session is not bound, return FALSE.
139      if(!session->attributes.isBound)
140          return FALSE;
141
142      // Compute the bind value for the entity.
143      SessionComputeBoundEntity(associatedHandle, &entity);
144
145      // Compare to the bind value in the session.
146      session->attributes.requestWasBound =
147              Memory2BEqual(&entity.b, &session->u1.boundEntity.b);
148      return session->attributes.requestWasBound;
149  }
```

7.4.3.4 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 4

Return Value	Meaning
TRUE	policy session is required
FALSE	policy session is not required

```
150  static BOOL
151  IsPolicySessionRequired(
152      TPM_CC           commandCode,   // IN: command code
153      UINT32           sessionIndex   // IN: session index
154      )
155  {
156      AUTH_ROLE       role = CommandAuthRole(commandCode, sessionIndex);
157      TPM_HT          type = HandleGetType(s_associatedHandles[sessionIndex]);
158
159      if(role == AUTH_DUP)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
160             return TRUE;
161
162         if(role == AUTH_ADMIN)
163         {
164             if(type == TPM_HT_TRANSIENT)
165             {
166                 OBJECT      *object = ObjectGet(s_associatedHandles[sessionIndex]);
167
168                 if(object->publicArea.objectAttributes.adminWithPolicy == CLEAR)
169                     return FALSE;
170             }
171             return TRUE;
172         }
173
174         if(type == TPM_HT_PCR)
175         {
176             if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
177             {
178                 TPM2B_DIGEST        policy;
179                 TPMI_ALG_HASH       policyAlg;
180                 policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
181                                              &policy);
182                 if(policyAlg != TPM_ALG_NULL)
183                     return TRUE;
184             }
185         }
186         return FALSE;
187     }
```

7.4.3.5 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 auth, 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 5

Return Value	Meaning
TRUE	*authValue* is available
FALSE	*authValue* is not available

```
188     static BOOL
189     IsAuthValueAvailable(
190         TPM_HANDLE      handle,         // IN: handle of entity
191         TPM_CC          commandCode,    // IN: commandCode
192         UINT32          sessionIndex    // IN: session index
193         )
194     {
195         BOOL            result = FALSE;
196         // If a policy session is required, the entity can not be authorized by
197         // authValue. However, at this point, the policy session requirement should
198         // already have been checked.
199         pAssert(!IsPolicySessionRequired(commandCode, sessionIndex));
200
201         switch(HandleGetType(handle))
202         {
203             case TPM_HT_PERMANENT:
204                 switch(handle)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
205             {
206                     // At this point hierarchy availability has already been
207                     // checked so primary seed handles are always available here
208                 case TPM_RH_OWNER:
209                 case TPM_RH_ENDORSEMENT:
210                 case TPM_RH_PLATFORM:
211 #ifdef VENDOR_PERMANENT
212                     // This vendor defined handle associated with the
213                     // manufacturer's shared secret
214                 case VENDOR_PERMANENT:
215 #endif
216                     // NullAuth is always available.
217                 case TPM_RH_NULL:
218                      // At the point when authValue availability is checked, control
219                     // path has already passed the DA check so LockOut auth is
220                     // always available here
221                 case TPM_RH_LOCKOUT:
222 
223                     result = TRUE;
224                     break;
225                 default:
226                     // Otherwise authValue is not available.
227                     break;
228             }
229             break;
230         case TPM_HT_TRANSIENT:
231             // A persistent object has already been loaded and the internal
232             // handle changed.
233             {
234                 OBJECT          *object;
235                 object = ObjectGet(handle);
236 
237                 // authValue is always available for a sequence object.
238                 if(ObjectIsSequence(object))
239                 {
240                     result =  TRUE;
241                     break;
242                 }
243                 // authValue is available for an object if it has its sensitive
244                 // portion loaded and
245                 //  1. userWithAuth bit is SET, or
246                 //  2. ADMIN role is required
247                 if(   object->attributes.publicOnly == CLEAR
248                    &&   (object->publicArea.objectAttributes.userWithAuth == SET
249                        || (CommandAuthRole(commandCode, sessionIndex) == AUTH_ADMIN
250                         &&  object->publicArea.objectAttributes.adminWithPolicy
251                            == CLEAR)))
252                     result = TRUE;
253             }
254             break;
255         case TPM_HT_NV_INDEX:
256             // NV Index.
257             {
258                 NV_INDEX        nvIndex;
259                 NvGetIndexInfo(handle, &nvIndex);
260                 if(IsWriteOperation(commandCode))
261                 {
262                     if (nvIndex.publicArea.attributes.TPMA_NV_AUTHWRITE == SET)
263                         result = TRUE;
264 
265                 }
266                 else
267                 {
268                     if (nvIndex.publicArea.attributes.TPMA_NV_AUTHREAD == SET)
269                         result = TRUE;
270                 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
271             }
272             break;
273         case TPM_HT_PCR:
274             // PCR handle.
275             // authValue is always allowed for PCR
276             result =  TRUE;
277             break;
278         default:
279             // Otherwise, authValue is not available
280             break;
281     }
282     return result;
283 }
```

7.4.3.6 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 6

Return Value	Meaning
TRUE	*authPolicy* is available
FALSE	*authPolicy* is not available

```
284 static BOOL
285 IsAuthPolicyAvailable(
286     TPM_HANDLE       handle,        // IN: handle of entity
287     TPM_CC           commandCode,   // IN: commandCode
288     UINT32           sessionIndex   // IN: session index
289     )
290 {
291     BOOL             result = FALSE;
292     switch(HandleGetType(handle))
293     {
294         case TPM_HT_PERMANENT:
295             switch(handle)
296             {
297                 // At this point hierarchy availability has already been checked.
298                 case TPM_RH_OWNER:
299                     if (gp.ownerPolicy.t.size != 0)
300                         result = TRUE;
301                     break;
302
303                 case TPM_RH_ENDORSEMENT:
304                     if (gp.endorsementPolicy.t.size != 0)
305                         result = TRUE;
306                     break;
307
308                 case TPM_RH_PLATFORM:
309                     if (gc.platformPolicy.t.size != 0)
310                         result = TRUE;
311                     break;
312                 default:
313                     break;
314             }
315             break;
316         case TPM_HT_TRANSIENT:
317             {
318                 // Object handle.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
319                 // An evict object would already have been loaded and given a
320                 // transient object handle by this point.
321                 OBJECT  *object = ObjectGet(handle);
322                 // Policy authorization is not available for an object with only
323                 // public portion loaded.
324                 if(object->attributes.publicOnly == CLEAR)
325                 {
326                     // Policy authorization is always available for an object but
327                     // is never available for a sequence.
328                     if(!ObjectIsSequence(object))
329                         result = TRUE;
330                 }
331                 break;
332             }
333         case TPM_HT_NV_INDEX:
334             // An NV Index.
335             {
336                 NV_INDEX         nvIndex;
337                 NvGetIndexInfo(handle, &nvIndex);
338                 // If the policy size is not zero, check if policy can be used.
339                 if(nvIndex.publicArea.authPolicy.t.size != 0)
340                 {
341                     // If policy session is required for this handle, always
342                     // uses policy regardless of the attributes bit setting
343                     if(IsPolicySessionRequired(commandCode, sessionIndex))
344                         result = TRUE;
345                     // Otherwise, the presence of the policy depends on the NV
346                     // attributes.
347                     else if(IsWriteOperation(commandCode))
348                     {
349                         if (   nvIndex.publicArea.attributes.TPMA_NV_POLICYWRITE
350                             == SET)
351                             result = TRUE;
352                     }
353                     else
354                     {
355                         if (    nvIndex.publicArea.attributes.TPMA_NV_POLICYREAD
356                             ==  SET)
357                             result = TRUE;
358                     }
359                 }
360             }
361             break;
362         case TPM_HT_PCR:
363             // PCR handle.
364             if(PCRPolicyIsAvailable(handle))
365                 result = TRUE;
366             break;
367         default:
368             break;
369     }
370     return result;
371 }
```

7.4.4 Session Parsing Functions

7.4.4.1 ComputeCpHash()

This function computes the *cpHash* as defined in ISO/IEC 11889-2 and specified in ISO/IEC 11889-1.

```
372 static void
373 ComputeCpHash(
374     TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
375         TPM_CC            commandCode,        // IN: command code
376         UINT32            handleNum,          // IN: number of handles
377         TPM_HANDLE        handles[],          // IN: array of handles
378         UINT32            parmBufferSize,     // IN: size of input parameter area
379         BYTE             *parmBuffer,         // IN: input parameter area
380         TPM2B_DIGEST     *cpHash,             // OUT: cpHash
381         TPM2B_DIGEST     *nameHash            // OUT: name hash of command
382         )
383     {
384         UINT32            i;
385         HASH_STATE        hashState;
386         TPM2B_NAME        name;
387
388         // cpHash = hash(commandCode [ || authName1
389         //                            [ || authName2
390         //                            [ || authName 3 ]]]
391         //                            [ || parameters])
392         // A cpHash can contain just a commandCode only if the lone session is
393         // an audit session.
394
395         // Start cpHash.
396         cpHash->t.size = CryptStartHash(hashAlg, &hashState);
397
398         //  Add commandCode.
399         CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
400
401         //  Add authNames for each of the handles.
402         for(i = 0; i < handleNum; i++)
403         {
404             name.t.size = EntityGetName(handles[i], &name.t.name);
405             CryptUpdateDigest2B(&hashState, &name.b);
406         }
407
408         //  Add the parameters.
409         CryptUpdateDigest(&hashState, parmBufferSize, parmBuffer);
410
411         //  Complete the hash.
412         CryptCompleteHash2B(&hashState, &cpHash->b);
413
414         // If the nameHash is needed, compute it here.
415         if(nameHash != NULL)
416         {
417             // Start name hash. hashState may be reused.
418             nameHash->t.size = CryptStartHash(hashAlg, &hashState);
419
420             //  Adding names.
421             for(i = 0; i < handleNum; i++)
422             {
423                 name.t.size = EntityGetName(handles[i], &name.t.name);
424                 CryptUpdateDigest2B(&hashState, &name.b);
425             }
426             //  Complete hash.
427             CryptCompleteHash2B(&hashState, &nameHash->b);
428         }
429         return;
430     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

7.4.4.2 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*[].

© ISO/IEC 2015 – All rights reserved

Table 7

Error Returns	Meaning
TPM_RC_AUTH_FAIL	auth fails and increments DA failure count
TPM_RC_BAD_AUTH	auth fails but DA does not apply

```
431  static TPM_RC
432  CheckPWAuthSession(
433      UINT32           sessionIndex   // IN: index of session to be processed
434      )
435  {
436      TPM2B_AUTH      authValue;
437      TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
438
439      // Strip trailing zeros from the password.
440      MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
441
442      // Get the auth value and size.
443      authValue.t.size = EntityGetAuthValue(associatedHandle, &authValue.t.buffer);
444
445      // Success if the digests are identical.
446      if(Memory2BEqual(&s_inputAuthValues[sessionIndex].b, &authValue.b))
447      {
448          return TPM_RC_SUCCESS;
449      }
450      else                    // if the digests are not identical
451      {
452          // Invoke DA protection if applicable.
453          return IncrementLockout(sessionIndex);
454      }
455  }
```

7.4.4.3 ComputeCommandHMAC()

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

```
456  static void
457  ComputeCommandHMAC(
458      UINT32           sessionIndex,  // IN: index of session to be processed
459      TPM2B_DIGEST    *cpHash,        // IN: cpHash
460      TPM2B_DIGEST    *hmac           // OUT: authorization HMAC
461      )
462  {
463      TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
464      TPM2B_KEY        key;
465      BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
466      BYTE            *buffer;
467      UINT32           marshalSize;
468      HMAC_STATE       hmacState;
469      TPM2B_NONCE     *nonceDecrypt;
470      TPM2B_NONCE     *nonceEncrypt;
471      SESSION         *session;
472      TPM_HT           sessionHandleType =
473                             HandleGetType(s_sessionHandles[sessionIndex]);
474
475      nonceDecrypt = NULL;
476      nonceEncrypt = NULL;
477
478      // Determine if extra nonceTPM values are going to be required.
479      // If this is the first session (sessionIndex = 0) and it is an authorization
480      // session that uses an HMAC, then check if additional session nonces are to be
481      // included.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
482         if(   sessionIndex == 0
483            && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
484         {
485             // If there is a decrypt session and if this is not the decrypt session,
486             // then an extra nonce may be needed.
487             if(   s_decryptSessionIndex != UNDEFINED_INDEX
488                && s_decryptSessionIndex != sessionIndex)
489             {
490                 // Will add the nonce for the decrypt session.
491                 SESSION *decryptSession
492                             = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
493                 nonceDecrypt = &decryptSession->nonceTPM;
494             }
495             // Now repeat for the encrypt session.
496             if(   s_encryptSessionIndex != UNDEFINED_INDEX
497                && s_encryptSessionIndex != sessionIndex
498                && s_encryptSessionIndex != s_decryptSessionIndex)
499             {
500                 // Have to have the nonce for the encrypt session.
501                 SESSION *encryptSession
502                             = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
503                 nonceEncrypt = &encryptSession->nonceTPM;
504             }
505         }
506
507         // Continue with the HMAC processing.
508         session = SessionGet(s_sessionHandles[sessionIndex]);
509
510         // Generate HMAC key.
511         MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
512
513         // Check if the session has an associated handle and if the associated entity
514         // is the one to which the session is bound. If not, add the authValue of
515         // this entity to the HMAC key.
516         // If the session is bound to the object or the session is a policy session
517         // with no authValue required, do not include the authValue in the HMAC key.
518         // Note: For a policy session, its isBound attribute is CLEARED.
519
520         // If the session isn't used for authorization, then there is no auth value
521         // to add
522         if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
523         {
524             // used for auth so see if this is a policy session with authValue needed
525             // or an hmac session that is not bound
526             if(         sessionHandleType == TPM_HT_POLICY_SESSION
527                     && session->attributes.isAuthValueNeeded == SET
528                 ||      sessionHandleType == TPM_HT_HMAC_SESSION
529                     && !IsSessionBindEntity(s_associatedHandles[sessionIndex], session)
530               )
531             {
532                 // add the authValue to the HMAC key
533                 pAssert((sizeof(AUTH_VALUE) + key.t.size) <= <K>sizeof(key.t.buffer));
534                 key.t.size =   key.t.size
535                              + EntityGetAuthValue(s_associatedHandles[sessionIndex],
536                                                   (AUTH_VALUE *)&(key.t.buffer[key.t.size]));
537             }
538         }
539
540          // if the HMAC key size is 0, a NULL string HMAC is allowed
541          if(    key.t.size == 0
542              && s_inputAuthValues[sessionIndex].t.size == 0)
543          {
544              hmac->t.size = 0;
545              return;
546          }
547
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
548         // Start HMAC
549         hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
550
551         //  Add cpHash
552         CryptUpdateDigest2B(&hmacState, &cpHash->b);
553
554         //  Add nonceCaller
555         CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
556
557         //  Add nonceTPM
558         CryptUpdateDigest2B(&hmacState, &session->nonceTPM.b);
559
560         //  If needed, add nonceTPM for decrypt session
561         if(nonceDecrypt != NULL)
562             CryptUpdateDigest2B(&hmacState, &nonceDecrypt->b);
563
564         //  If needed, add nonceTPM for encrypt session
565         if(nonceEncrypt != NULL)
566             CryptUpdateDigest2B(&hmacState, &nonceEncrypt->b);
567
568         //  Add sessionAttributes
569         buffer = marshalBuffer;
570         marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
571                                            &buffer, NULL);
572         CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
573
574         // Complete the HMAC computation
575         CryptCompleteHMAC2B(&hmacState, &hmac->b);
576
577         return;
578     }
```

7.4.4.4 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 8

Error Returns	Meaning
TPM_RC_AUTH_FAIL	auth failure caused *failureCount* increment
TPM_RC_BAD_AUTH	auth failure did not cause *failureCount* increment

```
579     static TPM_RC
580     CheckSessionHMAC(
581         UINT32           sessionIndex,  // IN: index of session to be processed
582         TPM2B_DIGEST    *cpHash         // IN: cpHash of the command
583         )
584     {
585         TPM2B_DIGEST        hmac;           // authHMAC for comparing
586
587         // Compute authHMAC
588         ComputeCommandHMAC(sessionIndex, cpHash, &hmac);
589
590         // Compare the input HMAC with the authHMAC computed above.
591         if(!Memory2BEqual(&s_inputAuthValues[sessionIndex].b,  &hmac.b))
592         {
593             // If an HMAC session has a failure, invoke the anti-hammering
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
594             // if it applies to the authorized entity or the session.
595             // Otherwise, just indicate that the authorization is bad.
596             return IncrementLockout(sessionIndex);
597         }
598         return TPM_RC_SUCCESS;
599     }
```

7.4.4.5 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.

Table 9

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

```
600     static TPM_RC
601     CheckPolicyAuthSession(
602         UINT32          sessionIndex,  // IN: index of session to be processed
603         TPM_CC          commandCode,   // IN: command code
604         TPM2B_DIGEST    *cpHash,       // IN: cpHash using the algorithm of this
605                                        //     session
606         TPM2B_DIGEST    *nameHash      // IN: nameHash using the session algorithm
607         )
608     {
609         TPM_RC           result = TPM_RC_SUCCESS;
610         SESSION         *session;
611         TPM2B_DIGEST     authPolicy;
612         TPMI_ALG_HASH    policyAlg;
613         UINT8            locality;
614
615         // Initialize pointer to the auth session.
616         session = SessionGet(s_sessionHandles[sessionIndex]);
617
618         // If the command is TPM_RC_PolicySecret(), make sure that
619         // either password or authValue is required
620         if(     commandCode == TPM_CC_PolicySecret
621             &&  session->attributes.isPasswordNeeded == CLEAR
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
622             &&  session->attributes.isAuthValueNeeded == CLEAR)
623             return TPM_RC_MODE;
624
625         // See if the PCR counter for the session is still valid.
626         if( !SessionPCRValueIsCurrent(s_sessionHandles[sessionIndex]) )
627             return TPM_RC_PCR_CHANGED;
628
629         // Get authPolicy.
630         policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
631                                         &authPolicy);
632         // Compare authPolicy.
633         if(!Memory2BEqual(&session->u2.policyDigest.b, &authPolicy.b))
634             return TPM_RC_POLICY_FAIL;
635
636         // Policy is OK so check if the other factors are correct
637
638         // Compare policy hash algorithm.
639         if(policyAlg != session->authHashAlg)
640             return TPM_RC_POLICY_FAIL;
641
642         // Compare timeout.
643         if(session->timeOut != 0)
644         {
645             // Cannot compare time if clock stop advancing.  An TPM_RC_NV_UNAVAILABLE
646             // or TPM_RC_NV_RATE error may be returned here.
647             result = NvIsAvailable();
648             if(result != TPM_RC_SUCCESS)
649                 return result;
650
651             if(session->timeOut < go.clock)
652                 return TPM_RC_EXPIRED;
653         }
654
655         // If command code is provided it must match
656         if(session->commandCode != 0)
657         {
658             if(session->commandCode != commandCode)
659                 return TPM_RC_POLICY_CC;
660         }
661         else
662         {
663             // If command requires a DUP or ADMIN authorization, the session must have
664             // command code set.
665             AUTH_ROLE   role = CommandAuthRole(commandCode, sessionIndex);
666             if(role == AUTH_ADMIN || role == AUTH_DUP)
667                 return TPM_RC_POLICY_FAIL;
668         }
669         // Check command locality.
670         {
671             BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
672             BYTE        *buffer = sessionLocality;
673
674             // Get existing locality setting in canonical form
675             TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
676
677             // See if the locality has been set
678             if(sessionLocality[0] != 0)
679             {
680                 // If so, get the current locality
681                 locality = _plat__LocalityGet();
682                 if (locality < 5)
683                 {
684                     if(    ((sessionLocality[0] & (1 << locality)) == 0)
685                         || sessionLocality[0] > 31)
686                         return TPM_RC_LOCALITY;
687                 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
688             else if (locality > 31)
689             {
690                 if(sessionLocality[0] != locality)
691                     return TPM_RC_LOCALITY;
692             }
693             else
694             {
695                 // Could throw an assert here but a locality error is just
696                 // as good. It just means that, whatever the locality is, it isn't
697                 // the locality requested so...
698                 return TPM_RC_LOCALITY;
699             }
700         }
701     } // end of locality check
702
703     // Check physical presence.
704     if(   session->attributes.isPPRequired == SET
705        && !_plat__PhysicalPresenceAsserted())
706         return TPM_RC_PP;
707
708     // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
709     // DUP role for this handle.
710     if(session->u1.cpHash.b.size != 0)
711     {
712         if(session->attributes.iscpHashDefined)
713         {
714             // Compare cpHash.
715             if(!Memory2BEqual(&session->u1.cpHash.b, &cpHash->b))
716                 return TPM_RC_POLICY_FAIL;
717         }
718         else
719         {
720             // Compare nameHash.
721             // When cpHash is not defined, nameHash is placed in its space.
722             if(!Memory2BEqual(&session->u1.cpHash.b, &nameHash->b))
723                 return TPM_RC_POLICY_FAIL;
724         }
725     }
726     if(session->attributes.checkNvWritten)
727     {
728         NV_INDEX        nvIndex;
729
730         // If this is not an NV index, the policy makes no sense so fail it.
731         if(HandleGetType(s_associatedHandles[sessionIndex])!= TPM_HT_NV_INDEX)
732             return TPM_RC_POLICY_FAIL;
733
734         // Get the index data
735         NvGetIndexInfo(s_associatedHandles[sessionIndex], &nvIndex);
736
737         // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
738         if(   (nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
739            != (session->attributes.nvWrittenState == SET))
740             return TPM_RC_POLICY_FAIL;
741     }
742
743     return TPM_RC_SUCCESS;
744 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

7.4.4.6 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.

© ISO/IEC 2015 – All rights reserved

Table 10

Error Returns	Meaning
TPM_RC_SUCCSS	unmarshaled without error
TPM_RC_SIZE	the number of bytes unmarshaled is not the same as the value for *authorizationSize* in the command

```
745  static TPM_RC
746  RetrieveSessionData (
747      TPM_CC           commandCode,   // IN: command code
748      UINT32          *sessionCount,  // OUT: number of sessions found
749      BYTE            *sessionBuffer, // IN: pointer to the session buffer
750      INT32            bufferSize     // IN: size of the session buffer
751      )
752  {
753      int          sessionIndex;
754      int          i;
755      TPM_RC       result;
756      SESSION     *session;
757      TPM_HT       sessionType;
758
759      s_decryptSessionIndex = UNDEFINED_INDEX;
760      s_encryptSessionIndex = UNDEFINED_INDEX;
761      s_auditSessionIndex = UNDEFINED_INDEX;
762
763      for(sessionIndex = 0; bufferSize > 0; sessionIndex++)
764      {
765          // If maximum allowed number of sessions has been parsed, return a size
766          // error with a session number that is larger than the number of allowed
767          // sessions
768          if(sessionIndex == MAX_SESSION_NUM)
769              return TPM_RC_SIZE + TPM_RC_S + g_rcIndex[sessionIndex+1];
770
771          // make sure that the associated handle for each session starts out
772          // unassigned
773          s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
774
775          // First parameter: Session handle.
776          result = TPMI_SH_AUTH_SESSION_Unmarshal(&s_sessionHandles[sessionIndex],
777                                                  &sessionBuffer, &bufferSize, TRUE);
778          if(result != TPM_RC_SUCCESS)
779              return result + TPM_RC_S + g_rcIndex[sessionIndex];
780
781          // Second parameter: Nonce.
782          result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
783                                         &sessionBuffer, &bufferSize);
784          if(result != TPM_RC_SUCCESS)
785              return result + TPM_RC_S + g_rcIndex[sessionIndex];
786
787          // Third parameter: sessionAttributes.
788          result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
789                                          &sessionBuffer, &bufferSize);
790          if(result != TPM_RC_SUCCESS)
791              return result + TPM_RC_S + g_rcIndex[sessionIndex];
792
793          // Fourth parameter: authValue (PW or HMAC).
794          result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
795                                        &sessionBuffer, &bufferSize);
796          if(result != TPM_RC_SUCCESS)
797              return result + TPM_RC_S + g_rcIndex[sessionIndex];
798
799          if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
800          {
801              // A PWAP session needs additional processing.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
802             //     Can't have any attributes set other than continueSession bit
803             if(   s_attributes[sessionIndex].encrypt
804                || s_attributes[sessionIndex].decrypt
805                || s_attributes[sessionIndex].audit
806                || s_attributes[sessionIndex].auditExclusive
807                || s_attributes[sessionIndex].auditReset
808               )
809                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
810
811             //     The nonce size must be zero.
812             if(s_nonceCaller[sessionIndex].t.size != 0)
813                 return TPM_RC_NONCE + TPM_RC_S + g_rcIndex[sessionIndex];
814
815             continue;
816         }
817         // For not password sessions...
818
819         // Find out if the session is loaded.
820         if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
821             return TPM_RC_REFERENCE_S0 + sessionIndex;
822
823         sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
824         session = SessionGet(s_sessionHandles[sessionIndex]);
825         // Check if the session is an HMAC/policy session.
826         if(   (   session->attributes.isPolicy == SET
827                && sessionType == TPM_HT_HMAC_SESSION
828               )
829            || (   session->attributes.isPolicy == CLEAR
830                &&  sessionType == TPM_HT_POLICY_SESSION
831               )
832           )
833             return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
834
835         // Check that this handle has not previously been used.
836         for(i = 0; i < sessionIndex; i++)
837         {
838             if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
839                 return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
840         }
841
842         // If the session is used for parameter encryption or audit as well, set
843         // the corresponding indices.
844
845         // First process decrypt.
846         if(s_attributes[sessionIndex].decrypt)
847         {
848             // Check if the commandCode allows command parameter encryption.
849             if(DecryptSize(commandCode) == 0)
850                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
851
852             // Encrypt attribute can only appear in one session
853             if(s_decryptSessionIndex != UNDEFINED_INDEX)
854                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
855
856             // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
857             if(session->symmetric.algorithm == TPM_ALG_NULL)
858                 return TPM_RC_SYMMETRIC + TPM_RC_S + g_rcIndex[sessionIndex];
859
860             // All checks passed, so set the index for the session used to decrypt
861             // a command parameter.
862             s_decryptSessionIndex = sessionIndex;
863         }
864
865         // Now process encrypt.
866         if(s_attributes[sessionIndex].encrypt)
867         {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
868             // Check if the commandCode allows response parameter encryption.
869             if(EncryptSize(commandCode) == 0)
870                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
871
872             // Encrypt attribute can only appear in one session.
873             if(s_encryptSessionIndex != UNDEFINED_INDEX)
874                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
875
876             // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
877             if(session->symmetric.algorithm == TPM_ALG_NULL)
878                 return TPM_RC_SYMMETRIC + TPM_RC_S + g_rcIndex[sessionIndex];
879
880             // All checks passed, so set the index for the session used to encrypt
881             // a response parameter.
882             s_encryptSessionIndex = sessionIndex;
883         }
884
885         // At last process audit.
886         if(s_attributes[sessionIndex].audit)
887         {
888             // Audit attribute can only appear in one session.
889             if(s_auditSessionIndex != UNDEFINED_INDEX)
890                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
891
892             // An audit session can not be policy session.
893             if(   HandleGetType(s_sessionHandles[sessionIndex])
894                == TPM_HT_POLICY_SESSION)
895                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
896
897             // If this is a reset of the audit session, or the first use
898             // of the session as an audit session, it doesn't matter what
899             // the exclusive state is. The session will become exclusive.
900             if(   s_attributes[sessionIndex].auditReset == CLEAR
901                && session->attributes.isAudit == SET)
902             {
903                 // Not first use or reset. If auditExlusive is SET, then this
904                 // session must be the current exclusive session.
905                 if(   s_attributes[sessionIndex].auditExclusive == SET
906                    && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
907                    return TPM_RC_EXCLUSIVE;
908             }
909
910             s_auditSessionIndex = sessionIndex;
911         }
912
913         // Initialize associated handle as undefined. This will be changed when
914         // the handles are processed.
915         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
916
917     }
918
919
920     // Set the number of sessions found.
921     *sessionCount = sessionIndex;
922     return TPM_RC_SUCCESS;
923 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

7.4.4.7 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.*

© ISO/IEC 2015 – All rights reserved

Table 11

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

```
924  static TPM_RC
925  CheckLockedOut(
926      BOOL             lockoutAuthCheck   // IN: TRUE if checking is for lockoutAuth
927      )
928  {
929      TPM_RC      result;
930
931      // If NV is unavailable, and current cycle state recorded in NV is not
932      // SHUTDOWN_NONE, refuse to check any authorization because we would
933      // not be able to handle a DA failure.
934      result = NvIsAvailable();
935      if(result != TPM_RC_SUCCESS && gp.orderlyState != SHUTDOWN_NONE)
936          return result;
937
938      // Check if DA info needs to be updated in NV.
939      if(s_DAPendingOnNV)
940      {
941          // If NV is accessible, ...
942          if(result == TPM_RC_SUCCESS)
943          {
944              // ... write the pending DA data and proceed.
945              NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED,
946                              &gp.lockOutAuthEnabled);
947              NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
948              g_updateNV = TRUE;
949              s_DAPendingOnNV = FALSE;
950          }
951          else
952          {
953              // Otherwise no authorization can be checked.
954              return result;
955          }
956      }
957
958      // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
959      // is disabled...
960      if(lockoutAuthCheck)
961      {
962          if(gp.lockOutAuthEnabled == FALSE)
963              return TPM_RC_LOCKOUT;
964      }
965      else
966      {
967          // ... or if the number of failed tries has been maxed out.
968          if(gp.failedTries >= gp.maxTries)
969              return TPM_RC_LOCKOUT;
970      }
971      return TPM_RC_SUCCESS;
972  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

7.4.4.8 CheckAuthSession()

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

© ISO/IEC 2015 – All rights reserved

Table 12

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

```
973  static TPM_RC
974  CheckAuthSession(
975      TPM_CC           commandCode,   // IN: commandCode
976      UINT32           sessionIndex,  // IN: index of session to be processed
977      TPM2B_DIGEST    *cpHash,        // IN: cpHash
978      TPM2B_DIGEST    *nameHash       // IN: nameHash
979      )
980  {
981      TPM_RC           result;
982      SESSION         *session = NULL;
983      TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
984      TPM_HANDLE       associatedHandle = s_associatedHandles[sessionIndex];
985      TPM_HT           sessionHandleType = HandleGetType(sessionHandle);
986
987      pAssert(sessionHandle != TPM_RH_UNASSIGNED);
988
989      if(sessionHandle != TPM_RS_PW)
990          session = SessionGet(sessionHandle);
991
992      pAssert(sessionHandleType != TPM_HT_POLICY_SESSION || session != NULL);
993
994      // If the authorization session is not a policy session, or if the policy
995      // session requires authorization, then check lockout.
996      if(    sessionHandleType != TPM_HT_POLICY_SESSION
997         || session->attributes.isAuthValueNeeded
998         || session->attributes.isPasswordNeeded)
999      {
1000         // See if entity is subject to lockout.
1001         if(!IsDAExempted(associatedHandle))
1002         {
1003             // If NV is unavailable, and current cycle state recorded in NV is not
1004             // SHUTDOWN_NONE, refuse to check any authorization because we would
1005             // not be able to handle a DA failure.
1006             result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1007             if(result != TPM_RC_SUCCESS)
1008                 return result;
1009         }
1010     }
1011
1012     if(associatedHandle == TPM_RH_PLATFORM)
1013     {
1014         // If the physical presence is required for this command, check for PP
1015         // assertion. If it isn't asserted, no point going any further.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1016            if(   PhysicalPresenceIsRequired(commandCode)
1017               && !_plat__PhysicalPresenceAsserted()
1018              )
1019                return TPM_RC_PP;
1020        }
1021        // If a policy session is required, make sure that it is being used.
1022        if(   IsPolicySessionRequired(commandCode, sessionIndex)
1023           && sessionHandleType != TPM_HT_POLICY_SESSION)
1024            return TPM_RC_AUTH_TYPE;
1025
1026        // If this is a PW authorization, check it and return.
1027        if(sessionHandle == TPM_RS_PW)
1028        {
1029            if(IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
1030                return CheckPWAuthSession(sessionIndex);
1031            else
1032                return TPM_RC_AUTH_UNAVAILABLE;
1033        }
1034        // If this is a policy session, ...
1035        if(sessionHandleType == TPM_HT_POLICY_SESSION)
1036        {
1037            // ... see if the entity has a policy, ...
1038            if( !IsAuthPolicyAvailable(associatedHandle, commandCode, sessionIndex))
1039                return TPM_RC_AUTH_UNAVAILABLE;
1040            // ... and check the policy session.
1041            result = CheckPolicyAuthSession(sessionIndex, commandCode,
1042                                            cpHash, nameHash);
1043            if (result != TPM_RC_SUCCESS)
1044                return result;
1045        }
1046        else
1047        {
1048            // For non policy, the entity being accessed must allow authorization
1049            // with an auth value. This is required even if the auth value is not
1050            // going to be used in an HMAC because it is bound.
1051            if(!IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
1052                return TPM_RC_AUTH_UNAVAILABLE;
1053        }
1054        // At this point, the session must be either a policy or an HMAC session.
1055        session = SessionGet(s_sessionHandles[sessionIndex]);
1056
1057        if(     sessionHandleType == TPM_HT_POLICY_SESSION
1058            &&  session->attributes.isPasswordNeeded == SET)
1059        {
1060            // For policy session that requires a password, check it as PWAP session.
1061            return CheckPWAuthSession(sessionIndex);
1062        }
1063        else
1064        {
1065            // For other policy or HMAC sessions, have its HMAC checked.
1066            return CheckSessionHMAC(sessionIndex, cpHash);
1067        }
1068    }
1069    #ifdef  TPM_CC_GetCommandAuditDigest
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

7.4.4.9 CheckCommandAudit()

This function checks if the current command may trigger command audit, and if it is safe to perform the action.

© ISO/IEC 2015 – All rights reserved

Table 13

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

```
1070    static TPM_RC
1071    CheckCommandAudit(
1072        TPM_CC           commandCode,       // IN: Command code
1073        UINT32           handleNum,         // IN: number of element in handle array
1074        TPM_HANDLE       handles[],         // IN: array of handles
1075        BYTE            *parmBufferStart,   // IN: start of parameter buffer
1076        UINT32           parmBufferSize     // IN: size of parameter buffer
1077        )
1078    {
1079        TPM_RC      result = TPM_RC_SUCCESS;
1080
1081        // If audit is implemented, need to check to see if auditing is being done
1082        // for this command.
1083        if(CommandAuditIsRequired(commandCode))
1084        {
1085            // If the audit digest is clear and command audit is required, NV must be
1086            // available so that TPM2_GetCommandAuditDigest() is able to increment
1087            // audit counter. If NV is not available, the function bails out to prevent
1088            // the TPM from attempting an operation that would fail anyway.
1089            if(    gr.commandAuditDigest.t.size == 0
1090               || commandCode == TPM_CC_GetCommandAuditDigest)
1091            {
1092                result = NvIsAvailable();
1093                if(result != TPM_RC_SUCCESS)
1094                    return result;
1095            }
1096            ComputeCpHash(gp.auditHashAlg, commandCode, handleNum,
1097                          handles, parmBufferSize, parmBufferStart,
1098                          &s_cpHashForCommandAudit, NULL);
1099        }
1100
1101        return TPM_RC_SUCCESS;
1102    }
1103    #endif
```

7.4.4.10 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.

Table 14

Error Returns	Meaning
various	parsing failure or authorization failure

```
1104    TPM_RC
1105    ParseSessionBuffer(
1106        TPM_CC           commandCode,           // IN: Command code
1107        UINT32           handleNum,             // IN: number of element in handle array
1108        TPM_HANDLE       handles[],             // IN: array of handles
1109        BYTE            *sessionBufferStart,    // IN: start of session buffer
1110        UINT32           sessionBufferSize,     // IN: size of session buffer
1111        BYTE            *parmBufferStart,       // IN: start of parameter buffer
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
1112        UINT32          parmBufferSize        // IN: size of parameter buffer
1113        )
1114    {
1115        TPM_RC          result;
1116        UINT32          i;
1117        INT32           size = 0;
1118        TPM2B_AUTH      extraKey;
1119        UINT32          sessionIndex;
1120        SESSION        *session;
1121        TPM2B_DIGEST    cpHash;
1122        TPM2B_DIGEST    nameHash;
1123        TPM_ALG_ID      cpHashAlg = TPM_ALG_NULL;  // algID for the last computed
1124                                                   // cpHash
1125
1126        // Check if a command allows any session in its session area.
1127        if(!IsSessionAllowed(commandCode))
1128            return TPM_RC_AUTH_CONTEXT;
1129
1130        // Default-initialization.
1131        s_sessionNum = 0;
1132        cpHash.t.size = 0;
1133
1134        result = RetrieveSessionData(commandCode, &s_sessionNum,
1135                                     sessionBufferStart, sessionBufferSize);
1136        if(result != TPM_RC_SUCCESS)
1137            return result;
1138
1139        // There is no command in the TPM spec that has more handles than
1140        // MAX_SESSION_NUM.
1141        pAssert(handleNum <= MAX_SESSION_NUM);
1142
1143        // Associate the session with an authorization handle.
1144        for(i = 0; i < handleNum; i++)
1145        {
1146            if(CommandAuthRole(commandCode, i) != AUTH_NONE)
1147            {
1148                // If the received session number is less than the number of handle
1149                // that requires authorization, an error should be returned.
1150                // Note: for all the ISO/IEC 11889 commands, handles requiring
1151                // authorization come first in a command input.
1152                if(i > (s_sessionNum - 1))
1153                    return TPM_RC_AUTH_MISSING;
1154
1155                // Record the handle associated with the authorization session
1156                s_associatedHandles[i] = handles[i];
1157            }
1158        }
1159
1160        // Consistency checks are done first to avoid auth failure when the command
1161        // will not be executed anyway.
1162        for(sessionIndex = 0; sessionIndex < s_sessionNum; sessionIndex++)
1163        {
1164            // PW session must be an authorization session
1165            if(s_sessionHandles[sessionIndex] == TPM_RS_PW )
1166            {
1167                if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1168                    return TPM_RC_HANDLE + g_rcIndex[sessionIndex];
1169            }
1170            else
1171            {
1172                session = SessionGet(s_sessionHandles[sessionIndex]);
1173
1174                // A trial session can not appear in session area, because it cannot
1175                // be used for authorization, audit or encrypt/decrypt.
1176                if(session->attributes.isTrialPolicy == SET)
1177                    return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1178
1179            // See if the session is bound to a DA protected entity
1180            // NOTE: Since a policy session is never bound, a policy is still
1181            // usable even if the object is DA protected and the TPM is in
1182            // lockout.
1183            if(session->attributes.isDaBound == SET)
1184            {
1185                result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1186                if(result != TPM_RC_SUCCESS)
1187                    return result;
1188            }
1189            // If the current cpHash is the right one, don't re-compute.
1190            if(cpHashAlg != session->authHashAlg)   // different so compute
1191            {
1192                cpHashAlg = session->authHashAlg;   // save this new algID
1193                ComputeCpHash(session->authHashAlg, commandCode, handleNum,
1194                              handles, parmBufferSize, parmBufferStart,
1195                              &cpHash, &nameHash);
1196            }
1197            // If this session is for auditing, save the cpHash.
1198            if(s_attributes[sessionIndex].audit)
1199                s_cpHashForAudit = cpHash;
1200        }
1201
1202        // if the session has an associated handle, check the auth
1203        if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1204        {
1205            result = CheckAuthSession(commandCode, sessionIndex,
1206                                      &cpHash, &nameHash);
1207            if(result != TPM_RC_SUCCESS)
1208                return RcSafeAddToResult(result,
1209                                         TPM_RC_S + g_rcIndex[sessionIndex]);
1210        }
1211        else
1212        {
1213            // a session that is not for authorization must either be encrypt,
1214            // decrypt, or audit
1215            if(     s_attributes[sessionIndex].audit == CLEAR
1216                &&  s_attributes[sessionIndex].encrypt == CLEAR
1217                &&  s_attributes[sessionIndex].decrypt == CLEAR)
1218                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
1219
1220            // check HMAC for encrypt/decrypt/audit only sessions
1221            result =  CheckSessionHMAC(sessionIndex, &cpHash);
1222            if(result != TPM_RC_SUCCESS)
1223                return RcSafeAddToResult(result,
1224                                         TPM_RC_S + g_rcIndex[sessionIndex]);
1225        }
1226    }
1227
1228  #ifdef  TPM_CC_GetCommandAuditDigest
1229    // Check if the command should be audited.
1230    result = CheckCommandAudit(commandCode, handleNum, handles,
1231                               parmBufferStart, parmBufferSize);
1232    if(result != TPM_RC_SUCCESS)
1233        return result;              // No session number to reference
1234  #endif
1235
1236    // Decrypt the first parameter if applicable. This should be the last operation
1237    // in session processing.
1238    // If the encrypt session is associated with a handle and the handle's
1239    // authValue is available, then authValue is concatenated with sessionAuth to
1240    // generate encryption key, no matter if the handle is the session bound entity
1241    // or not.
1242    if(s_decryptSessionIndex != UNDEFINED_INDEX)
1243    {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1244            // Get size of the leading size field in decrypt parameter
1245            if(   s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED
1246               && IsAuthValueAvailable(s_associatedHandles[s_decryptSessionIndex],
1247                                       commandCode,
1248                                       s_decryptSessionIndex)
1249              )
1250            {
1251                extraKey.b.size=
1252                    EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1253                                       &extraKey.t.buffer);
1254            }
1255            else
1256            {
1257                extraKey.b.size = 0;
1258            }
1259            size = DecryptSize(commandCode);
1260            result = CryptParameterDecryption(
1261                        s_sessionHandles[s_decryptSessionIndex],
1262                        &s_nonceCaller[s_decryptSessionIndex].b,
1263                        parmBufferSize, (UINT16)size,
1264                        &extraKey,
1265                        parmBufferStart);
1266            if(result != TPM_RC_SUCCESS)
1267                return RcSafeAddToResult(result,
1268                                         TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1269        }
1270
1271        return TPM_RC_SUCCESS;
1272    }
```

7.4.4.11 CheckAuthNoSession()

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

Table 15

Error Returns	Meaning
TPM_RC_AUTH_MISSING	failure - one or more handles require auth

```
1273    TPM_RC
1274    CheckAuthNoSession(
1275        TPM_CC            commandCode,        // IN: Command Code
1276        UINT32            handleNum,          // IN: number of handles in command
1277        TPM_HANDLE        handles[],          // IN: array of handles
1278        BYTE             *parmBufferStart,    // IN: start of parameter buffer
1279        UINT32            parmBufferSize      // IN: size of parameter buffer
1280        )
1281    {
1282        UINT32 i;
1283        TPM_RC            result = TPM_RC_SUCCESS;
1284
1285        // Check if the commandCode requires authorization
1286        for(i = 0; i < handleNum; i++)
1287        {
1288            if(CommandAuthRole(commandCode, i) != AUTH_NONE)
1289                return TPM_RC_AUTH_MISSING;
1290        }
1291
1292    #ifdef  TPM_CC_GetCommandAuditDigest
1293        // Check if the command should be audited.
1294        result = CheckCommandAudit(commandCode, handleNum, handles,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1295                                parmBufferStart, parmBufferSize);
1296        if(result != TPM_RC_SUCCESS) return result;
1297    #endif
1298
1299        // Initialize number of sessions to be 0
1300        s_sessionNum = 0;
1301
1302        return TPM_RC_SUCCESS;
1303    }
```

7.4.5 Response Session Processing

7.4.5.1 Introduction

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

7.4.5.2 ComputeRpHash()

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

```
1304    static void
1305    ComputeRpHash(
1306        TPM_ALG_ID       hashAlg,           // IN: hash algorithm to compute rpHash
1307        TPM_CC           commandCode,       // IN: commandCode
1308        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1309        BYTE            *resParmBuffer,     // IN: response parameter buffer
1310        TPM2B_DIGEST    *rpHash             // OUT: rpHash
1311        )
1312    {
1313        // The command result in rpHash is always TPM_RC_SUCCESS.
1314        TPM_RC      responseCode = TPM_RC_SUCCESS;
1315        HASH_STATE  hashState;
1316
1317        //   rpHash := hash(responseCode || commandCode || parameters)
1318
1319        // Initiate hash creation.
1320        rpHash->t.size = CryptStartHash(hashAlg, &hashState);
1321
1322        // Add hash constituents.
1323        CryptUpdateDigestInt(&hashState, sizeof(TPM_RC), &responseCode);
1324        CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
1325        CryptUpdateDigest(&hashState, resParmBufferSize, resParmBuffer);
1326
1327        // Complete hash computation.
1328        CryptCompleteHash2B(&hashState, &rpHash->b);
1329
1330        return;
1331    }
```

7.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```
1332    static void
1333    InitAuditSession(
1334        SESSION         *session        // session to be initialized
1335        )
1336    {
1337        // Mark session as an audit session.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
1338        session->attributes.isAudit = SET;
1339
1340        // Audit session can not be bound.
1341        session->attributes.isBound = CLEAR;
1342
1343        // Size of the audit log is the size of session hash algorithm digest.
1344        session->u2.auditDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
1345
1346        // Set the original digest value to be 0.
1347        MemorySet(&session->u2.auditDigest.t.buffer,
1348                  0,
1349                  session->u2.auditDigest.t.size);
1350
1351        return;
1352    }
```

7.4.5.4 Audit()

This function updates the audit digest in an audit session.

```
1353    static void
1354    Audit(
1355        SESSION         *auditSession,      // IN: loaded audit session
1356        TPM_CC           commandCode,       // IN: commandCode
1357        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1358        BYTE            *resParmBuffer      // IN: response parameter buffer
1359        )
1360    {
1361        TPM2B_DIGEST     rpHash;            // rpHash for response
1362        HASH_STATE       hashState;
1363
1364        // Compute rpHash
1365        ComputeRpHash(auditSession->authHashAlg,
1366                      commandCode,
1367                      resParmBufferSize,
1368                      resParmBuffer,
1369                      &rpHash);
1370
1371        // auditDigestnew :=  hash (auditDigestold || cpHash || rpHash)
1372
1373        // Start hash computation.
1374        CryptStartHash(auditSession->authHashAlg, &hashState);
1375
1376        // Add old digest.
1377        CryptUpdateDigest2B(&hashState, &auditSession->u2.auditDigest.b);
1378
1379        // Add cpHash and rpHash.
1380        CryptUpdateDigest2B(&hashState, &s_cpHashForAudit.b);
1381        CryptUpdateDigest2B(&hashState, &rpHash.b);
1382
1383        // Finalize the hash.
1384        CryptCompleteHash2B(&hashState, &auditSession->u2.auditDigest.b);
1385
1386        return;
1387    }
1388    #ifdef  TPM_CC_GetCommandAuditDigest
```

7.4.5.5 CommandAudit()

This function updates the command audit digest.

```
1389    static void
1390    CommandAudit(
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1391        TPM_CC           commandCode,       // IN: commandCode
1392        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1393        BYTE            *resParmBuffer      // IN: response parameter buffer
1394        )
1395    {
1396        if(CommandAuditIsRequired(commandCode))
1397        {
1398            TPM2B_DIGEST    rpHash;        // rpHash for response
1399            HASH_STATE      hashState;
1400
1401            // Compute rpHash.
1402            ComputeRpHash(gp.auditHashAlg, commandCode, resParmBufferSize,
1403                          resParmBuffer, &rpHash);
1404
1405            // If the digest.size is one, it indicates the special case of changing
1406            // the audit hash algorithm. For this case, no audit is done on exit.
1407            // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1408            // force an update to the NV on exit so that the change in digest will
1409            // be recorded. So, it is safe to exit here without setting any flags
1410            // because the digest change will be written to NV when this code exits.
1411            if(gr.commandAuditDigest.t.size == 1)
1412            {
1413                gr.commandAuditDigest.t.size = 0;
1414                return;
1415            }
1416
1417            // If the digest size is zero, need to start a new digest and increment
1418            // the audit counter.
1419            if(gr.commandAuditDigest.t.size == 0)
1420            {
1421                gr.commandAuditDigest.t.size = CryptGetHashDigestSize(gp.auditHashAlg);
1422                MemorySet(gr.commandAuditDigest.t.buffer,
1423                          0,
1424                          gr.commandAuditDigest.t.size);
1425
1426                // Bump the counter and save its value to NV.
1427                gp.auditCounter++;
1428                NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
1429                g_updateNV = TRUE;
1430            }
1431
1432            // auditDigestnew := hash (auditDigestold || cpHash || rpHash)
1433
1434            //  Start hash computation.
1435            CryptStartHash(gp.auditHashAlg, &hashState);
1436
1437            //  Add old digest.
1438            CryptUpdateDigest2B(&hashState, &gr.commandAuditDigest.b);
1439
1440            //  Add cpHash
1441            CryptUpdateDigest2B(&hashState, &s_cpHashForCommandAudit.b);
1442
1443            //  Add rpHash
1444            CryptUpdateDigest2B(&hashState, &rpHash.b);
1445
1446            //  Finalize the hash.
1447            CryptCompleteHash2B(&hashState, &gr.commandAuditDigest.b);
1448        }
1449        return;
1450    }
1451    #endif
```

7.4.5.6 UpdateAuditSessionStatus()

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

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.

```
1452  static void
1453  UpdateAuditSessionStatus(
1454      TPM_CC           commandCode,       // IN: commandCode
1455      UINT32           resParmBufferSize, // IN: size of response parameter buffer
1456      BYTE            *resParmBuffer      // IN: response parameter buffer
1457      )
1458  {
1459      UINT32           i;
1460      TPM_HANDLE       auditSession = TPM_RH_UNASSIGNED;
1461
1462      // Iterate through sessions
1463      for (i = 0; i < s_sessionNum; i++)
1464      {
1465          SESSION     *session;
1466
1467          // PW session do not have a loaded session and can not be an audit
1468          // session either.  Skip it.
1469          if(s_sessionHandles[i] == TPM_RS_PW) continue;
1470
1471          session = SessionGet(s_sessionHandles[i]);
1472
1473          // If a session is used for audit
1474          if(s_attributes[i].audit == SET)
1475          {
1476              // An audit session has been found
1477              auditSession = s_sessionHandles[i];
1478
1479              // If the session has not been an audit session yet, or
1480              // the auditSetting bits indicate a reset, initialize it and set
1481              // it to be the exclusive session
1482              if(   session->attributes.isAudit == CLEAR
1483                 || s_attributes[i].auditReset == SET
1484                )
1485              {
1486                  InitAuditSession(session);
1487                  g_exclusiveAuditSession = auditSession;
1488              }
1489              else
1490              {
1491                  // Check if the audit session is the current exclusive audit
1492                  // session and, if not, clear previous exclusive audit session.
1493                  if(g_exclusiveAuditSession != auditSession)
1494                      g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1495              }
1496
1497              // Report audit session exclusivity.
1498              if(g_exclusiveAuditSession == auditSession)
1499              {
1500                  s_attributes[i].auditExclusive = SET;
1501              }
1502              else
1503              {
1504                  s_attributes[i].auditExclusive = CLEAR;
1505              }
1506
1507              // Extend audit log.
1508              Audit(session, commandCode, resParmBufferSize, resParmBuffer);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1509            }
1510        }
1511
1512        // If no audit session is found in the command, and the command allows
1513        // a session then, clear the current exclusive
1514        // audit session.
1515        if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(commandCode))
1516        {
1517            g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1518        }
1519
1520        return;
1521    }
```

7.4.5.7 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```
1522    static void
1523    ComputeResponseHMAC(
1524        UINT32           sessionIndex,      // IN: session index to be processed
1525        SESSION         *session,           // IN: loaded session
1526        TPM_CC           commandCode,       // IN: commandCode
1527        TPM2B_NONCE     *nonceTPM,          // IN: nonceTPM
1528        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1529        BYTE            *resParmBuffer,     // IN: response parameter buffer
1530        TPM2B_DIGEST    *hmac               // OUT: authHMAC
1531        )
1532    {
1533        TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1534        TPM2B_KEY        key;       // HMAC key
1535        BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
1536        BYTE            *buffer;
1537        UINT32           marshalSize;
1538        HMAC_STATE       hmacState;
1539        TPM2B_DIGEST     rp_hash;
1540
1541        // Compute rpHash.
1542        ComputeRpHash(session->authHashAlg, commandCode, resParmBufferSize,
1543                      resParmBuffer, &rp_hash);
1544
1545        // Generate HMAC key
1546        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1547
1548        // Check if the session has an associated handle and the associated entity is
1549        // the one that the session is bound to.
1550        // If not bound, add the authValue of this entity to the HMAC key.
1551        if(   s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED
1552           &&  !( HandleGetType(s_sessionHandles[sessionIndex])
1553                  == TPM_HT_POLICY_SESSION
1554              &&   session->attributes.isAuthValueNeeded == CLEAR)
1555           && !session->attributes.requestWasBound)
1556        {
1557            pAssert((sizeof(AUTH_VALUE) + key.t.size) <= <K>sizeof(key.t.buffer));
1558            key.t.size = key.t.size +
1559                         EntityGetAuthValue(s_associatedHandles[sessionIndex],
1560                                            (AUTH_VALUE *)&key.t.buffer[key.t.size]);
1561        }
1562
1563        // if the HMAC key size for a policy session is 0, the response HMAC is
1564        // computed according to the input HMAC
1565        if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1566            && key.t.size == 0
1567            && s_inputAuthValues[sessionIndex].t.size == 0)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1568        {
1569            hmac->t.size = 0;
1570            return;
1571        }
1572
1573        // Start HMAC computation.
1574        hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
1575
1576        // Add hash components.
1577        CryptUpdateDigest2B(&hmacState, &rp_hash.b);
1578        CryptUpdateDigest2B(&hmacState, &nonceTPM->b);
1579        CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
1580
1581        // Add session attributes.
1582        buffer = marshalBuffer;
1583        marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1584        CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
1585
1586        // Finalize HMAC.
1587        CryptCompleteHMAC2B(&hmacState, &hmac->b);
1588
1589        return;
1590    }
```

7.4.5.8 BuildSingleResponseAuth()

Function to compute response for an authorization session.

```
1591    static void
1592    BuildSingleResponseAuth(
1593        UINT32           sessionIndex,      // IN: session index to be processed
1594        TPM_CC           commandCode,       // IN: commandCode
1595        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1596        BYTE            *resParmBuffer,     // IN: response parameter buffer
1597        TPM2B_AUTH      *auth               // OUT: authHMAC
1598        )
1599    {
1600        // For password authorization, field is empty.
1601        if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1602        {
1603            auth->t.size = 0;
1604        }
1605        else
1606        {
1607            // Fill in policy/HMAC based session response.
1608            SESSION     *session = SessionGet(s_sessionHandles[sessionIndex]);
1609
1610            // If the session is a policy session with isPasswordNeeded SET, the auth
1611            // field is empty.
1612            if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1613                && session->attributes.isPasswordNeeded == SET)
1614                auth->t.size = 0;
1615            else
1616                // Compute response HMAC.
1617                ComputeResponseHMAC(sessionIndex,
1618                                    session,
1619                                    commandCode,
1620                                    &session->nonceTPM,
1621                                    resParmBufferSize,
1622                                    resParmBuffer,
1623                                    auth);
1624        }
1625
1626        return;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
}
```

7.4.5.9 UpdateTPMNonce()

Updates TPM nonce in both internal session or response if applicable.

```
static void
UpdateTPMNonce(
    UINT16          noncesSize,     // IN: number of elements in 'nonces' array
    TPM2B_NONCE     nonces[]        // OUT: nonceTPM
    )
{
    UINT32      i;
    pAssert(noncesSize >= s_sessionNum);
    for(i = 0; i < s_sessionNum; i++)
    {
        SESSION     *session;
        // For PW session, nonce is 0.
        if(s_sessionHandles[i] == TPM_RS_PW)
        {
            nonces[i].t.size = 0;
            continue;
        }
        session = SessionGet(s_sessionHandles[i]);
        // Update nonceTPM in both internal session and response.
        CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
        nonces[i] = session->nonceTPM;
    }
    return;
}
```

7.4.5.10 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time.

b) Clears a policy session since nonce is rolling.

```
static void
UpdateInternalSession(
    void
    )
{
    UINT32      i;
    for(i = 0; i < s_sessionNum; i++)
    {
        // For PW session, no update.
        if(s_sessionHandles[i] == TPM_RS_PW) continue;

        if(s_attributes[i].continueSession == CLEAR)
        {
            // Close internal session.
            SessionFlush(s_sessionHandles[i]);
        }
        else
        {
            // 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)
            {
                SESSION     *session = SessionGet(s_sessionHandles[i]);

```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
1676                // When the nonce rolls it starts a new timing interval for the
1677                // policy session.
1678                SessionResetPolicyData(session);
1679                session->startTime = go.clock;
1680            }
1681        }
1682    }
1683    return;
1684  }
```

7.4.5.11 BuildResponseSession()

Function to build Session buffer in a response.

```
1685  void
1686  BuildResponseSession(
1687      TPM_ST           tag,           // IN: tag
1688      TPM_CC           commandCode,   // IN: commandCode
1689      UINT32           resHandleSize, // IN: size of response handle buffer
1690      UINT32           resParmSize,   // IN: size of response parameter buffer
1691      UINT32          *resSessionSize // OUT: response session area
1692      )
1693  {
1694      BYTE            *resParmBuffer;
1695      TPM2B_NONCE  responseNonces[MAX_SESSION_NUM];
1696
1697      // Compute response parameter buffer start.
1698      resParmBuffer = MemoryGetResponseBuffer(commandCode) + sizeof(TPM_ST) +
1699                      sizeof(UINT32) + sizeof(TPM_RC) + resHandleSize;
1700
1701      // For TPM_ST_SESSIONS, there is parameterSize field.
1702      if(tag == TPM_ST_SESSIONS)
1703          resParmBuffer += sizeof(UINT32);
1704
1705      // Session nonce should be updated before parameter encryption
1706      if(tag == TPM_ST_SESSIONS)
1707      {
1708          UpdateTPMNonce(MAX_SESSION_NUM, responseNonces);
1709
1710          // Encrypt first parameter if applicable. Parameter encryption should
1711          // happen after nonce update and before any rpHash is computed.
1712          // If the encrypt session is associated with a handle, the authValue of
1713          // this handle will be concatenated with sessionAuth to generate
1714          // encryption key, no matter if the handle is the session bound entity
1715          // or not. The authValue is added to sessionAuth only when the authValue
1716          // is available.
1717          if(s_encryptSessionIndex != UNDEFINED_INDEX)
1718          {
1719              UINT32          size;
1720              TPM2B_AUTH      extraKey;
1721
1722              // Get size of the leading size field
1723              if(   s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED
1724                 && IsAuthValueAvailable(s_associatedHandles[s_encryptSessionIndex],
1725                                         commandCode, s_encryptSessionIndex)
1726                )
1727              {
1728                  extraKey.b.size =
1729                      EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1730                                         &extraKey.t.buffer);
1731              }
1732              else
1733              {
1734                  extraKey.b.size = 0;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1735                }
1736                size = EncryptSize(commandCode);
1737                CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1738                                         &s_nonceCaller[s_encryptSessionIndex].b,
1739                                         (UINT16)size,
1740                                         &extraKey,
1741                                         resParmBuffer);
1742
1743            }
1744
1745        }
1746        // Audit session should be updated first regardless of the tag.
1747        // A command with no session may trigger a change of the exclusivity state.
1748        UpdateAuditSessionStatus(commandCode, resParmSize, resParmBuffer);
1749
1750        // Audit command.
1751        CommandAudit(commandCode, resParmSize, resParmBuffer);
1752
1753        // Process command with sessions.
1754        if(tag == TPM_ST_SESSIONS)
1755        {
1756            UINT32          i;
1757            BYTE           *buffer;
1758            TPM2B_DIGEST    responseAuths[MAX_SESSION_NUM];
1759
1760            pAssert(s_sessionNum > 0);
1761
1762            // Iterate over each session in the command session area, and create
1763            // corresponding sessions for response.
1764            for(i = 0; i < s_sessionNum; i++)
1765            {
1766                BuildSingleResponseAuth(
1767                                         i,
1768                                         commandCode,
1769                                         resParmSize,
1770                                         resParmBuffer,
1771                                         &responseAuths[i]);
1772                // Make sure that continueSession is SET on any Password session.
1773                // This makes it marginally easier for the management software
1774                // to keep track of the closed sessions.
1775                if(   s_attributes[i].continueSession == CLEAR
1776                   && s_sessionHandles[i] == TPM_RS_PW)
1777                {
1778                    s_attributes[i].continueSession = SET;
1779                }
1780            }
1781
1782            // Assemble Response Sessions.
1783            *resSessionSize = 0;
1784            buffer = resParmBuffer + resParmSize;
1785            for(i = 0; i < s_sessionNum; i++)
1786            {
1787                *resSessionSize += TPM2B_NONCE_Marshal(&responseNonces[i],
1788                                                      &buffer, NULL);
1789                *resSessionSize += TPMA_SESSION_Marshal(&s_attributes[i],
1790                                                       &buffer, NULL);
1791                *resSessionSize += TPM2B_DIGEST_Marshal(&responseAuths[i],
1792                                                       &buffer, NULL);
1793            }
1794
1795            // Update internal sessions after completing response buffer computation.
1796            UpdateInternalSession();
1797        }
1798        else
1799        {
1800            // Process command with no session.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1801            *resSessionSize = 0;
1802        }
1803
1804        return;
1805    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

8 Command Support Functions

8.1 Introduction

Clause 8 contains support routines that are called by the command action code in ISO/IEC 11889-3. The functions are grouped by the command group that is supported by the functions.

8.2 Attestation Command Support (Attest_spt.c)

8.2.1 Includes

```
#include "InternalRoutines.h"
#include "Attest_spt_fp.h"
```

8.2.2 Functions

8.2.2.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

Table 16

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	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*

```
TPM_RC
FillInAttestInfo(
    TPMI_DH_OBJECT       signHandle,    // IN: handle of signing object
    TPMT_SIG_SCHEME     *scheme,        // IN/OUT: scheme to be used for signing
    TPM2B_DATA          *data,          // IN: qualifying data
    TPMS_ATTEST         *attest         // OUT: attest structure
    )
{
    TPM_RC                  result;
    TPMI_RH_HIERARCHY       signHierarhcy;

    result = CryptSelectSignScheme(signHandle, scheme);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Magic number
    attest->magic = TPM_GENERATED_VALUE;

    if(signHandle == TPM_RH_NULL)
    {
        BYTE    *buffer;
        // For null sign handle, the QN is TPM_RH_NULL
        buffer = attest->qualifiedSigner.t.name;
        attest->qualifiedSigner.t.size =
            TPM_HANDLE_Marshal(&signHandle, &buffer, NULL);
    }
    else
    {
        // Certifying object qualified name
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
32          // if the scheme is anonymous, this is an empty buffer
33          if(CryptIsSchemeAnonymous(scheme->scheme))
34              attest->qualifiedSigner.t.size = 0;
35          else
36              ObjectGetQualifiedName(signHandle, &attest->qualifiedSigner);
37      }
38
39      // current clock in plain text
40      TimeFillInfo(&attest->clockInfo);
41
42      // Firmware version in plain text
43      attest->firmwareVersion = ((UINT64) gp.firmwareV1 << (<K>sizeof(UINT32) * 8));
44      attest->firmwareVersion += gp.firmwareV2;
45
46      // Get the hierarchy of sign object.  For NULL sign handle, the hierarchy
47      // will be TPM_RH_NULL
48      signHierarhcy = EntityGetHierarchy(signHandle);
49      if(signHierarhcy != TPM_RH_PLATFORM && signHierarhcy != TPM_RH_ENDORSEMENT)
50      {
51          // For sign object is not in platform or endorsement hierarchy,
52          // obfuscate the clock and firmwereVersion information
53          UINT64          obfuscation[2];
54          TPMI_ALG_HASH   hashAlg;
55
56          // Get hash algorithm
57          if(signHandle == TPM_RH_NULL || signHandle == TPM_RH_OWNER)
58          {
59              hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
60          }
61          else
62          {
63              OBJECT          *signObject = NULL;
64              signObject = ObjectGet(signHandle);
65              hashAlg = signObject->publicArea.nameAlg;
66          }
67          // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters"for normative KDF
68          // label values.
69          KDFa(hashAlg, &gp.shProof.b, "OBFUSCATE",
70               &attest->qualifiedSigner.b, NULL, 128, (BYTE *)&obfuscation[0], NULL);
71
72          // Obfuscate data
73          attest->firmwareVersion += obfuscation[0];
74          attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
75          attest->clockInfo.restartCount += (UINT32)obfuscation[1];
76      }
77
78      // External data
79      if(CryptIsSchemeAnonymous(scheme->scheme))
80          attest->extraData.t.size = 0;
81      else
82      {
83          // If we move the data to the attestation structure, then we will not use
84          // it in the signing operation except as part of the signed data
85          attest->extraData = *data;
86          data->t.size = 0;
87      }
88
89      return TPM_RC_SUCCESS;
90  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

8.2.2.2 SignAttestInfo()

Sign a TPMS_ATTEST structure. If *signHandle* is TPM_RH_NULL, a null signature is returned.

© ISO/IEC 2015 – All rights reserved

Table 17

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*signHandle* references not a signing key
TPM_RC_SCHEME	*scheme* is not compatible with *signHandle* type
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
TPM_RC
SignAttestInfo(
    TPMI_DH_OBJECT       signHandle,         // IN: handle of 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
                                             //     process
    TPM2B_ATTEST        *attest,             // OUT: marshaled attest blob to be
                                             //     signed
    TPMT_SIGNATURE      *signature           // OUT: signature
    )
{
    TPM_RC                  result;
    TPMI_ALG_HASH           hashAlg;
    BYTE                    *buffer;
    HASH_STATE              hashState;
    TPM2B_DIGEST            digest;


    // Marshal TPMS_ATTEST structure for hash
    buffer = attest->t.attestationData;
    attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);

    if(signHandle == TPM_RH_NULL)
    {
        signature->sigAlg = TPM_ALG_NULL;
    }
    else
    {
        // Attestation command may cause the orderlyState to be cleared due to
        // the reporting of clock info.  If this is the case, check if NV is
        // available first.
        if(gp.orderlyState != SHUTDOWN_NONE)
        {
            // The command needs NV update.  Check if NV is available.
            // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
            // this point
            result = NvIsAvailable();
            if(result != TPM_RC_SUCCESS)
                return result;
        }

        // Compute hash
        hashAlg = scheme->details.any.hashAlg;
        digest.t.size = CryptStartHash(hashAlg, &hashState);
        CryptUpdateDigest(&hashState, attest->t.size, attest->t.attestationData);
        CryptCompleteHash2B(&hashState, &digest.b);

        // If there is qualifying data, need to rehash the the data
        // hash(qualifyingData || hash(attestationData))
        if(qualifyingData->t.size != 0)
        {
            CryptStartHash(hashAlg, &hashState);
            CryptUpdateDigest(&hashState,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
                                qualifyingData->t.size,
                                qualifyingData->t.buffer);
            CryptUpdateDigest(&hashState, digest.t.size, digest.t.buffer);
            CryptCompleteHash2B(&hashState, &digest.b);
        }

        // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
        // TPM_RC_ATTRIBUTES error may be returned at this point
        return CryptSign(signHandle,
                         scheme,
                         &digest,
                         signature);
    }

    return TPM_RC_SUCCESS;
}
```

8.3 Context Management Command Support (Context_spt.c)

8.3.1 Includes

```
#include "InternalRoutines.h"
#include "Context_spt_fp.h"
```

8.3.2 Functions

8.3.2.1 ComputeContextProtectionKey()

This function retrieves the symmetric protection key for context encryption It is used by TPM2_ConextSave() and TPM2_ContextLoad() to create the symmetric encryption key and iv

```
void
ComputeContextProtectionKey(
    TPMS_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_AUTH      *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);
    MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence,
               sizeof(contextBlob->sequence),
               sizeof(sequence2B.t.buffer));

    // Get handle value in 2B format
    handle2B.t.size = sizeof(contextBlob->savedHandle);
    MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle,
               sizeof(contextBlob->savedHandle),
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
                sizeof(handle2B.t.buffer));

    // 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
    KDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, "CONTEXT", &sequence2B.b,
         &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL);

    // Copy part of the returned value as the key
    MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size,
                sizeof(symKey->t.buffer));

    // Copy the rest as the IV
    MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size,
                sizeof(iv->t.buffer));

    return;
}
```

8.3.2.2 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_AUTH          *proof;
    UINT16              integritySize;

    // Get proof value
    proof = HierarchyGetProof(contextBlob->hierarchy);

    // Start HMAC
    integrity->t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
                                         &proof->b, &hmacState);

    // 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
    CryptUpdateDigestInt(&hmacState, sizeof(gp.totalResetCount),
                         &gp.totalResetCount);

    // If this is a ST_CLEAR object, add the clear count
    // so that this contest cannot be loaded after a TPM Restart
    if(contextBlob->savedHandle == 0x80000002)
        CryptUpdateDigestInt(&hmacState, sizeof(gr.clearCount), &gr.clearCount);

    // Adding sequence number to the HMAC to make sure that it doesn't
    // get changed
    CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->sequence),
                         &contextBlob->sequence);

```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
91      // Protect the handle
92      CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->savedHandle),
93                           &contextBlob->savedHandle);
94
95      // Adding sensitive contextData, skip the leading integrity area
96      CryptUpdateDigest(&hmacState, contextBlob->contextBlob.t.size - integritySize,
97                        contextBlob->contextBlob.t.buffer + integritySize);
98
99      // Complete HMAC
100     CryptCompleteHMAC2B(&hmacState, &integrity->b);
101
102     return;
103 }
```

8.3.2.3 SequenceDataImportExport()

This function is used scan through the sequence object and either modify the hash state data for LIB_EXPORT or to import it into the internal format

```
104 void
105 SequenceDataImportExport(
106     OBJECT          *object,        // IN: the object containing the sequence data
107     OBJECT          *exportObject,  // IN/OUT: the object structure that will get
108                                     //     the exported hash state
109     IMPORT_EXPORT    direction
110     )
111 {
112     int                  count = 1;
113     HASH_OBJECT         *internalFmt = (HASH_OBJECT *)object;
114     HASH_OBJECT         *externalFmt = (HASH_OBJECT *)exportObject;
115
116
117     if(object->attributes.eventSeq)
118         count = HASH_COUNT;
119     for(; count; count--)
120         CryptHashStateImportExport(&internalFmt->state.hashState[count - 1],
121                                    externalFmt->state.hashState, direction);
122 }
```

8.4 Policy Command Support (Policy_spt.c)

8.4.1 Includes

```
1   #include "InternalRoutines.h"
2   #include "Policy_spt_fp.h"
3   #include "PolicySigned_fp.h"
4   #include "PolicySecret_fp.h"
5   #include "PolicyTicket_fp.h"
```

8.4.2 Functions

8.4.2.1 PolicyParameterChecks()

This function validates the common parameters of TPM2_PolicySiged() and TPM2_PolicySecret(). The common parameters are *nonceTPM*, *expiration*, and *cpHashA*.

```
6   TPM_RC
7   PolicyParameterChecks(
8       SESSION         *session,
9       UINT64           authTimeout,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
10      TPM2B_DIGEST    *cpHashA,
11      TPM2B_NONCE     *nonce,
12      TPM_RC           nonceParameterNumber,
13      TPM_RC           cpHashParameterNumber,
14      TPM_RC           expirationParameterNumber
15      )
16  {
17      TPM_RC           result;
18
19      // Validate that input nonceTPM is correct if present
20      if(nonce != NULL && nonce->t.size != 0)
21      {
22          if(!Memory2BEqual(&nonce->b, &session->nonceTPM.b))
23              return TPM_RC_NONCE + RC_PolicySigned_nonceTPM;
24      }
25      // If authTimeout is set (expiration != 0...
26      if(authTimeout != 0)
27      {
28          // ...then nonce must be present
29          // nonce present isn't checked in PolicyTicket
30          if(nonce != NULL && nonce->t.size == 0)
31              // This error says that the time has expired but it is pointing
32              // at the nonceTPM value.
33              return TPM_RC_EXPIRED + nonceParameterNumber;
34
35          // Validate input expiration.
36          // Cannot compare time if clock stop advancing.  A TPM_RC_NV_UNAVAILABLE
37          // or TPM_RC_NV_RATE error may be returned here.
38          result = NvIsAvailable();
39          if(result != TPM_RC_SUCCESS)
40              return result;
41
42          if(authTimeout < go.clock)
43              return TPM_RC_EXPIRED + expirationParameterNumber;
44      }
45      // If the cpHash is present, then check it
46      if(cpHashA != NULL && cpHashA->t.size != 0)
47      {
48          // The cpHash input has to have the correct size
49          if(cpHashA->t.size != session->u2.policyDigest.t.size)
50              return TPM_RC_SIZE + cpHashParameterNumber;
51
52          // If the cpHash has already been set, then this input value
53          // must match the current value.
54          if(     session->u1.cpHash.b.size != 0
55              &&  !Memory2BEqual(&cpHashA->b, &session->u1.cpHash.b))
56              return TPM_RC_CPHASH;
57      }
58      return TPM_RC_SUCCESS;
59  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

8.4.2.2 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.

```
60  void
61  PolicyContextUpdate(
62      TPM_CC           commandCode,   // IN: command code
63      TPM2B_NAME      *name,          // IN: name of entity
64      TPM2B_NONCE     *ref,           // IN: the reference data
65      TPM2B_DIGEST    *cpHash,        // IN: the cpHash (optional)
66      UINT64           policyTimeout,
67      SESSION         *session        // IN/OUT: policy session to be updated
```


© ISO/IEC 2015 – All rights reserved

```
    )
{
    HASH_STATE          hashState;
    UINT16              policyDigestSize;

    // Start hash
    policyDigestSize = CryptStartHash(session->authHashAlg, &hashState);

    // policyDigest size should always be the digest size of session hash algorithm.
    pAssert(session->u2.policyDigest.t.size == policyDigestSize);

    // add old digest
    CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);

    // add commandCode
    CryptUpdateDigestInt(&hashState, sizeof(commandCode), &commandCode);

    // add name if applicable
    if(name != NULL)
        CryptUpdateDigest2B(&hashState, &name->b);

    // Complete the digest and get the results
    CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);

    // Start second hash computation
    CryptStartHash(session->authHashAlg, &hashState);

    // add policyDigest
    CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);

    // add policyRef
    if(ref != NULL)
        CryptUpdateDigest2B(&hashState, &ref->b);

    // Complete second digest
    CryptCompleteHash2B(&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 iscoHashDefined attribute
    if(cpHash != NULL && cpHash->t.size != 0)
    {
        session->u1.cpHash = *cpHash;
        session->attributes.iscpHashDefined = SET;
    }

    // update the timeout if it is specified
    if(policyTimeout!= 0)
    {
    // If the timeout has not been set, then set it to the new value
        if(session->timeOut == 0)
            session->timeOut = policyTimeout;
        else if(session->timeOut > policyTimeout)
            session->timeOut =  policyTimeout;
    }
    return;
}
```

8.5 NV Command Support (NV_spt.c)

8.5.1 Includes

```
#include "InternalRoutines.h"
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
#include "NV_spt_fp.h"
```

8.5.2 Functions

8.5.2.1 NvReadAccessChecks()

Common routine for validating a read Used by TPM2_NV_Read(), TPM2_NV_ReadLock() and TPM2_PolicyNV().

Table 18

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	*authHandle* is not allowed to authorize read of the index
TPM_RC_NV_LOCKED	Read locked
TPM_RC_NV_UNINITIALIZED	Try to read an uninitialized index

```
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 written
    )
{
    NV_INDEX         nvIndex;

    // Get NV index info
    NvGetIndexInfo(nvHandle, &nvIndex);

// This check may be done before doing authorization checks as is done in this
// version of the reference code. If not done there, then uncomment the next
// three lines.
//    // If data is read locked, returns an error
//    if(nvIndex.publicArea.attributes.TPMA_NV_READLOCKED == SET)
//        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 auth then ONWERWRITE must be SET
        if(! nvIndex.publicArea.attributes.TPMA_NV_OWNERREAD)
            return TPM_RC_NV_AUTHORIZATION;
    }
    else if(authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided auth then PPWRITE must be SET
        if(!nvIndex.publicArea.attributes.TPMA_NV_PPREAD)
            return TPM_RC_NV_AUTHORIZATION;
    }
    // If neither Owner nor Platform provided auth, 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(nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
47          return TPM_RC_NV_UNINITIALIZED;
48
49      return TPM_RC_SUCCESS;
50  }
```

8.5.2.2 NvWriteAccessChecks()

Common routine for validating a write Used by TPM2_NV_Write(), TPM2_NV_Increment(), TPM2_SetBits(), and TPM2_NV_WriteLock().

Table 19

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	Authorization fails
TPM_RC_NV_LOCKED	Write locked

```
51  TPM_RC
52  NvWriteAccessChecks(
53      TPM_HANDLE       authHandle,    // IN: the handle that provided the
54                                      //     authorization
55      TPM_HANDLE       nvHandle       // IN: the handle of the NV index to be written
56      )
57  {
58      NV_INDEX         nvIndex;
59
60      // Get NV index info
61      NvGetIndexInfo(nvHandle, &nvIndex);
62
63  // This check may be done before doing authorization checks as is done in this
64  // version of the reference code. If not done there, then uncomment the next
65  // three lines.
66  //    // If data is write locked, returns an error
67  //    if(nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED == SET)
68  //        return TPM_RC_NV_LOCKED;
69
70      // If the authorization was provided by the owner or platform, then check
71      // that the attributes allow the write.  If the authorization handle
72      // is the same as the index, then the checks were made when the authorization
73      // was checked..
74      if(authHandle == TPM_RH_OWNER)
75      {
76          // If Owner provided auth then ONWERWRITE must be SET
77          if(! nvIndex.publicArea.attributes.TPMA_NV_OWNERWRITE)
78              return TPM_RC_NV_AUTHORIZATION;
79      }
80      else if(authHandle == TPM_RH_PLATFORM)
81      {
82          // If Platform provided auth then PPWRITE must be SET
83          if(!nvIndex.publicArea.attributes.TPMA_NV_PPWRITE)
84              return TPM_RC_NV_AUTHORIZATION;
85      }
86      // If neither Owner nor Platform provided auth, make sure that it was
87      // provided by this index.
88      else if(authHandle != nvHandle)
89              return TPM_RC_NV_AUTHORIZATION;
90
91      return TPM_RC_SUCCESS;
92  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

8.6 Object Command Support (Object_spt.c)

8.6.1 Includes

```
#include "InternalRoutines.h"
#include "Object_spt_fp.h"
#include <Platform.h>
```

8.6.2 Local Functions

8.6.2.1 EqualCryptSet()

Check if the crypto sets in two public areas are equal.

Table 20

Error Returns	Meaning
TPM_RC_ASYMMETRIC	mismatched parameters
TPM_RC_HASH	mismatched name algorithm
TPM_RC_TYPE	mismatched type

```
static TPM_RC
EqualCryptSet(
    TPMT_PUBLIC     *publicArea1,   // IN: public area 1
    TPMT_PUBLIC     *publicArea2    // IN: public area 2
    )
{
    UINT16              size1;
    UINT16              size2;
    BYTE                params1[sizeof(TPMU_PUBLIC_PARMS)];
    BYTE                params2[sizeof(TPMU_PUBLIC_PARMS)];
    BYTE                *buffer;

    // Compare name hash
    if(publicArea1->nameAlg != publicArea2->nameAlg)
        return TPM_RC_HASH;

    // Compare algorithm
    if(publicArea1->type != publicArea2->type)
        return TPM_RC_TYPE;

    // TPMU_PUBLIC_PARMS field should be identical
    buffer = params1;
    size1 = TPMU_PUBLIC_PARMS_Marshal(&publicArea1->parameters, &buffer,
                                      NULL, publicArea1->type);
    buffer = params2;
    size2 = TPMU_PUBLIC_PARMS_Marshal(&publicArea2->parameters, &buffer,
                                      NULL, publicArea2->type);

    if(size1 != size2 || !MemoryEqual(params1, params2, size1))
        return TPM_RC_ASYMMETRIC;

    return TPM_RC_SUCCESS;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

8.6.2.2 GetIV2BSize()

Get the size of TPM2B_IV in canonical form that will be append to the start of the sensitive data. It includes both size of size field and size of iv data.

```
37  static UINT16
38  GetIV2BSize(
39      TPM_HANDLE       protectorHandle    // IN: the protector handle
40      )
41  {
42      OBJECT              *protector = NULL; // Pointer to the protector object
43      TPM_ALG_ID          symAlg;
44      UINT16              keyBits;
45
46      // Determine the symmetric algorithm and size of key
47      if(protectorHandle == TPM_RH_NULL)
48      {
49          // Use the context encryption algorithm and key size
50          symAlg = CONTEXT_ENCRYPT_ALG;
51          keyBits = CONTEXT_ENCRYPT_KEY_BITS;
52      }
53      else
54      {
55          protector = ObjectGet(protectorHandle);
56          symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
57          keyBits= protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
58      }
59
60      // The IV size is a UINT16 size field plus the block size of the symmetric
61      // algorithm
62      return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
63  }
```

8.6.2.3 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.

```
64  static void
65  ComputeProtectionKeyParms(
66      TPM_HANDLE      protectorHandle,   // IN: the protector handle
67      TPM_ALG_ID      hashAlg,           // IN: hash algorithm for KDFa
68      TPM2B_NAME      *name,             // IN: name of the object
69      TPM2B_SEED      *seedIn,           // IN: optional seed for duplication blob.
70                                         //     For non duplication blob, this
71                                         //     parameter should be NULL
72      TPM_ALG_ID      *symAlg,           // OUT: the symmetric algorithm
73      UINT16          *keyBits,          // OUT: the symmetric key size in bits
74      TPM2B_SYM_KEY   *symKey            // OUT: the symmetric key
75      )
76  {
77      TPM2B_SEED          *seed = NULL;
78      OBJECT              *protector = NULL; // Pointer to the protector
79
80      // Determine the algorithms for the KDF and the encryption/decryption
81      // For TPM_RH_NULL, using context settings
82      if(protectorHandle == TPM_RH_NULL)
83      {
84          // Use the context encryption algorithm and key size
85          *symAlg = CONTEXT_ENCRYPT_ALG;
86          symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
87              *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
88          }
89          else
90          {
91              TPMT_SYM_DEF_OBJECT *symDef;
92              protector = ObjectGet(protectorHandle);
93              symDef = &protector->publicArea.parameters.asymDetail.symmetric;
94              *symAlg = symDef->algorithm;
95              *keyBits= symDef->keyBits.sym;
96              symKey->t.size = (*keyBits + 7) / 8;
97          }
98
99          // Get seed for KDF
100         seed = GetSeedForKDF(protectorHandle, seedIn);
101
102         // KDFa to generate symmetric key and IV value
103         // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters"
104         KDFa(hashAlg, (TPM2B *)seed, "STORAGE", (TPM2B *)name, NULL,
105              symKey->t.size * 8, symKey->t.buffer, NULL);
106
107         return;
108     }
```

8.6.2.4 ComputeOuterIntegrity()

The sensitive area parameter is a buffer that holds a space for the integrity value and the marshaled sensitive area. The caller should skip over the area set aside for the integrity value and compute the hash of the remainder of the object. The size field of sensitive is in unmarshaled form and the sensitive area contents is an array of bytes.

```
109 static void
110 ComputeOuterIntegrity(
111     TPM2B_NAME      *name,              // IN: the name of the object
112     TPM_HANDLE       protectorHandle,   // IN: The handle of the object that
113                                         //     provides protection.  For object, it
114                                         //     is parent handle. For credential, it
115                                         //     is the handle of encrypt object.  For
116                                         //     a Temporary Object, it is TPM_RH_NULL
117     TPMI_ALG_HASH    hashAlg,           // IN: algorithm to use for integrity
118     TPM2B_SEED      *seedIn,            // IN: an external seed may be provided for
119                                         //     duplication blob. For non duplication
120                                         //     blob, this parameter should be NULL
121     UINT32           sensitiveSize,     // IN: size of the marshaled sensitive data
122     BYTE            *sensitiveData,     // IN: sensitive area
123     TPM2B_DIGEST    *integrity          // OUT: integrity
124     )
125 {
126     HMAC_STATE          hmacState;
127
128     TPM2B_DIGEST        hmacKey;
129     TPM2B_SEED          *seed = NULL;
130
131     // Get seed for KDF
132     seed = GetSeedForKDF(protectorHandle, seedIn);
133
134     // Determine the HMAC key bits
135     hmacKey.t.size = CryptGetHashDigestSize(hashAlg);
136
137     // KDFa to generate HMAC key
138     // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters"
139     KDFa(hashAlg, (TPM2B *)seed, "INTEGRITY", NULL, NULL,
140          hmacKey.t.size * 8, hmacKey.t.buffer, NULL);
141
142     // Start HMAC and get the size of the digest which will become the integrity
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
143         integrity->t.size = CryptStartHMAC2B(hashAlg, &hmacKey.b, &hmacState);
144
145         // Adding the marshaled sensitive area to the integrity value
146         CryptUpdateDigest(&hmacState, sensitiveSize, sensitiveData);
147
148         // Adding name
149         CryptUpdateDigest2B(&hmacState, (TPM2B *)name);
150
151         // Compute HMAC
152         CryptCompleteHMAC2B(&hmacState, &integrity->b);
153
154         return;
155     }
```

8.6.2.5 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap.

```
156     static void
157     ComputeInnerIntegrity(
158         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
159         TPM2B_NAME      *name,          // IN: the name of the object
160         UINT16           dataSize,      // IN: the size of sensitive data
161         BYTE            *sensitiveData, // IN: sensitive data
162         TPM2B_DIGEST    *integrity      // OUT: inner integrity
163         )
164     {
165         HASH_STATE      hashState;
166
167         // Start hash and get the size of the digest which will become the integrity
168         integrity->t.size = CryptStartHash(hashAlg, &hashState);
169
170         // Adding the marshaled sensitive area to the integrity value
171         CryptUpdateDigest(&hashState, dataSize, sensitiveData);
172
173         // Adding name
174         CryptUpdateDigest2B(&hashState, &name->b);
175
176         // Compute hash
177         CryptCompleteHash2B(&hashState, &integrity->b);
178
179         return;
180
181     }
```

8.6.2.6 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 assume 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.

```
182     static UINT16
183     ProduceInnerIntegrity(
184         TPM2B_NAME      *name,          // IN: the name of the object
185         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
186         UINT16           dataSize,      // IN: the size of sensitive data, excluding the
187                                         //     leading integrity buffer size
188         BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
189                                         //     it.  At input, the leading bytes of this
190                                         //     buffer is reserved for integrity
191         )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
{
    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) + CryptGetHashDigestSize(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;
}
```

8.6.2.7 CheckInnerIntegrity()

This function check integrity of inner blob.

Table 21

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

```
static TPM_RC
CheckInnerIntegrity(
    TPM2B_NAME      *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(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
            result = TPM_RC_INTEGRITY;
    }
    return result;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
}
```

8.6.3 Public Functions

8.6.3.1 AreAttributesForParent()

This function is called by create, load, and import functions.

Table 22

Return Value	Meaning
TRUE	properties are those of a parent
FALSE	properties are not those of a parent

```
BOOL
AreAttributesForParent(
    OBJECT          *parentObject   // IN: parent handle
    )
{
    // This function is only called when a parent is needed.  Any
    // time a "parent" is used, it must be authorized. When
    // the authorization is checked, both the public and sensitive
    // areas must be loaded. Just make sure...
    pAssert(parentObject->attributes.publicOnly == CLEAR);


    if(ObjectDataIsStorage(&parentObject->publicArea))
        return TRUE;
    else
        return FALSE;
}
```

8.6.3.2 SchemeChecks()

This function validates the schemes in the public area of an object. This function is called by TPM2_LoadExternal() and PublicAttributesValidation().

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 23

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public parameters
TPM_RC_ATTRIBUTES	attempt to inject sensitive data for an asymmetric key; or attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed.hash object
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unexpected object type; or non-duplicable storage key and its parent have different types

```
261  TPM_RC
262  SchemeChecks(
263      BOOL             load,          // IN: TRUE if load checks, FALSE if
264                                      //     TPM2_Create()
265      TPMI_DH_OBJECT   parentHandle,  // IN: input parent handle
266      TPMT_PUBLIC     *publicArea     // IN: public area of the object
267      )
268  {
269
270      // Checks for an asymmetric key
271      if(CryptIsAsymAlgorithm(publicArea->type))
272      {
273          TPMT_ASYM_SCHEME        *keyScheme;
274          keyScheme = &publicArea->parameters.asymDetail.scheme;
275
276          // An asymmetric key can't be injected
277          // This is only checked when creating an object
278          if(!load && (publicArea->objectAttributes.sensitiveDataOrigin == CLEAR))
279              return TPM_RC_ATTRIBUTES;
280
281          if(load && !CryptAreKeySizesConsistent(publicArea))
282              return TPM_RC_KEY;
283
284          // Keys that are both signing and decrypting must have TPM_ALG_NULL
285          // for scheme
286          if(    publicArea->objectAttributes.sign == SET
287             && publicArea->objectAttributes.decrypt == SET
288             && keyScheme->scheme != TPM_ALG_NULL)
289              return TPM_RC_SCHEME;
290
291          // A restrict sign key must have a non-NULL scheme
292          if(     publicArea->objectAttributes.restricted == SET
293              &&  publicArea->objectAttributes.sign == SET
294              &&  keyScheme->scheme == TPM_ALG_NULL)
295              return TPM_RC_SCHEME;
296
297          // Keys must have a valid sign or decrypt scheme, or a TPM_ALG_NULL
298          // scheme
299          // NOTE: The unmarshaling for a public area will unmarshal based on the
300          // object type. If the type is an RSA key, then only RSA schemes will be
301          // allowed because a TPMI_ALG_RSA_SCHEME will be unmarshaled and it
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
302         // consists only of those algorithms that are allowed with an RSA key.
303         // This means that there is no need to again make sure that the algorithm
304         // is compatible with the object type.
305         if(   keyScheme->scheme != TPM_ALG_NULL
306            && (   (   publicArea->objectAttributes.sign == SET
307                    && !CryptIsSignScheme(keyScheme->scheme)
308                   )
309                || (   publicArea->objectAttributes.decrypt == SET
310                    && !CryptIsDecryptScheme(keyScheme->scheme)
311                   )
312               )
313           )
314             return TPM_RC_SCHEME;
315
316         // Special checks for an ECC key
317 #ifdef TPM_ALG_ECC
318         if(publicArea->type == TPM_ALG_ECC)
319         {
320             TPM_ECC_CURVE        curveID = publicArea->parameters.eccDetail.curveID;
321             const TPMT_ECC_SCHEME  *curveScheme = CryptGetCurveSignScheme(curveID);
322             // The curveId must be valid or the unmarshaling is busted.
323             pAssert(curveScheme != NULL);
324
325             // If the curveID requires a specific scheme, then the key must select
326             // the same scheme
327             if(curveScheme->scheme != TPM_ALG_NULL)
328             {
329                 if(keyScheme->scheme != curveScheme->scheme)
330                     return TPM_RC_SCHEME;
331                 // The scheme can allow any hash, or not...
332                 if(   curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
333                    && (   keyScheme->details.anySig.hashAlg
334                        != curveScheme->details.anySig.hashAlg
335                       )
336                   )
337                     return TPM_RC_SCHEME;
338             }
339             // For now, the KDF must be TPM_ALG_NULL
340             if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
341                 return TPM_RC_KDF;
342         }
343 #endif
344
345         // Checks for a storage key (restricted + decryption)
346         if(   publicArea->objectAttributes.restricted == SET
347            && publicArea->objectAttributes.decrypt == SET)
348         {
349             // A storage key must have a valid protection key
350             if(   publicArea->parameters.asymDetail.symmetric.algorithm
351                == TPM_ALG_NULL)
352                 return TPM_RC_SYMMETRIC;
353
354             // A storage key must have a null scheme
355             if(publicArea->parameters.asymDetail.scheme.scheme != TPM_ALG_NULL)
356                 return TPM_RC_SCHEME;
357
358             // A storage key must match its parent algorithms unless
359             // it is duplicable or a primary (including Temporary Primary Objects)
360             if(   HandleGetType(parentHandle) != TPM_HT_PERMANENT
361                && publicArea->objectAttributes.fixedParent == SET
362               )
363             {
364                 // If the object to be created is a storage key, and is fixedParent,
365                 // its crypto set has to match its parent's crypto set. TPM_RC_TYPE,
366                 // TPM_RC_HASH or TPM_RC_ASYMMETRIC may be returned at this point
367                 return EqualCryptSet(publicArea,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
368                                     &(ObjectGet(parentHandle)->publicArea));
369             }
370         }
371         else
372         {
373             // Non-storage keys must have TPM_ALG_NULL for the symmetric algorithm
374             if(   publicArea->parameters.asymDetail.symmetric.algorithm
375                != TPM_ALG_NULL)
376                 return TPM_RC_SYMMETRIC;
377
378         }// End of asymmetric decryption key checks
379     } // End of asymmetric checks
380
381     // Check for bit attributes
382     else if(publicArea->type == TPM_ALG_KEYEDHASH)
383     {
384         TPMT_KEYEDHASH_SCHEME   *scheme
385             = &publicArea->parameters.keyedHashDetail.scheme;
386         // If both sign and decrypt are set the scheme must be TPM_ALG_NULL
387         // and the scheme selected when the key is used.
388         // If neither sign nor decrypt is set, the scheme must be TPM_ALG_NULL
389         // because this is a data object.
390         if(     publicArea->objectAttributes.sign
391             ==  publicArea->objectAttributes.decrypt)
392         {
393             if(scheme->scheme != TPM_ALG_NULL)
394                 return TPM_RC_SCHEME;
395             return TPM_RC_SUCCESS;
396         }
397         // If this is a decryption key, make sure that is is XOR and that there
398         // is a KDF
399         else if(publicArea->objectAttributes.decrypt)
400         {
401             if(   scheme->scheme != TPM_ALG_XOR
402                || scheme->details.xor.hashAlg == TPM_ALG_NULL)
403                 return TPM_RC_SCHEME;
404             if(scheme->details.xor.kdf == TPM_ALG_NULL)
405                 return TPM_RC_KDF;
406             return TPM_RC_SUCCESS;
407
408         }
409         // only supported signing scheme for keyedHash object is HMAC
410         if(   scheme->scheme != TPM_ALG_HMAC
411            || scheme->details.hmac.hashAlg == TPM_ALG_NULL)
412             return TPM_RC_SCHEME;
413
414         // end of the checks for keyedHash
415         return TPM_RC_SUCCESS;
416     }
417     else if (publicArea->type == TPM_ALG_SYMCIPHER)
418     {
419         // Must be a decrypting key and may not be a signing key
420         if(   publicArea->objectAttributes.decrypt == CLEAR
421            || publicArea->objectAttributes.sign == SET
422           )
423             return TPM_RC_ATTRIBUTES;
424     }
425     else
426         return TPM_RC_TYPE;
427
428     return TPM_RC_SUCCESS;
429 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

8.6.3.3 PublicAttributesValidation()

This function validates the values in the public area of an object. This function is called by TPM2_Create(), TPM2_Load(), and TPM2_CreatePrimary().

Table 24

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public parameters
TPM_RC_ATTRIBUTES	*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
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	*authPolicy* size does not match digest size of the name algorithm in *publicArea*
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unexpected object type; or non-duplicable storage key and its parent have different types

```
430  TPM_RC
431  PublicAttributesValidation(
432      BOOL              load,          // IN: TRUE if load checks, FALSE if
433                                       //     TPM2_Create()
434      TPMI_DH_OBJECT    parentHandle,  // IN: input parent handle
435      TPMT_PUBLIC      *publicArea     // IN: public area of the object
436      )
437  {
438      OBJECT            *parentObject = NULL;
439
440      if(HandleGetType(parentHandle) != TPM_HT_PERMANENT)
441          parentObject = ObjectGet(parentHandle);
442
443      // Check authPolicy digest consistency
444      if(    publicArea->authPolicy.t.size != 0
445         && (   publicArea->authPolicy.t.size
446             != CryptGetHashDigestSize(publicArea->nameAlg)
447            )
448        )
449          return TPM_RC_SIZE;
450
451      // If the parent is fixedTPM (including a Primary Object) the object must have
452      // the same value for fixedTPM and fixedParent
453      if(    parentObject == NULL
454          || parentObject->publicArea.objectAttributes.fixedTPM == SET)
455      {
456          if(   publicArea->objectAttributes.fixedParent
457             != publicArea->objectAttributes.fixedTPM
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
458             )
459               return TPM_RC_ATTRIBUTES;
460         }
461         else
462             // The parent is not fixedTPM so the object can't be fixedTPM
463             if(publicArea->objectAttributes.fixedTPM == SET)
464                 return  TPM_RC_ATTRIBUTES;
465
466         // A restricted object cannot be both sign and decrypt and it can't be neither
467         // sign nor decrypt
468         if (   publicArea->objectAttributes.restricted == SET
469             && (   publicArea->objectAttributes.decrypt
470                 == publicArea->objectAttributes.sign)
471            )
472             return TPM_RC_ATTRIBUTES;
473
474         // A fixedTPM object can not have encryptedDuplication bit SET
475         if(    publicArea->objectAttributes.fixedTPM == SET
476             && publicArea->objectAttributes.encryptedDuplication == SET)
477             return TPM_RC_ATTRIBUTES;
478
479         // If a parent object has fixedTPM CLEAR, the child must have the
480         // same encryptedDuplication value as its parent.
481         // Primary objects are considered to have a fixedTPM parent (the seeds).
482        if(      (   parentObject != NULL
483                  && parentObject->publicArea.objectAttributes.fixedTPM == CLEAR)
484            // Get here if parent is not fixed TPM
485            &&  (   publicArea->objectAttributes.encryptedDuplication
486                 != parentObject->publicArea.objectAttributes.encryptedDuplication
487                )
488          )
489             return TPM_RC_ATTRIBUTES;
490
491       return SchemeChecks(load, parentHandle, publicArea);
492    }
```

8.6.3.4 FillInCreationData()

Fill in creation data for an object.

```
493    void
494    FillInCreationData(
495        TPMI_DH_OBJECT           parentHandle,  // IN: handle of parent
496        TPMI_ALG_HASH            nameHashAlg,   // IN: name hash algorithm
497        TPML_PCR_SELECTION      *creationPCR,   // IN: PCR selection
498        TPM2B_DATA              *outsideData,   // IN: outside data
499        TPM2B_CREATION_DATA     *outCreation,   // OUT: creation data for output
500        TPM2B_DIGEST            *creationDigest // OUT: creation digest
501        )
502    {
503        BYTE                 creationBuffer[sizeof(TPMS_CREATION_DATA)];
504        BYTE                *buffer;
505        HASH_STATE           hashState;
506
507        // Fill in TPMS_CREATION_DATA in outCreation
508
509        // Compute PCR digest
510        PCRComputeCurrentDigest(nameHashAlg, creationPCR,
511                                &outCreation->t.creationData.pcrDigest);
512
513        // Put back PCR selection list
514        outCreation->t.creationData.pcrSelect = *creationPCR;
515
516        // Get locality
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
517         outCreation->t.creationData.locality
518             = LocalityGetAttributes(_plat__LocalityGet());
519 
520         outCreation->t.creationData.parentNameAlg = TPM_ALG_NULL;
521 
522         // If the parent is is either a primary seed or TPM_ALG_NULL, then  the Name
523         // and QN of the parent are the parent's handle.
524         if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
525         {
526             BYTE        *buffer = &outCreation->t.creationData.parentName.t.name[0];
527             outCreation->t.creationData.parentName.t.size =
528                 TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
529 
530             // Parent qualified name of a Temporary Object is the same as parent's
531             // name
532             MemoryCopy2B(&outCreation->t.creationData.parentQualifiedName.b,
533                          &outCreation->t.creationData.parentName.b,
534                          sizeof(outCreation->t.creationData.parentQualifiedName.t.name));
535 
536         }
537         else         // Regular object
538         {
539             OBJECT          *parentObject = ObjectGet(parentHandle);
540 
541             // Set name algorithm
542             outCreation->t.creationData.parentNameAlg =
543                 parentObject->publicArea.nameAlg;
544             // Copy parent name
545             outCreation->t.creationData.parentName = parentObject->name;
546 
547             // Copy parent qualified name
548             outCreation->t.creationData.parentQualifiedName =
549                 parentObject->qualifiedName;
550         }
551 
552         // Copy outside information
553         outCreation->t.creationData.outsideInfo = *outsideData;
554 
555         // Marshal creation data to canonical form
556         buffer = creationBuffer;
557         outCreation->t.size = TPMS_CREATION_DATA_Marshal(&outCreation->t.creationData,
558                                 &buffer, NULL);
559 
560         // Compute hash for creation field in public template
561         creationDigest->t.size = CryptStartHash(nameHashAlg, &hashState);
562         CryptUpdateDigest(&hashState, outCreation->t.size, creationBuffer);
563         CryptCompleteHash2B(&hashState, &creationDigest->b);
564 
565         return;
566     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

8.6.3.5 GetSeedForKDF()

Get a seed for KDF. The KDF for encryption and HMAC key use the same seed. It returns a pointer to the seed.

```
567     TPM2B_SEED*
568     GetSeedForKDF(
569         TPM_HANDLE       protectorHandle,   // IN: the protector handle
570         TPM2B_SEED      *seedIn             // IN: the optional input seed
571         )
572     {
573         OBJECT              *protector = NULL; // Pointer to the protector
574 
```

© ISO/IEC 2015 – All rights reserved

```
575         // Get seed for encryption key.  Use input seed if provided.
576         // Otherwise, using protector object's seedValue.  TPM_RH_NULL is the only
577         // exception that we may not have a loaded object as protector.  In such a
578         // case, use nullProof as seed.
579         if(seedIn != NULL)
580         {
581             return seedIn;
582         }
583         else
584         {
585             if(protectorHandle == TPM_RH_NULL)
586             {
587                 return (TPM2B_SEED *) &gr.nullProof;
588             }
589             else
590             {
591                 protector = ObjectGet(protectorHandle);
592                 return (TPM2B_SEED *) &protector->sensitive.seedValue;
593             }
594         }
595     }
```

8.6.3.6 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.

```
596     UINT16
597     ProduceOuterWrap(
598         TPM_HANDLE       protector,     // IN: The handle of the object that provides
599                                         //     protection.  For object, it is parent
600                                         //     handle. For credential, it is the handle
601                                         //     of encrypt object.
602         TPM2B_NAME      *name,          // IN: the name of the object
603         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
604         TPM2B_SEED      *seed,          // IN: an external seed may be provided for
605                                         //     duplication blob. For non duplication
606                                         //     blob, this parameter should be NULL
607         BOOL             useIV,         // IN: indicate if an IV is used
608         UINT16           dataSize,      // IN: the size of sensitive data, excluding the
609                                         //     leading integrity buffer size or the
610                                         //     optional iv size
611         BYTE            *outerBuffer    // IN/OUT: outer buffer with sensitive data in
612                                         //     it
613         )
614     {
615         TPM_ALG_ID      symAlg;
616         UINT16          keyBits;
617         TPM2B_SYM_KEY   symKey;
618         TPM2B_IV        ivRNG;          // IV from RNG
619         TPM2B_IV        *iv = NULL;
620         UINT16          ivSize = 0;     // size of iv area, including the size field
621
622         BYTE            *sensitiveData; // pointer to the sensitive data
623
624         TPM2B_DIGEST    integrity;
625         UINT16          integritySize;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
626         BYTE            *buffer;        // Auxiliary buffer pointer
627
628         // Compute the beginning of sensitive data.  The outer integrity should
629         // always exist if this function function is called to make an outer wrap
630         integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
631         sensitiveData = outerBuffer + integritySize;
632
633         // If iv is used, adjust the pointer of sensitive data and add iv before it
634         if(useIV)
635         {
636             ivSize = GetIV2BSize(protector);
637
638             // Generate IV from RNG.  The iv data size should be the total IV area
639             // size minus the size of size field
640             ivRNG.t.size = ivSize - sizeof(UINT16);
641             CryptGenerateRandom(ivRNG.t.size, ivRNG.t.buffer);
642
643             // Marshal IV to buffer
644             buffer = sensitiveData;
645             TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
646
647             // adjust sensitive data starting after IV area
648             sensitiveData += ivSize;
649
650             // Use iv for encryption
651             iv = &ivRNG;
652         }
653
654         // Compute symmetric key parameters for outer buffer encryption
655         ComputeProtectionKeyParms(protector, hashAlg, name, seed,
656                                   &symAlg, &keyBits, &symKey);
657         // Encrypt inner buffer in place
658         CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
659                               TPM_ALG_CFB, symKey.t.buffer, iv, dataSize,
660                               sensitiveData);
661
662         // Compute outer integrity.  Integrity computation includes the optional IV
663         // area
664         ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
665                               outerBuffer + integritySize, &integrity);
666
667         // Add integrity at the beginning of outer buffer
668         buffer = outerBuffer;
669         TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
670
671         // return the total size in outer wrap
672         return dataSize + integritySize + ivSize;
673
674     }
```

8.6.3.7 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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 25

Error Returns	Meaning
TPM_RC_INSUFFICIENT	error during sensitive data unmarshaling
TPM_RC_INTEGRITY	sensitive data integrity is broken
TPM_RC_SIZE	error during sensitive data unmarshaling
TPM_RC_VALUE	IV size for CFB does not match the encryption algorithm block size

```
675  TPM_RC
676  UnwrapOuter(
677      TPM_HANDLE       protector,      // IN: The handle of the object that provides
678                                       //     protection.  For object, it is parent
679                                       //     handle. For credential, it is the handle
680                                       //     of encrypt object.
681      TPM2B_NAME      *name,           // IN: the name of the object
682      TPM_ALG_ID       hashAlg,        // IN: hash algorithm for outer wrap
683      TPM2B_SEED      *seed,           // IN: an external seed may be provided for
684                                       //     duplication blob. For non duplication
685                                       //     blob, this parameter should be NULL.
686      BOOL             useIV,          // IN: indicates if an IV is used
687      UINT16           dataSize,       // IN: size of sensitive data in outerBuffer,
688                                       //     including the leading integrity buffer
689                                       //     size, and an optional iv area
690      BYTE            *outerBuffer     // IN/OUT: sensitive data
691      )
692  {
693      TPM_RC          result;
694      TPM_ALG_ID      symAlg = TPM_ALG_NULL;
695      TPM2B_SYM_KEY   symKey;
696      UINT16          keyBits = 0;
697      TPM2B_IV        ivIn;               // input IV retrieved from input buffer
698      TPM2B_IV        *iv = NULL;
699
700      BYTE            *sensitiveData;     // pointer to the sensitive data
701
702      TPM2B_DIGEST    integrityToCompare;
703      TPM2B_DIGEST    integrity;
704      INT32           size;
705
706      // Unmarshal integrity
707      sensitiveData = outerBuffer;
708      size = (INT32) dataSize;
709      result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
710      if(result == TPM_RC_SUCCESS)
711      {
712          // Compute integrity to compare
713          ComputeOuterIntegrity(name, protector, hashAlg, seed,
714                                (UINT16) size, sensitiveData,
715                                &integrityToCompare);
716
717          // Compare outer blob integrity
718          if(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
719              return TPM_RC_INTEGRITY;
720
721          // Get the symmetric algorithm parameters used for encryption
722          ComputeProtectionKeyParms(protector, hashAlg, name, seed,
723                                    &symAlg, &keyBits, &symKey);
724
725          // Retrieve IV if it is used
726          if(useIV)
727          {
728              result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
729             if(result == TPM_RC_SUCCESS)
730             {
731                 // The input iv size for CFB must match the encryption algorithm
732                 // block size
733                 if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
734                     result = TPM_RC_VALUE;
735                 else
736                     iv = &ivIn;
737             }
738         }
739     }
740     // If no errors, decrypt private in place
741     if(result == TPM_RC_SUCCESS)
742         CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
743                               TPM_ALG_CFB, symKey.t.buffer, iv,
744                               (UINT16) size, sensitiveData);
745
746     return result;
747
748 }
```

8.6.3.8 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.

```
749 void
750 SensitiveToPrivate(
751     TPMT_SENSITIVE  *sensitive,     // IN: sensitive structure
752     TPM2B_NAME      *name,          // IN: the name of the object
753     TPM_HANDLE       parentHandle,  // IN: The parent's handle
754     TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  This
755                                     //     parameter is used when parentHandle is
756                                     //     NULL, in which case the object is
757                                     //     temporary.
758     TPM2B_PRIVATE   *outPrivate     // OUT: output private structure
759     )
760 {
761     BYTE               *buffer;             // Auxiliary buffer pointer
762     BYTE               *sensitiveData;      // pointer to the sensitive data
763     UINT16              dataSize;           // data blob size
764     TPMI_ALG_HASH       hashAlg;            // hash algorithm for integrity
765     UINT16              integritySize;
766     UINT16              ivSize;
767
768     pAssert(name != NULL && name->t.size != 0);
769
770     // Find the hash algorithm for integrity computation
771     if(parentHandle == TPM_RH_NULL)
772     {
773         // For Temporary Object, using self name algorithm
774         hashAlg = nameAlg;
775     }
776     else
777     {
778         // Otherwise, using parent's name algorithm
779         hashAlg = ObjectGetNameAlg(parentHandle);
780     }
781
782     // Starting of sensitive data without wrappers
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
783         sensitiveData = outPrivate->t.buffer;
784
785         // Compute the integrity size
786         integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
787
788         // Reserve space for integrity
789         sensitiveData += integritySize;
790
791         // Get iv size
792         ivSize = GetIV2BSize(parentHandle);
793
794         // Reserve space for iv
795         sensitiveData += ivSize;
796
797         // Marshal sensitive area, leaving the leading 2 bytes for size
798         buffer = sensitiveData + sizeof(UINT16);
799         dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
800
801         // Adding size before the data area
802         buffer = sensitiveData;
803         UINT16_Marshal(&dataSize, &buffer, NULL);
804
805         // Adjust the dataSize to include the size field
806         dataSize += sizeof(UINT16);
807
808         // Adjust the pointer to inner buffer including the iv
809         sensitiveData = outPrivate->t.buffer + ivSize;
810
811         //Produce outer wrap, including encryption and HMAC
812         outPrivate->t.size = ProduceOuterWrap(parentHandle, name, hashAlg, NULL,
813                                               TRUE, dataSize, outPrivate->t.buffer);
814
815         return;
816     }
```

8.6.3.9 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 26

Error Returns	Meaning
TPM_RC_INTEGRITY	if the private area integrity is bad
TPM_RC_SENSITIVE	unmarshal errors while unmarshaling TPMS_ENCRYPT from input private
TPM_RC_VALUE	outer wrapper does not have an *iV* of the correct size

```
817     TPM_RC
818     PrivateToSensitive(
819         TPM2B_PRIVATE   *inPrivate,     // IN: input private structure
820         TPM2B_NAME      *name,          // IN: the name of the object
821         TPM_HANDLE       parentHandle,  // IN: The parent's handle
822         TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  It is
823                                         //     passed separately because we only pass
824                                         //     name, rather than the whole public area
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
825                                     //     of the object.  This parameter is used in
826                                     //     the following two cases: 1. primary
827                                     //     objects. 2. duplication blob with inner
828                                     //     wrap.  In other cases, this parameter
829                                     //     will be ignored
830         TPMT_SENSITIVE  *sensitive      // OUT: sensitive structure
831         )
832     {
833         TPM_RC          result;
834
835         BYTE            *buffer;
836         INT32           size;
837         BYTE            *sensitiveData; // pointer to the sensitive data
838         UINT16          dataSize;
839         UINT16          dataSizeInput;
840         TPMI_ALG_HASH   hashAlg;        // hash algorithm for integrity
841         OBJECT          *parent = NULL;
842
843         UINT16          integritySize;
844         UINT16          ivSize;
845
846         // Make sure that name is provided
847         pAssert(name != NULL && name->t.size != 0);
848
849         // Find the hash algorithm for integrity computation
850         if(parentHandle == TPM_RH_NULL)
851         {
852             // For Temporary Object, using self name algorithm
853             hashAlg = nameAlg;
854         }
855         else
856         {
857             // Otherwise, using parent's name algorithm
858             hashAlg = ObjectGetNameAlg(parentHandle);
859         }
860
861         // unwrap outer
862         result = UnwrapOuter(parentHandle, name, hashAlg, NULL, TRUE,
863                              inPrivate->t.size, inPrivate->t.buffer);
864         if(result != TPM_RC_SUCCESS)
865             return result;
866
867         // Compute the inner integrity size.
868         integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
869
870         // Get iv size
871         ivSize = GetIV2BSize(parentHandle);
872
873         // The starting of sensitive data and data size without outer wrapper
874         sensitiveData = inPrivate->t.buffer + integritySize + ivSize;
875         dataSize = inPrivate->t.size - integritySize - ivSize;
876
877         // Unmarshal input data size
878         buffer = sensitiveData;
879         size = (INT32) dataSize;
880         result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
881         if(result == TPM_RC_SUCCESS)
882         {
883             if((dataSizeInput + sizeof(UINT16)) != dataSize)
884                 result = TPM_RC_SENSITIVE;
885             else
886             {
887                 // Unmarshal sensitive buffer to sensitive structure
888                 result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
889                 if(result != TPM_RC_SUCCESS || size != 0)
890                 {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
891                 pAssert(   (parent == NULL)
892                         || parent->publicArea.objectAttributes.fixedTPM == CLEAR);
893                 result = TPM_RC_SENSITIVE;
894             }
895             else
896             {
897                 // Always remove trailing zeros at load so that it is not necessary
898                 // to check
899                 // each time auth is checked.
900                 MemoryRemoveTrailingZeros(&(sensitive->authValue));
901             }
902         }
903     }
904     return result;
905 }
```

8.6.3.10 SensitiveToDuplicate()

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

a) marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE

b) apply inner wrap to the sensitive area if required

c) apply outer wrap if required

```
906 void
907 SensitiveToDuplicate(
908     TPMT_SENSITIVE          *sensitive,     // IN: sensitive structure
909     TPM2B_NAME              *name,          // IN: the name of the object
910     TPM_HANDLE               parentHandle,  // IN: The new parent's handle
911     TPM_ALG_ID               nameAlg,       // IN: hash algorithm in public area.
912                                             //     It is passed separately because
913                                             //     we only pass name, rather than
914                                             //     the whole public area of the
915                                             //     object.
916     TPM2B_SEED              *seed,          // IN: the external seed. If external
917                                             //     seed is provided with size of 0,
918                                             //     no outer wrap should be applied
919                                             //     to duplication blob.
920     TPMT_SYM_DEF_OBJECT     *symDef,        // IN: Symmetric key definition. If the
921                                             //     symmetric key algorithm is NULL,
922                                             //     no inner wrap should be applied.
923     TPM2B_DATA              *innerSymKey,   // IN/OUT: a symmetric key may be
924                                             //     provided to encrypt the inner
925                                             //     wrap of a duplication blob. May
926                                             //     be generated here if needed.
927     TPM2B_PRIVATE           *outPrivate     // OUT: output private structure
928     )
929 {
930     BYTE            *buffer;        // Auxiliary buffer pointer
931     BYTE            *sensitiveData; // pointer to the sensitive data
932     TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;// The hash algorithm for outer wrap
933     TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;// The hash algorithm for inner wrap
934     UINT16          dataSize;       // data blob size
935     BOOL            doInnerWrap = FALSE;
936     BOOL            doOuterWrap = FALSE;
937
938     // Make sure that name is provided
939     pAssert(name != NULL && name->t.size != 0);
940
941     // Make sure symDef and innerSymKey are not NULL
942     pAssert(symDef != NULL && innerSymKey != NULL);
943
944     // Starting of sensitive data without wrappers
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
945         sensitiveData = outPrivate->t.buffer;
946
947         // Find out if inner wrap is required
948         if(symDef->algorithm != TPM_ALG_NULL)
949         {
950             doInnerWrap = TRUE;
951             // Use self nameAlg as inner hash algorithm
952             innerHash = nameAlg;
953             // Adjust sensitive data pointer
954             sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
955         }
956
957         // Find out if outer wrap is required
958         if(seed->t.size != 0)
959         {
960             doOuterWrap = TRUE;
961             // Use parent nameAlg as outer hash algorithm
962             outerHash = ObjectGetNameAlg(parentHandle);
963             // Adjust sensitive data pointer
964             sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
965         }
966
967         // Marshal sensitive area, leaving the leading 2 bytes for size
968         buffer = sensitiveData + sizeof(UINT16);
969         dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
970
971         // Adding size before the data area
972         buffer = sensitiveData;
973         UINT16_Marshal(&dataSize, &buffer, NULL);
974
975         // Adjust the dataSize to include the size field
976         dataSize += sizeof(UINT16);
977
978         // Apply inner wrap for duplication blob.  It includes both integrity and
979         // encryption
980         if(doInnerWrap)
981         {
982             BYTE            *innerBuffer = NULL;
983             BOOL            symKeyInput = TRUE;
984             innerBuffer = outPrivate->t.buffer;
985             // Skip outer integrity space
986             if(doOuterWrap)
987                 innerBuffer += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
988             dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
989                                              innerBuffer);
990
991             // Generate inner encryption key if needed
992             if(innerSymKey->t.size == 0)
993             {
994                 innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
995                 CryptGenerateRandom(innerSymKey->t.size, innerSymKey->t.buffer);
996
997                 // TPM generates symmetric encryption.  Set the flag to FALSE
998                 symKeyInput = FALSE;
999             }
1000            else
1001            {
1002                // assume the input key size should matches the symmetric definition
1003                pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1004
1005            }
1006
1007            // Encrypt inner buffer in place
1008            CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
1009                                  symDef->keyBits.sym, TPM_ALG_CFB,
1010                                  innerSymKey->t.buffer, NULL, dataSize,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
                                innerBuffer);

        // If the symmetric encryption key is imported, clear the buffer for
        // output
        if(symKeyInput)
            innerSymKey->t.size = 0;
    }

    // Apply outer wrap for duplication blob.  It includes both integrity and
    // encryption
    if(doOuterWrap)
    {
        dataSize = ProduceOuterWrap(parentHandle, name, outerHash, seed, FALSE,
                                    dataSize, outPrivate->t.buffer);
    }

    // Data size for output
    outPrivate->t.size = dataSize;

    return;
}
```

8.6.3.11 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 27

Error Returns	Meaning
TPM_RC_INSUFFICIENT	unmarshaling sensitive data from *inPrivate* failed
TPM_RC_INTEGRITY	*inPrivate* data integrity is broken
TPM_RC_SIZE	unmarshaling sensitive data from *inPrivate* failed

```
TPM_RC
DuplicateToSensitive(
    TPM2B_PRIVATE           *inPrivate,     // IN: input private structure
    TPM2B_NAME              *name,          // IN: the name of the object
    TPM_HANDLE               parentHandle,  // IN: The parent's handle
    TPM_ALG_ID               nameAlg,       // IN: hash algorithm in public area.
    TPM2B_SEED              *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_DATA              *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;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1054        INT32           size;
1055        BYTE            *sensitiveData; // pointer to the sensitive data
1056        UINT16          dataSize;
1057        UINT16          dataSizeInput;
1058
1059        // Make sure that name is provided
1060        pAssert(name != NULL && name->t.size != 0);
1061
1062        // Make sure symDef and innerSymKey are not NULL
1063        pAssert(symDef != NULL && innerSymKey != NULL);
1064
1065        // Starting of sensitive data
1066        sensitiveData = inPrivate->t.buffer;
1067        dataSize = inPrivate->t.size;
1068
1069        // Find out if outer wrap is applied
1070        if(seed->t.size != 0)
1071        {
1072            TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;
1073
1074            // Use parent nameAlg as outer hash algorithm
1075            outerHash = ObjectGetNameAlg(parentHandle);
1076            result = UnwrapOuter(parentHandle, name, outerHash, seed, FALSE,
1077                                 dataSize, sensitiveData);
1078            if(result != TPM_RC_SUCCESS)
1079                return result;
1080
1081            // Adjust sensitive data pointer and size
1082            sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1083            dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1084        }
1085        // Find out if inner wrap is applied
1086        if(symDef->algorithm != TPM_ALG_NULL)
1087        {
1088            TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;
1089
1090            // assume the input key size should matches the symmetric definition
1091            pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1092
1093            // Decrypt inner buffer in place
1094            CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
1095                                  symDef->keyBits.sym, TPM_ALG_CFB,
1096                                  innerSymKey->t.buffer, NULL, dataSize,
1097                                  sensitiveData);
1098
1099            // Use self nameAlg as inner hash algorithm
1100            innerHash = nameAlg;
1101
1102            // Check inner integrity
1103            result = CheckInnerIntegrity(name, innerHash, dataSize, sensitiveData);
1104            if(result != TPM_RC_SUCCESS)
1105                return result;
1106
1107            // Adjust sensitive data pointer and size
1108            sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
1109            dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
1110        }
1111
1112        // Unmarshal input data size
1113        buffer = sensitiveData;
1114        size = (INT32) dataSize;
1115        result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1116        if(result == TPM_RC_SUCCESS)
1117        {
1118            if((dataSizeInput + sizeof(UINT16)) != dataSize)
1119                result = TPM_RC_SIZE;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1120            else
1121            {
1122                // Unmarshal sensitive buffer to sensitive structure
1123                result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1124                // if the results is OK make sure that all the data was unmarshaled
1125                if(result == TPM_RC_SUCCESS && size != 0)
1126                    result = TPM_RC_SIZE;
1127            }
1128        }
1129        // Always remove trailing zeros at load so that it is not necessary to check
1130        // each time auth is checked.
1131        if(result == TPM_RC_SUCCESS)
1132            MemoryRemoveTrailingZeros(&(sensitive->authValue));
1133        return result;
1134    }
```

8.6.3.12 SecretToCredential()

This function prepare 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

```
1135    void
1136    SecretToCredential(
1137        TPM2B_DIGEST        *secret,        // IN: secret information
1138        TPM2B_NAME          *name,          // IN: the name of the object
1139        TPM2B_SEED          *seed,          // IN: an external seed.
1140        TPM_HANDLE           protector,     // IN: The protector's handle
1141        TPM2B_ID_OBJECT     *outIDObject    // OUT: output credential
1142        )
1143    {
1144        BYTE                *buffer;        // Auxiliary buffer pointer
1145        BYTE                *sensitiveData; // pointer to the sensitive data
1146        TPMI_ALG_HASH        outerHash;     // The hash algorithm for outer wrap
1147        UINT16               dataSize;      // data blob size
1148
1149        pAssert(secret != NULL && outIDObject != NULL);
1150
1151        // use protector's name algorithm as outer hash
1152        outerHash = ObjectGetNameAlg(protector);
1153
1154        // Marshal secret area to credential buffer, leave space for integrity
1155        sensitiveData = outIDObject->t.credential
1156                      + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1157
1158        // Marshal secret area
1159        buffer = sensitiveData;
1160        dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1161
1162        // Apply outer wrap
1163        outIDObject->t.size = ProduceOuterWrap(protector,
1164                                               name,
1165                                               outerHash,
1166                                               seed,
1167                                               FALSE,
1168                                               dataSize,
1169                                               outIDObject->t.credential);
1170        return;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
}
```

8.6.3.13 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 28

Error Returns	Meaning
TPM_RC_INSUFFICIENT	error during credential unmarshaling
TPM_RC_INTEGRITY	credential integrity is broken
TPM_RC_SIZE	error during credential unmarshaling
TPM_RC_VALUE	IV size does not match the encryption algorithm block size

```
TPM_RC
CredentialToSecret(
    TPM2B_ID_OBJECT     *inIDObject,    // IN: input credential blob
    TPM2B_NAME          *name,          // IN: the name of the object
    TPM2B_SEED          *seed,          // IN: an external seed.
    TPM_HANDLE           protector,     // IN: The protector's handle
    TPM2B_DIGEST        *secret         // OUT: secret information
    )
{
    TPM_RC                   result;
    BYTE                    *buffer;
    INT32                    size;
    TPMI_ALG_HASH            outerHash;     // The hash algorithm for outer wrap
    BYTE                    *sensitiveData; // pointer to the sensitive data
    UINT16                   dataSize;

    // use protector's name algorithm as outer hash
    outerHash = ObjectGetNameAlg(protector);

    // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
    result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
                         inIDObject->t.size, inIDObject->t.credential);
    if(result == TPM_RC_SUCCESS)
    {
        // Compute the beginning of sensitive data
        sensitiveData = inIDObject->t.credential
                      + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
        dataSize = inIDObject->t.size
                 - (sizeof(UINT16) + CryptGetHashDigestSize(outerHash));

        // 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;
    }
    return result;
```

IECNORM.COM Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9 Subsystem

9.1 CommandAudit.c

9.1.1 Introduction

This file contains the functions that support command audit.

9.1.2 Includes

```
#include "InternalRoutines.h"
```

9.1.3 Functions

9.1.3.1 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.

```
void
CommandAuditPreInstall_Init(
    void
    )
{
    // Clear all the audit commands
    MemorySet(gp.auditComands, 0x00,
              ((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8);

    // TPM_CC_SetCommandCodeAuditStatus always being audited
    if(CommandIsImplemented(TPM_CC_SetCommandCodeAuditStatus))
        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
    NvWriteReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
    NvWriteReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
    NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);

    return;
}
```

9.1.3.2 CommandAuditStartup()

This function clears the command audit digest on a TPM Reset.

```
void
CommandAuditStartup(
    STARTUP_TYPE     type           // IN: start up type
    )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
{
    if(type == SU_RESET)
    {
        // Reset the digest size to initialize the digest
        gr.commandAuditDigest.t.size = 0;
    }

}
```

9.1.3.3 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 29

Return Value	Meaning
TRUE	the command code audit status was changed
FALSE	the command code audit status was not changed

```
BOOL
CommandAuditSet(
    TPM_CC           commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // Only SET a bit if the corresponding command is implemented
    if(CommandIsImplemented(commandCode))
    {
        // Can't audit shutdown
        if(commandCode != TPM_CC_Shutdown)
        {
            bitPos = commandCode - TPM_CC_FIRST;
            if(!BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
            {
                // Set bit
                BitSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
                return TRUE;
            }
        }
    }
    // No change
    return FALSE;
}
```

9.1.3.4 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

Table 30

Return Value	Meaning
TRUE	the command code audit status was changed
FALSE	the command code audit status was not changed

```
BOOL
CommandAuditClear(
    TPM_CC           commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // Do nothing if the command is not implemented
    if(CommandIsImplemented(commandCode))
    {
        // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
        // cleared
        if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
        {
            bitPos = commandCode - TPM_CC_FIRST;
            if(BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
            {
                // Clear bit
                BitClear(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
                return TRUE;
            }
        }
    }
    // No change
    return FALSE;
}
```

9.1.3.5 CommandAuditIsRequired()

This function indicates if the audit flag is SET for a command.

Table 31

Return Value	Meaning
TRUE	if command is audited
FALSE	if command is not audited

```
BOOL
CommandAuditIsRequired(
    TPM_CC           commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    bitPos = commandCode - TPM_CC_FIRST;

    // Check the bit map.  If the bit is SET, command audit is required
    if((gp.auditComands[bitPos/8] & (1 << (bitPos % 8))) != 0)
        return TRUE;
    else
        return FALSE;

}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.1.3.6 CommandAuditCapGetCCList()

This function returns a list of commands that have their audit bit SET.

The list starts at the input *commandCode*.

Table 32

Return Value	Meaning
YES	if there are more command code available
NO	all the available command code has been returned

```
108  TPMI_YES_NO
109  CommandAuditCapGetCCList(
110      TPM_CC            commandCode,   // IN: start command code
111      UINT32            count,         // IN: count of returned TPM_CC
112      TPML_CC          *commandList    // OUT: list of TPM_CC
113      )
114  {
115      TPMI_YES_NO      more = NO;
116      UINT32           i;
117
118      // Initialize output handle list
119      commandList->count = 0;
120
121      // The maximum count of command we may return is MAX_CAP_CC
122      if(count > MAX_CAP_CC) count = MAX_CAP_CC;
123
124      // If the command code is smaller than TPM_CC_FIRST, start from TPM_CC_FIRST
125      if(commandCode < TPM_CC_FIRST) commandCode = TPM_CC_FIRST;
126
127      // Collect audit commands
128      for(i = commandCode; i <= TPM_CC_LAST; i++)
129      {
130          if(CommandAuditIsRequired(i))
131          {
132              if(commandList->count < count)
133              {
134                  // If we have not filled up the return list, add this command
135                  // code to it
136                  commandList->commandCodes[commandList->count] = i;
137                  commandList->count++;
138              }
139              else
140              {
141                  // If the return list is full but we still have command
142                  // available, report this and stop iterating
143                  more = YES;
144                  break;
145              }
146          }
147      }
148
149      return more;
150
151  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.1.3.7 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.

```
152 void
153 CommandAuditGetDigest(
154     TPM2B_DIGEST    *digest         // OUT: command digest
155     )
156 {
157     TPM_CC                      i;
158     HASH_STATE                  hashState;
159
160     // Start hash
161     digest->t.size = CryptStartHash(gp.auditHashAlg, &hashState);
162
163     // Add command code
164     for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
165     {
166         if(CommandAuditIsRequired(i))
167         {
168             CryptUpdateDigestInt(&hashState, sizeof(i), &i);
169         }
170     }
171
172     // Complete hash
173     CryptCompleteHash2B(&hashState, &digest->b);
174
175     return;
176 }
```

9.2 DA.c

9.2.1 Introduction

This file contains the functions and data definitions relating to the dictionary attack logic.

9.2.2 Includes and Data Definitions

```
1 #define DA_C
2 #include "InternalRoutines.h"
```

9.2.3 Functions

9.2.3.1 DAPreInstall_Init()

This function initializes the DA parameters to their manufacturer-default values. The default values are determined by a platform-specific specification.

This function should not be called outside of a manufacturing or simulation environment.

The DA parameters will be restored to these initial values by TPM2_Clear().

```
3 void
4 DAPreInstall_Init(
5     void
6     )
7 {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
    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
    NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
    NvWriteReserved(NV_MAX_TRIES, &gp.maxTries);
    NvWriteReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
    NvWriteReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
    NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);

    return;
}
```

9.2.3.2 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.

```
void
DAStartup(
    STARTUP_TYPE     type           // IN: startup type
    )
{
    // For TPM Reset, if lockoutRecovery is 0, enable use of lockoutAuth.
    if(type == SU_RESET)
    {
        if(gp.lockoutRecovery == 0)
        {
            gp.lockOutAuthEnabled = TRUE;
            // Record the changes to NV
            NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
        }
    }

    // If DA has not been disabled and the previous shutdown is not orderly
    // failedTries is not already at its maximum then increment 'failedTries'
    if(    gp.recoveryTime != 0
        && g_prevOrderlyState == SHUTDOWN_NONE
        && gp.failedTries < gp.maxTries)
    {
        gp.failedTries++;
        // Record the change to NV
        NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
    }

    // Reset self healing timers
    s_selfHealTimer = g_time;
    s_lockoutTimer = g_time;

    return;
}
```

9.2.3.3 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
56  void
57  DARegisterFailure(
58      TPM_HANDLE       handle         // IN: handle for failure
59      )
60  {
61      // Reset the timer associated with lockout if the handle is the lockout auth.
62      if(handle == TPM_RH_LOCKOUT)
63          s_lockoutTimer = g_time;
64      else
65          s_selfHealTimer = g_time;
66
67      return;
68  }
```

9.2.3.4 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.

```
69  void
70  DASelfHeal(
71      void
72      )
73  {
74      // Regular auth self healing logic
75      // If no failed authorization tries, do nothing.  Otherwise, try to
76      // decrease failedTries
77      if(gp.failedTries != 0)
78      {
79          // if recovery time is 0, DA logic has been disabled.  Clear failed tries
80          // immediately
81          if(gp.recoveryTime == 0)
82          {
83              gp.failedTries = 0;
84              // Update NV record
85              NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
86          }
87          else
88          {
89              UINT64      decreaseCount;
90
91              // In the unlikely event that failedTries should become larger than
92              // maxTries
93              if(gp.failedTries > gp.maxTries)
94                  gp.failedTries = gp.maxTries;
95
96              // How much can failedTried be decreased
97              decreaseCount = ((g_time - s_selfHealTimer) / 1000) / gp.recoveryTime;
98
99              if(gp.failedTries <= (UINT32) decreaseCount)
100                 // should not set failedTries below zero
101                 gp.failedTries = 0;
102             else
103                 gp.failedTries -= (UINT32) decreaseCount;
104
105             // the cast prevents overflow of the product
106             s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
107             if(decreaseCount != 0)
108                 // If there was a change to the failedTries, record the changes
109                 // to NV
110                 NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
111         }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
    }


    // 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 - s_lockoutTimer)/1000) >= gp.lockoutRecovery)
            {
                gp.lockOutAuthEnabled = TRUE;
                // Record the changes to NV
                NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
            }
        }
    }

    return;
}
```

9.3 Hierarchy.c

9.3.1 Introduction

This file contains the functions used for managing and accessing the hierarchy-related values.

9.3.2 Includes

```
#include "InternalRoutines.h"
```

9.3.3 Functions

9.3.3.1 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.

```
void
HierarchyPreInstall_Init(
    void
    )
{
    // Allow lockout clear command
    gp.disableClear = FALSE;

    // Initialize Primary Seeds
    gp.EPSeed.t.size = PRIMARY_SEED_SIZE;
    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.EPSeed.t.buffer);
    gp.SPSeed.t.size = PRIMARY_SEED_SIZE;
    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.SPSeed.t.buffer);
    gp.PPSeed.t.size = PRIMARY_SEED_SIZE;
    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.PPSeed.t.buffer);

    // Initialize owner, endorsement and lockout auth
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
19      gp.ownerAuth.t.size = 0;
20      gp.endorsementAuth.t.size = 0;
21      gp.lockoutAuth.t.size = 0;
22
23      // Initialize owner, endorsement, and lockout policy
24      gp.ownerAlg = TPM_ALG_NULL;
25      gp.ownerPolicy.t.size = 0;
26      gp.endorsementAlg = TPM_ALG_NULL;
27      gp.endorsementPolicy.t.size = 0;
28      gp.lockoutAlg = TPM_ALG_NULL;
29      gp.lockoutPolicy.t.size = 0;
30
31      // Initialize ehProof, shProof and phProof
32      gp.phProof.t.size = PROOF_SIZE;
33      gp.shProof.t.size = PROOF_SIZE;
34      gp.ehProof.t.size = PROOF_SIZE;
35      CryptGenerateRandom(gp.phProof.t.size, gp.phProof.t.buffer);
36      CryptGenerateRandom(gp.shProof.t.size, gp.shProof.t.buffer);
37      CryptGenerateRandom(gp.ehProof.t.size, gp.ehProof.t.buffer);
38
39      // Write hierarchy data to NV
40      NvWriteReserved(NV_DISABLE_CLEAR, &gp.disableClear);
41      NvWriteReserved(NV_EP_SEED, &gp.EPSeed);
42      NvWriteReserved(NV_SP_SEED, &gp.SPSeed);
43      NvWriteReserved(NV_PP_SEED, &gp.PPSeed);
44      NvWriteReserved(NV_OWNER_AUTH, &gp.ownerAuth);
45      NvWriteReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
46      NvWriteReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
47      NvWriteReserved(NV_OWNER_ALG, &gp.ownerAlg);
48      NvWriteReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
49      NvWriteReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
50      NvWriteReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
51      NvWriteReserved(NV_LOCKOUT_ALG, &gp.lockoutAlg);
52      NvWriteReserved(NV_LOCKOUT_POLICY, &gp.lockoutPolicy);
53      NvWriteReserved(NV_PH_PROOF, &gp.phProof);
54      NvWriteReserved(NV_SH_PROOF, &gp.shProof);
55      NvWriteReserved(NV_EH_PROOF, &gp.ehProof);
56
57      return;
58  }
```

9.3.3.2 HierarchyStartup()

This function is called at TPM2_Startup() to initialize the hierarchy related values.

```
59  void
60  HierarchyStartup(
61      STARTUP_TYPE     type           // IN: start up type
62      )
63  {
64      // phEnable is SET on any startup
65      g_phEnable = TRUE;
66
67      // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
68      // TPM_RESTART
69      if(type != SU_RESUME)
70      {
71          gc.platformAuth.t.size = 0;
72          gc.platformPolicy.t.size = 0;
73
74          // enable the storage and endorsement hierarchies and the platformNV
75          gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
76      }
77
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
78      // nullProof and nullSeed are updated at every TPM_RESET
79      if(type == SU_RESET)
80      {
81          gr.nullProof.t.size = PROOF_SIZE;
82          CryptGenerateRandom(gr.nullProof.t.size,
83                              gr.nullProof.t.buffer);
84          gr.nullSeed.t.size = PRIMARY_SEED_SIZE;
85          CryptGenerateRandom(PRIMARY_SEED_SIZE, gr.nullSeed.t.buffer);
86      }
87
88      return;
89  }
```

9.3.3.3 HierarchyGetProof()

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

```
90  TPM2B_AUTH *
91  HierarchyGetProof(
92      TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy constant
93      )
94  {
95      TPM2B_AUTH          *auth = NULL;
96
97      switch(hierarchy)
98      {
99      case TPM_RH_PLATFORM:
100         // phProof for TPM_RH_PLATFORM
101         auth = &gp.phProof;
102         break;
103     case TPM_RH_ENDORSEMENT:
104         // ehProof for TPM_RH_ENDORSEMENT
105         auth = &gp.ehProof;
106         break;
107     case TPM_RH_OWNER:
108         // shProof for TPM_RH_OWNER
109         auth = &gp.shProof;
110         break;
111     case TPM_RH_NULL:
112         // nullProof for TPM_RH_NULL
113         auth = &gr.nullProof;
114         break;
115     default:
116         pAssert(FALSE);
117         break;
118     }
119     return auth;
120
121 }
```

9.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```
122 TPM2B_SEED *
123 HierarchyGetPrimarySeed(
124     TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy
125     )
126 {
127     TPM2B_SEED          *seed = NULL;
128     switch(hierarchy)
129     {
130     case TPM_RH_PLATFORM:
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
131             seed = &gp.PPSeed;
132             break;
133         case TPM_RH_OWNER:
134             seed = &gp.SPSeed;
135             break;
136         case TPM_RH_ENDORSEMENT:
137             seed = &gp.EPSeed;
138             break;
139         case TPM_RH_NULL:
140             return &gr.nullSeed;
141         default:
142             pAssert(FALSE);
143             break;
144         }
145         return seed;
146     }
```

9.3.3.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE The TPM_RH_NULL hierarchy is always enabled.

Table 33

Return Value	Meaning
TRUE	hierarchy is enabled
FALSE	hierarchy is disabled

```
147     BOOL
148     HierarchyIsEnabled(
149         TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy
150         )
151     {
152         BOOL            enabled = FALSE;
153
154         switch(hierarchy)
155         {
156         case TPM_RH_PLATFORM:
157             enabled = g_phEnable;
158             break;
159         case TPM_RH_OWNER:
160             enabled = gc.shEnable;
161             break;
162         case TPM_RH_ENDORSEMENT:
163             enabled = gc.ehEnable;
164             break;
165         case TPM_RH_NULL:
166             enabled = TRUE;
167             break;
168         default:
169             pAssert(FALSE);
170             break;
171         }
172         return enabled;
173     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.4 NV.c

9.4.1 Introduction

The NV memory is divided into two area: dynamic space for user defined NV Indices and evict objects, and reserved space for TPM persistent and state save data.

9.4.2 Includes, Defines and Data Definitions

```
#define NV_C
#include "InternalRoutines.h"
#include <Platform.h>
```

NV Index/evict object iterator value

```
typedef     UINT32          NV_ITER;        // type of a NV iterator
#define     NV_ITER_INIT    0xFFFFFFFF      // initial value to start an
                                            // iterator
```

9.4.3 NV Utility Functions

9.4.3.1 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 call to NvIsAvailable().

```
void
NvCheckState(void)
{
    int     func_return;

    func_return = _plat__IsNvAvailable();
    if(func_return == 0)
    {
        s_NvStatus = TPM_RC_SUCCESS;
    }
    else if(func_return == 1)
    {
        s_NvStatus = TPM_RC_NV_UNAVAILABLE;
    }
    else
    {
        s_NvStatus = TPM_RC_NV_RATE;
    }

    return;
}
```

9.4.3.2 NvIsAvailable()

This function returns the NV availability parameter.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

Table 34

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

```
TPM_RC
NvIsAvailable(
    void
    )
{
    return s_NvStatus;
}
```

9.4.3.3 NvCommit

This is a wrapper for the platform function to commit pending NV writes.

```
BOOL
NvCommit(
    void
    )
{
    BOOL    success = (_plat__NvCommit() == 0);
    return success;
}
```

9.4.3.4 NvReadMaxCount()

This function returns the max NV counter value.

```
static UINT64
NvReadMaxCount(
    void
    )
{
    UINT64      countValue;
    _plat__NvMemoryRead(s_maxCountAddr, sizeof(UINT64), &countValue);
    return countValue;
}
```

9.4.3.5 NvWriteMaxCount()

This function updates the max counter value to NV memory.

```
static void
NvWriteMaxCount(
    UINT64          maxCount
    )
{
    _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &maxCount);
    return;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.4.4 NV Index and Persistent Object Access Functions

9.4.4.1 Introduction

These functions are used to access an NV Index and persistent object memory. In this implementation, the memory is simulated with RAM. The data in dynamic area is organized as a linked list, starting from address *s_evictNvStart*. The first 4 bytes of a node in this link list is the offset of next node, followed by the data entry. A 0-valued offset value indicates the end of the list. If the data entry area of the last node happens to reach the end of the dynamic area without space left for an additional 4 byte end marker, the end address, *s_evictNvEnd*, should serve as the mark of list end.

9.4.4.2 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_ITER_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.

```
60  static UINT32
61  NvNext(
62      NV_ITER         *iter
63      )
64  {
65      NV_ITER     currentIter;
66
67      // If iterator is at the beginning of list
68      if(*iter == NV_ITER_INIT)
69      {
70          // Initialize iterator
71          *iter = s_evictNvStart;
72      }
73
74      // If iterator reaches the end of NV space, or iterator indicates list end
75      if(*iter + sizeof(UINT32) > s_evictNvEnd || *iter == 0)
76          return 0;
77
78      // Save the current iter offset
79      currentIter = *iter;
80
81      // Adjust iter pointer pointing to next entity
82      // Read pointer value
83      _plat__NvMemoryRead(*iter, sizeof(UINT32), iter);
84
85      if(*iter == 0) return 0;
86
87      return currentIter + sizeof(UINT32);      // entity stores after the pointer
88  }
```

9.4.4.3 NvGetEnd()

Function to find the end of the NV dynamic data list.

```
89  static UINT32
90  NvGetEnd(
91      void
92      )
93  {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
94      NV_ITER         iter = NV_ITER_INIT;
95      UINT32          endAddr = s_evictNvStart;
96      UINT32          currentAddr;
97
98      while((currentAddr = NvNext(&iter)) != 0)
99          endAddr = currentAddr;
100
101     if(endAddr != s_evictNvStart)
102     {
103         // Read offset
104         endAddr -= sizeof(UINT32);
105         _plat__NvMemoryRead(endAddr, sizeof(UINT32), &endAddr);
106     }
107
108     return endAddr;
109 }
```

9.4.4.4 NvGetFreeByte

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

```
110 static UINT32
111 NvGetFreeByte(
112     void
113     )
114 {
115     return s_evictNvEnd - NvGetEnd();
116 }
```

9.4.4.5 NvGetEvictObjectSize

This function returns the size of an evict object in NV space.

```
117 static UINT32
118 NvGetEvictObjectSize(
119     void
120     )
121 {
122     return sizeof(TPM_HANDLE) + sizeof(OBJECT) + sizeof(UINT32);
123 }
```

9.4.4.6 NvGetCounterSize

This function returns the size of a counter index in NV space.

```
124 static UINT32
125 NvGetCounterSize(
126     void
127     )
128 {
129     // It takes an offset field, a handle and the sizeof(NV_INDEX) and
130     // sizeof(UINT64) for counter data
131     return sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + sizeof(UINT64) + sizeof(UINT32);
132 }
```

9.4.4.7 NvTestSpace()

This function will test if there is enough space to add a new entity.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 35

Return Value	Meaning
TRUE	space available
FALSE	no enough space

```
133  static BOOL
134  NvTestSpace(
135      UINT32           size,          // IN: size of the entity to be added
136      BOOL             isIndex        // IN: TRUE if the entity is an index
137      )
138  {
139      UINT32      remainByte = NvGetFreeByte();
140
141      // For NV Index, need to make sure that we do not allocate and Index if this
142      // would mean that the TPM cannot allocate the minimum number of evict
143      // objects.
144      if(isIndex)
145      {
146          // Get the number of persistent objects allocated
147          UINT32      persistentNum = NvCapGetPersistentNumber();
148
149          // If we have not allocated the requisite number of evict objects, then we
150          // need to reserve space for them.
151          // NOTE: some of this is not written as simply as it might seem because
152          // the values are all unsigned and subtracting needs to be done carefully
153          // so that an underflow doesn't cause problems.
154          if(persistentNum < MIN_EVICT_OBJECTS)
155          {
156              UINT32      needed = (MIN_EVICT_OBJECTS - persistentNum)
157                                   * NvGetEvictObjectSize();
158              if(needed > remainByte)
159                  remainByte = 0;
160              else
161                  remainByte -= needed;
162          }
163          // if the requisite number of evict objects have been allocated then
164          // no need to reserve additional space
165      }
166      // This checks for the size of the value being added plus the index value.
167      // NOTE: This does not check to see if the end marker can be placed in
168      // memory because the end marker will not be written if it will not fit.
169      return (size + sizeof(UINT32) <= remainByte);
170  }
```

9.4.4.8 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).

```
171  static void
172  NvAdd(
173      UINT32           totalSize,     // IN: total size needed for this entity For
174                                      //     evict object, totalSize is the same as
175                                      //     bufferSize.  For NV Index, totalSize is
176                                      //     bufferSize plus index data size
177      UINT32           bufferSize,    // IN: size of initial buffer
178      BYTE            *entity         // IN: initial buffer
179      )
180  {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
181         UINT32          endAddr;
182         UINT32          nextAddr;
183         UINT32          listEnd = 0;
184
185         // Get the end of data list
186         endAddr = NvGetEnd();
187
188         // Calculate the value of next pointer, which is the size of a pointer +
189         // the entity data size
190         nextAddr = endAddr + sizeof(UINT32) + totalSize;
191
192         // Write next pointer
193         _plat__NvMemoryWrite(endAddr, sizeof(UINT32), &nextAddr);
194
195         // Write entity data
196         _plat__NvMemoryWrite(endAddr + sizeof(UINT32), bufferSize, entity);
197
198         // Write the end of list if it is not going to exceed the NV space
199         if(nextAddr + sizeof(UINT32) <= s_evictNvEnd)
200             _plat__NvMemoryWrite(nextAddr, sizeof(UINT32), &listEnd);
201
202         // Set the flag so that NV changes are committed before the command completes.
203         g_updateNV = TRUE;
204     }
```

9.4.4.9 NvDelete()

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

```
205     static void
206     NvDelete(
207         UINT32           entityAddr     // IN: address of entity to be deleted
208         )
209     {
210         UINT32          next;
211         UINT32          entrySize;
212         UINT32          entryAddr = entityAddr - sizeof(UINT32);
213         UINT32          listEnd = 0;
214
215         // Get the offset of the next entry.
216         _plat__NvMemoryRead(entryAddr, sizeof(UINT32), &next);
217
218         // The size of this entry is the difference between the current entry and the
219         // next entry.
220         entrySize = next - entryAddr;
221
222         // Move each entry after the current one to fill the freed space.
223         // Stop when we have reached the end of all the indexes. There are two
224         // ways to detect the end of the list. The first is to notice that there
225         // is no room for anything else because we are at the end of NV. The other
226         // indication is that we find an end marker.
227
228         // The loop condition checks for the end of NV.
229         while(next + sizeof(UINT32) <= s_evictNvEnd)
230         {
231             UINT32      size, oldAddr, newAddr;
232
233             // Now check for the end marker
234             _plat__NvMemoryRead(next, sizeof(UINT32), &oldAddr);
235             if(oldAddr == 0)
236                 break;
237
238             size = oldAddr - next;
239
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
240             // Move entry
241             _plat__NvMemoryMove(next, next - entrySize, size);
242
243             // Update forward link
244             newAddr = oldAddr - entrySize;
245             _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &newAddr);
246             next = oldAddr;
247         }
248         // Mark the end of list
249         _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &listEnd);
250
251         // Set the flag so that NV changes are committed before the command completes.
252         g_updateNV = TRUE;
253     }
```

9.4.5 RAM-based NV Index Data Access Functions

9.4.5.1 Introduction

The data layout in ram buffer is {size of(*NV_handle*() + data), NV_handle(), data} for each NV Index data stored in RAM.

NV storage is updated when a NV Index is added or deleted. We do NOT updated NV storage when the data is updated.

9.4.5.2 NvTestRAMSpace()

This function indicates if there is enough RAM space to add a data for a new NV Index.

Table 36

Return Value	Meaning
TRUE	space available
FALSE	no enough space

```
254     static BOOL
255     NvTestRAMSpace(
256         UINT32          size            // IN: size of the data to be added to RAM
257         )
258     {
259         BOOL        success = (   s_ramIndexSize
260                                 + size
261                                 + sizeof(TPM_HANDLE) + sizeof(UINT32)
262                                 <= RAM_INDEX_SPACE);
263         return success;
264     }
```

9.4.5.3 NvGetRamIndexOffset

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.

```
265     static UINT32
266     NvGetRAMIndexOffset(
267         TPMI_RH_NV_INDEX     handle         // IN: NV handle
268         )
269     {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
270         UINT32      currAddr = 0;
271
272         while(currAddr < s_ramIndexSize)
273         {
274             TPMI_RH_NV_INDEX    currHandle;
275             UINT32              currSize;
276             currHandle = * (TPM_HANDLE *) &s_ramIndex[currAddr + sizeof(UINT32)];
277
278             // Found a match
279             if(currHandle == handle)
280
281                 // data buffer follows the handle and size field
282                 break;
283
284             currSize = * (UINT32 *) &s_ramIndex[currAddr];
285             currAddr += sizeof(UINT32) + currSize;
286         }
287
288         // We assume the index data is existing in RAM space
289         pAssert(currAddr < s_ramIndexSize);
290         return currAddr + sizeof(TPMI_RH_NV_INDEX) + sizeof(UINT32);
291     }
```

9.4.5.4 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.

```
292     static void
293     NvAddRAM(
294         TPMI_RH_NV_INDEX     handle,         // IN: NV handle
295         UINT32               size            // IN: size of data
296         )
297     {
298         // Add data space at the end of reserved RAM buffer
299         * (UINT32 *) &s_ramIndex[s_ramIndexSize] = size + sizeof(TPMI_RH_NV_INDEX);
300         * (TPMI_RH_NV_INDEX *) &s_ramIndex[s_ramIndexSize + sizeof(UINT32)] = handle;
301         s_ramIndexSize += sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX) + size;
302
303         pAssert(s_ramIndexSize <= RAM_INDEX_SPACE);
304
305         // Update NV version of s_ramIndexSize
306         _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
307
308         // Write reserved RAM space to NV to reflect the newly added NV Index
309         _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
310
311         return;
312     }
```

9.4.5.5 NvDeleteRAM()

This function is used to delete a RAM-backed NV Index data area.

This function assumes the data of NV Index exists in RAM.

```
313     static void
314     NvDeleteRAM(
315         TPMI_RH_NV_INDEX     handle          // IN: NV handle
316         )
317     {
318         UINT32          nodeOffset;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
319         UINT32          nextNode;
320         UINT32          size;
321
322         nodeOffset = NvGetRAMIndexOffset(handle);
323
324         // Move the pointer back to get the size field of this node
325         nodeOffset -= sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX);
326
327         // Get node size
328         size = * (UINT32 *) &s_ramIndex[nodeOffset];
329
330         // Get the offset of next node
331         nextNode = nodeOffset + sizeof(UINT32) + size;
332
333         // Move data
334         MemoryMove(s_ramIndex + nodeOffset, s_ramIndex + nextNode,
335                    s_ramIndexSize - nextNode, s_ramIndexSize - nextNode);
336
337         // Update RAM size
338         s_ramIndexSize -= size + sizeof(UINT32);
339
340         // Update NV version of s_ramIndexSize
341         _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
342
343         // Write reserved RAM space to NV to reflect the newly delete NV Index
344         _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
345
346         return;
347     }
```

9.4.6 Utility Functions

9.4.6.1 NvInitStatic()

This function initializes the static variables used in the NV subsystem.

```
348     static void
349     NvInitStatic(
350         void
351         )
352     {
353         UINT16      i;
354         UINT32      reservedAddr;
355
356         s_reservedSize[NV_DISABLE_CLEAR] = sizeof(gp.disableClear);
357         s_reservedSize[NV_OWNER_ALG] = sizeof(gp.ownerAlg);
358         s_reservedSize[NV_ENDORSEMENT_ALG] = sizeof(gp.endorsementAlg);
359         s_reservedSize[NV_LOCKOUT_ALG] = sizeof(gp.lockoutAlg);
360         s_reservedSize[NV_OWNER_POLICY] = sizeof(gp.ownerPolicy);
361         s_reservedSize[NV_ENDORSEMENT_POLICY] = sizeof(gp.endorsementPolicy);
362         s_reservedSize[NV_LOCKOUT_POLICY] = sizeof(gp.lockoutPolicy);
363         s_reservedSize[NV_OWNER_AUTH] = sizeof(gp.ownerAuth);
364         s_reservedSize[NV_ENDORSEMENT_AUTH] = sizeof(gp.endorsementAuth);
365         s_reservedSize[NV_LOCKOUT_AUTH] = sizeof(gp.lockoutAuth);
366         s_reservedSize[NV_EP_SEED] = sizeof(gp.EPSeed);
367         s_reservedSize[NV_SP_SEED] = sizeof(gp.SPSeed);
368         s_reservedSize[NV_PP_SEED] = sizeof(gp.PPSeed);
369         s_reservedSize[NV_PH_PROOF] = sizeof(gp.phProof);
370         s_reservedSize[NV_SH_PROOF] = sizeof(gp.shProof);
371         s_reservedSize[NV_EH_PROOF] = sizeof(gp.ehProof);
372         s_reservedSize[NV_TOTAL_RESET_COUNT] = sizeof(gp.totalResetCount);
373         s_reservedSize[NV_RESET_COUNT] = sizeof(gp.resetCount);
374         s_reservedSize[NV_PCR_POLICIES] = sizeof(gp.pcrPolicies);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
375         s_reservedSize[NV_PCR_ALLOCATED] = sizeof(gp.pcrAllocated);
376         s_reservedSize[NV_PP_LIST] = sizeof(gp.ppList);
377         s_reservedSize[NV_FAILED_TRIES] = sizeof(gp.failedTries);
378         s_reservedSize[NV_MAX_TRIES] = sizeof(gp.maxTries);
379         s_reservedSize[NV_RECOVERY_TIME] = sizeof(gp.recoveryTime);
380         s_reservedSize[NV_LOCKOUT_RECOVERY] = sizeof(gp.lockoutRecovery);
381         s_reservedSize[NV_LOCKOUT_AUTH_ENABLED] = sizeof(gp.lockOutAuthEnabled);
382         s_reservedSize[NV_ORDERLY] = sizeof(gp.orderlyState);
383         s_reservedSize[NV_AUDIT_COMMANDS] = sizeof(gp.auditComands);
384         s_reservedSize[NV_AUDIT_HASH_ALG] = sizeof(gp.auditHashAlg);
385         s_reservedSize[NV_AUDIT_COUNTER] = sizeof(gp.auditCounter);
386         s_reservedSize[NV_ALGORITHM_SET] = sizeof(gp.algorithmSet);
387         s_reservedSize[NV_FIRMWARE_V1] = sizeof(gp.firmwareV1);
388         s_reservedSize[NV_FIRMWARE_V2] = sizeof(gp.firmwareV2);
389         s_reservedSize[NV_ORDERLY_DATA] = sizeof(go);
390         s_reservedSize[NV_STATE_CLEAR] = sizeof(gc);
391         s_reservedSize[NV_STATE_RESET] = sizeof(gr);
392
393         // Initialize reserved data address.  In this implementation, reserved data
394         // is stored at the start of NV memory
395         reservedAddr = 0;
396         for(i = 0; i < NV_RESERVE_LAST; i++)
397         {
398             s_reservedAddr[i] = reservedAddr;
399             reservedAddr += s_reservedSize[i];
400         }
401
402         // Initialize auxiliary variable space for index/evict implementation.
403         // Auxiliary variables are stored after reserved data area
404         // RAM index copy starts at the beginning
405         s_ramIndexSizeAddr = reservedAddr;
406         s_ramIndexAddr = s_ramIndexSizeAddr + sizeof(UINT32);
407
408         // Maximum counter value
409         s_maxCountAddr = s_ramIndexAddr + RAM_INDEX_SPACE;
410
411         // dynamic memory start
412         s_evictNvStart = s_maxCountAddr + sizeof(UINT64);
413
414         // dynamic memory ends at the end of NV memory
415         s_evictNvEnd = NV_MEMORY_SIZE;
416
417         return;
418     }
```

9.4.6.2 NvInit()

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.

```
419     void
420     NvInit(
421         void
422         )
423     {
424         UINT32      nullPointer = 0;
425         UINT64      zeroCounter = 0;
426
427         // Initialize static variables
428         NvInitStatic();
429
430         // Initialize RAM index space as unused
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
431         _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &nullPointer);
432
433         // Initialize max counter value to 0
434         _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &zeroCounter);
435
436         // Initialize the next offset of the first entry in evict/index list to 0
437         _plat__NvMemoryWrite(s_evictNvStart, sizeof(TPM_HANDLE), &nullPointer);
438
439         return;
440
441     }
```

9.4.6.3 NvReadReserved()

This function is used to move reserved data from NV memory to RAM.

```
442     void
443     NvReadReserved(
444         NV_RESERVE       type,          // IN: type of reserved data
445         void            *buffer         // OUT: buffer receives the data.
446         )
447     {
448         // Input type should be valid
449         pAssert(type >= 0 && type < NV_RESERVE_LAST);
450
451         _plat__NvMemoryRead(s_reservedAddr[type], s_reservedSize[type], buffer);
452         return;
453     }
```

9.4.6.4 NvWriteReserved()

This function is used to post a reserved data for writing to NV memory. Before the TPM completes the operation, the value will be written.

```
454     void
455     NvWriteReserved(
456         NV_RESERVE       type,          // IN: type of reserved data
457         void            *buffer         // IN: data buffer
458         )
459     {
460         // Input type should be valid
461         pAssert(type >= 0 && type < NV_RESERVE_LAST);
462
463         _plat__NvMemoryWrite(s_reservedAddr[type], s_reservedSize[type], buffer);
464
465         // Set the flag that a NV write happens
466         g_updateNV = TRUE;
467         return;
468     }
```

9.4.6.5 NvReadPersistent()

This function reads persistent data to the RAM copy of the *gp* structure.

```
469     void
470     NvReadPersistent(
471         void
472         )
473     {
474         // Hierarchy persistent data
475         NvReadReserved(NV_DISABLE_CLEAR, &gp.disableClear);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
476         NvReadReserved(NV_OWNER_ALG, &gp.ownerAlg);
477         NvReadReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
478         NvReadReserved(NV_LOCKOUT_ALG, &gp.lockoutAlg);
479         NvReadReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
480         NvReadReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
481         NvReadReserved(NV_LOCKOUT_POLICY, &gp.lockoutPolicy);
482         NvReadReserved(NV_OWNER_AUTH, &gp.ownerAuth);
483         NvReadReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
484         NvReadReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
485         NvReadReserved(NV_EP_SEED, &gp.EPSeed);
486         NvReadReserved(NV_SP_SEED, &gp.SPSeed);
487         NvReadReserved(NV_PP_SEED, &gp.PPSeed);
488         NvReadReserved(NV_PH_PROOF, &gp.phProof);
489         NvReadReserved(NV_SH_PROOF, &gp.shProof);
490         NvReadReserved(NV_EH_PROOF, &gp.ehProof);
491
492         // Time persistent data
493         NvReadReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
494         NvReadReserved(NV_RESET_COUNT, &gp.resetCount);
495
496         // PCR persistent data
497         NvReadReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
498         NvReadReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
499
500         // Physical Presence persistent data
501         NvReadReserved(NV_PP_LIST, &gp.ppList);
502
503         // Dictionary attack values persistent data
504         NvReadReserved(NV_FAILED_TRIES, &gp.failedTries);
505         NvReadReserved(NV_MAX_TRIES, &gp.maxTries);
506         NvReadReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
507         NvReadReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
508         NvReadReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
509
510         // Orderly State persistent data
511         NvReadReserved(NV_ORDERLY, &gp.orderlyState);
512
513         // Command audit values persistent data
514         NvReadReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
515         NvReadReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
516         NvReadReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
517
518         // Algorithm selection persistent data
519         NvReadReserved(NV_ALGORITHM_SET, &gp.algorithmSet);
520
521         // Firmware version persistent data
522         NvReadReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
523         NvReadReserved(NV_FIRMWARE_V2, &gp.firmwareV2);
524
525         return;
526     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

9.4.6.6 NvIsPlatformPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the platform.

© ISO/IEC 2015 – All rights reserved

Table 37

Return Value	Meaning
TRUE	handle references a platform persistent object
FALSE	handle does not reference platform persistent object and may reference an owner persistent object either

```
527  BOOL
528  NvIsPlatformPersistentHandle(
529      TPM_HANDLE       handle          // IN: handle
530      )
531  {
532      return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
533  }
```

9.4.6.7 NvIsOwnerPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the owner.

Table 38

Return Value	Meaning
TRUE	handle is owner persistent handle
FALSE	handle is not owner persistent handle and may not be a persistent handle at all

```
534  BOOL
535  NvIsOwnerPersistentHandle(
536      TPM_HANDLE       handle          // IN: handle
537      )
538  {
539      return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
540  }
```

9.4.6.8 NvNextIndex()

This function returns the offset in NV of the next NV Index entry. A value of 0 indicates the end of the list.

```
541  static UINT32
542  NvNextIndex(
543      NV_ITER          *iter
544      )
545  {
546      UINT32      addr;
547      TPM_HANDLE  handle;
548
549      while((addr = NvNext(iter)) != 0)
550      {
551          // Read handle
552          _plat__NvMemoryRead(addr, sizeof(TPM_HANDLE), &handle);
553          if(HandleGetType(handle) == TPM_HT_NV_INDEX)
554              return addr;
555      }
556
557      pAssert(addr == 0);
558      return addr;
559  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

9.4.6.9 NvNextEvict()

This function returns the offset in NV of the next evict object entry. A value of 0 indicates the end of the list.

```
560  static UINT32
561  NvNextEvict(
562      NV_ITER         *iter
563      )
564  {
565      UINT32      addr;
566      TPM_HANDLE  handle;
567
568      while((addr = NvNext(iter)) != 0)
569      {
570          // Read handle
571          _plat__NvMemoryRead(addr, sizeof(TPM_HANDLE), &handle);
572          if(HandleGetType(handle) == TPM_HT_PERSISTENT)
573              return addr;
574      }
575
576      pAssert(addr == 0);
577      return addr;
578  }
```

9.4.6.10 NvFindHandle()

this function returns the offset in NV memory of the entity associated with the input handle. A value of zero indicates that handle does not exist reference an existing persistent object or defined NV Index.

```
579  static UINT32
580  NvFindHandle(
581      TPM_HANDLE       handle
582      )
583  {
584      UINT32          addr;
585      NV_ITER         iter = NV_ITER_INIT;
586
587      while((addr = NvNext(&iter)) != 0)
588      {
589          TPM_HANDLE          entityHandle;
590          // Read handle
591          _plat__NvMemoryRead(addr, sizeof(TPM_HANDLE), &entityHandle);
592          if(entityHandle == handle)
593              return addr;
594      }
595
596      pAssert(addr == 0);
597      return addr;
598  }
```

9.4.6.11 NvPowerOn()

This function is called at _TPM_Init() to initialize the NV environment.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 39

Return Value	Meaning
TRUE	all NV was initialized
FALSE	the NV containing saved state had an error and TPM2_Startup(CLEAR) is required

```
599  BOOL
600  NvPowerOn(
601      void
602      )
603  {
604      int          nvError = 0;
605      // If power was lost, need to re-establish the RAM data that is loaded from
606      // NV and initialize the static variables
607      if(_plat__WasPowerLost(TRUE))
608      {
609          if((nvError = _plat__NVEnable(0)) < 0)
610              FAIL(FATAL_ERROR_NV_UNRECOVERABLE);
611
612          NvInitStatic();
613      }
614
615      return nvError == 0;
616  }
```

9.4.6.12 NvStateSave()

This function is used to cause the memory containing the RAM backed NV Indices to be written to NV.

```
617  void
618  NvStateSave(
619      void
620      )
621  {
622      // Write RAM backed NV Index info to NV
623      // No need to save s_ramIndexSize because we save it to NV whenever it is
624      // updated.
625      _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
626
627      // Set the flag so that an NV write happens before the command completes.
628      g_updateNV = TRUE;
629
630      return;
631  }
```

9.4.6.13 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.

```
632  void
633  NvEntityStartup(
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
634         STARTUP_TYPE     type           // IN: start up type
635         )
636     {
637         NV_ITER             iter = NV_ITER_INIT;
638         UINT32              currentAddr;        // offset points to the current entity
639
640         // Restore RAM index data
641         _plat__NvMemoryRead(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
642         _plat__NvMemoryRead(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
643
644         // If recovering from state save, do nothing
645         if(type == SU_RESUME)
646             return;
647
648         // Iterate all the NV Index to clear the locks
649         while((currentAddr = NvNextIndex(&iter)) != 0)
650         {
651             NV_INDEX    nvIndex;
652             UINT32      indexAddr;              // NV address points to index info
653             TPMA_NV     attributes;
654
655             indexAddr = currentAddr + sizeof(TPM_HANDLE);
656
657             // Read NV Index info structure
658             _plat__NvMemoryRead(indexAddr, sizeof(NV_INDEX), &nvIndex);
659             attributes = nvIndex.publicArea.attributes;
660
661             // Clear read/write lock
662             if(attributes.TPMA_NV_READLOCKED == SET)
663                 attributes.TPMA_NV_READLOCKED = CLEAR;
664
665             if(     attributes.TPMA_NV_WRITELOCKED == SET
666                 &&  (   attributes.TPMA_NV_WRITTEN == CLEAR
667                     ||  attributes.TPMA_NV_WRITEDEFINE == CLEAR
668                     )
669                )
670                 attributes.TPMA_NV_WRITELOCKED = CLEAR;
671
672             // Reset NV data for TPMA_NV_CLEAR_STCLEAR
673             if(attributes.TPMA_NV_CLEAR_STCLEAR == SET)
674             {
675                 attributes.TPMA_NV_WRITTEN = CLEAR;
676                 attributes.TPMA_NV_WRITELOCKED = CLEAR;
677             }
678
679             // Reset NV data for orderly values that are not counters
680             // NOTE: The function has already exited on a TPM Resume, so the only
681             // things being processed are TPM Restart and TPM Reset
682             if(     type == SU_RESET
683                 &&  attributes.TPMA_NV_ORDERLY == SET
684                 &&  attributes.TPMA_NV_COUNTER == CLEAR
685                )
686                 attributes.TPMA_NV_WRITTEN = CLEAR;
687
688             // Write NV Index info back if it has changed
689             if(*((UINT32 *)&attributes) != *((UINT32 *)&nvIndex.publicArea.attributes))
690             {
691                 nvIndex.publicArea.attributes = attributes;
692                 _plat__NvMemoryWrite(indexAddr, sizeof(NV_INDEX), &nvIndex);
693
694                 // Set the flag that a NV write happens
695                 g_updateNV = TRUE;
696             }
697             // Set the lower bits in an orderly counter to 1 for a non-orderly startup
698             if(   g_prevOrderlyState == SHUTDOWN_NONE
699                && attributes.TPMA_NV_WRITTEN == SET)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
700             {
701                 if(   attributes.TPMA_NV_ORDERLY == SET
702                    && attributes.TPMA_NV_COUNTER == SET)
703                 {
704                     TPMI_RH_NV_INDEX    nvHandle;
705                     UINT64              counter;
706 
707                     // Read NV handle
708                     _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);
709 
710                     // Read the counter value saved to NV upon the last roll over.
711                     // Do not use RAM backed storage for this once.
712                     nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = CLEAR;
713                     NvGetIntIndexData(nvHandle, &nvIndex, &counter);
714                     nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = SET;
715 
716                     // Set the lower bits of counter to 1's
717                     counter |= MAX_ORDERLY_COUNT;
718 
719                     // Write back to RAM
720                     NvWriteIndexData(nvHandle, &nvIndex, 0, sizeof(counter), &counter);
721 
722                     // No write to NV because an orderly shutdown will update the
723                     // counters.
724 
725                 }
726             }
727         }
728 
729         return;
730 
731     }
```

9.4.7 NV Access Functions

9.4.7.1 Introduction

This set of functions provide accessing NV Index and persistent objects based using a handle for reference to the entity.

9.4.7.2 NvIsUndefinedIndex()

This function is used to verify that an NV Index is not defined. This is only used by TPM2_NV_DefineSpace().

Table 40

Return Value	Meaning
TRUE	the handle points to an existing NV Index
FALSE	the handle points to a non-existent Index

```
732     BOOL
733     NvIsUndefinedIndex(
734         TPMI_RH_NV_INDEX     handle         // IN: handle
735         )
736     {
737         UINT32          entityAddr;         // offset points to the entity
738 
739         pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
740
741         // Find the address of index
742         entityAddr = NvFindHandle(handle);
743
744         // If handle is not found, return TPM_RC_SUCCESS
745         if(entityAddr == 0)
746             return TPM_RC_SUCCESS;
747
748         // NV Index is defined
749         return TPM_RC_NV_DEFINED;
750     }
```

9.4.7.3 NvIndexIsAccessible()

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

Table 41

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 and index created using platform auth
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

```
751     TPM_RC
752     NvIndexIsAccessible(
753         TPMI_RH_NV_INDEX     handle,         // IN: handle
754         TPM_CC               commandCode     // IN: the command
755         )
756     {
757         UINT32              entityAddr;     // offset points to the entity
758         NV_INDEX            nvIndex;        //
759
760         pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
761
762         // Find the address of index
763         entityAddr = NvFindHandle(handle);
764
765         // If handle is not found, return TPM_RC_HANDLE
766         if(entityAddr == 0)
767             return TPM_RC_HANDLE;
768
769         // Read NV Index info structure
770         _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
771                             &nvIndex);
772
773         if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
774         {
775             // if shEnable is CLEAR, an ownerCreate NV Index should not be
776             // indicated as present
777             if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
778             {
779                 if(gc.shEnable == FALSE)
780                     return TPM_RC_HANDLE;
781             }
782             // if phEnableNV is CLEAR, a platform created Index should not
783             // be visible
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
784             else if(gc.phEnableNV == FALSE)
785                 return TPM_RC_HANDLE;
786         }
787
788         // If the Index is write locked and this is an NV Write operation...
789         if(     nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED
790             &&  IsWriteOperation(commandCode))
791         {
792             // then return a locked indication unless the command is TPM2_NV_WriteLock
793             if(commandCode != TPM_CC_NV_WriteLock)
794                 return TPM_RC_NV_LOCKED;
795             return TPM_RC_SUCCESS;
796         }
797         // If the Index is read locked and this is an NV Read operation...
798         if(     nvIndex.publicArea.attributes.TPMA_NV_READLOCKED
799             && IsReadOperation(commandCode))
800         {
801             // then return a locked indication unless the command is TPM2_NV_ReadLock
802             if(commandCode != TPM_CC_NV_ReadLock)
803                 return TPM_RC_NV_LOCKED;
804             return TPM_RC_SUCCESS;
805         }
806
807         // NV Index is accessible
808         return TPM_RC_SUCCESS;
809     }
```

9.4.7.4 NvIsUndefinedEvictHandle()

This function indicates if a handle does not reference an existing persistent object. This function requires that the handle be in the proper range for persistent objects.

Table 42

Return Value	Meaning
TRUE	handle does not reference an existing persistent object
FALSE	handle does reference an existing persistent object

```
810     static BOOL
811     NvIsUndefinedEvictHandle(
812         TPM_HANDLE      handle          // IN: handle
813         )
814     {
815         UINT32          entityAddr;     // offset points to the entity
816         pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
817
818         // Find the address of evict object
819         entityAddr = NvFindHandle(handle);
820
821         // If handle is not found, return TRUE
822         if(entityAddr == 0)
823             return TRUE;
824         else
825             return FALSE;
826     }
```

9.4.7.5 NvGetEvictObject()

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 43

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

```
827  TPM_RC
828  NvGetEvictObject(
829      TPM_HANDLE       handle,        // IN: handle
830      OBJECT          *object         // OUT: object data
831      )
832  {
833      UINT32          entityAddr;         // offset points to the entity
834      TPM_RC          result = TPM_RC_SUCCESS;
835
836      pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
837
838      // Find the address of evict object
839      entityAddr = NvFindHandle(handle);
840
841      // If handle is not found, return an error
842      if(entityAddr == 0)
843          result = TPM_RC_HANDLE;
844      else
845          // Read evict object
846          _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE),
847                              sizeof(OBJECT),
848                              object);
849
850      // whether there is an error or not, make sure that the evict
851      // status of the object is set so that the slot will get freed on exit
852      object->attributes.evict = SET;
853
854      return result;
855  }
```

9.4.7.6 NvGetIndexInfo()

This function is used to retrieve the contents of an NV Index.

An implementation is allowed to save the NV Index in a vendor-defined format. If the format is different from the default used by the reference code, then this function would be changed to reformat the data into the default format.

A prerequisite to calling this function is that the handle must be known to reference a defined NV Index.

```
856  void
857  NvGetIndexInfo(
858      TPMI_RH_NV_INDEX     handle,        // IN: handle
859      NV_INDEX            *nvIndex        // OUT: NV index structure
860      )
861  {
862      UINT32               entityAddr;    // offset points to the entity
863
864      pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
865
866      // Find the address of NV index
867      entityAddr = NvFindHandle(handle);
868      pAssert(entityAddr != 0);
869
870      // This implementation uses the default format so just
871      // read the data in
872      _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
873                          nvIndex);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
874
875         return;
876     }
```

9.4.7.7 NvInitialCounter()

This function returns the value to be used when a counter index is initialized. It will scan the NV counters and find the highest value in any active counter. It will use that value as the starting point. If there are no active counters, it will use the value of the previous largest counter.

```
877     UINT64
878     NvInitialCounter(
879         void
880         )
881     {
882         UINT64          maxCount;
883         NV_ITER         iter = NV_ITER_INIT;
884         UINT32          currentAddr;
885
886         // Read the maxCount value
887         maxCount = NvReadMaxCount();
888
889         // Iterate all existing counters
890         while((currentAddr = NvNextIndex(&iter)) != 0)
891         {
892             TPMI_RH_NV_INDEX    nvHandle;
893             NV_INDEX            nvIndex;
894
895             // Read NV handle
896             _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);
897
898             // Get NV Index
899             NvGetIndexInfo(nvHandle, &nvIndex);
900             if(    nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
901                 && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
902             {
903                 UINT64      countValue;
904                 // Read counter value
905                 NvGetIntIndexData(nvHandle, &nvIndex, &countValue);
906                 if(countValue > maxCount)
907                     maxCount = countValue;
908             }
909         }
910         // Initialize the new counter value to be maxCount + 1
911         // A counter is only initialized the first time it is written. The
912         // way to write a counter is with TPM2_NV_INCREMENT(). Since the
913         // "initial" value of a defined counter is the largest count value that
914         // may have existed in this index previously, then the first use would
915         // add one to that value.
916         return maxCount;
917     }
```

9.4.7.8 NvGetIndexData()

This function is used to access the data in an NV Index. The data is returned as a byte sequence. Since counter values are kept in native format, they are converted to canonical form before being returned.

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.

```
918     void
919     NvGetIndexData(
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
920         TPMI_RH_NV_INDEX     handle,         // IN: handle
921         NV_INDEX            *nvIndex,        // IN: RAM image of index header
922         UINT32               offset,         // IN: offset of NV data
923         UINT16               size,           // IN: size of NV data
924         void                *data            // OUT: data buffer
925         )
926     {
927
928         pAssert(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET);
929
930         if(   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
931            || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET)
932         {
933             // Read bit or counter data in canonical form
934             UINT64      dataInInt;
935             NvGetIntIndexData(handle, nvIndex, &dataInInt);
936             UINT64_TO_BYTE_ARRAY(dataInInt, (BYTE *)data);
937         }
938         else
939         {
940             if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
941             {
942                 UINT32      ramAddr;
943
944                 // Get data from RAM buffer
945                 ramAddr = NvGetRAMIndexOffset(handle);
946                 MemoryCopy(data, s_ramIndex + ramAddr + offset, size, size);
947             }
948             else
949             {
950                 UINT32      entityAddr;
951                 entityAddr = NvFindHandle(handle);
952                 // Get data from NV
953                 // Skip NV Index info, read data buffer
954                 entityAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
955                 // Read the data
956                 _plat__NvMemoryRead(entityAddr, size, data);
957             }
958         }
959         return;
960     }
```

9.4.7.9 NvGetIntIndexData()

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.

```
961     void
962     NvGetIntIndexData(
963         TPMI_RH_NV_INDEX     handle,         // IN: handle
964         NV_INDEX            *nvIndex,        // IN: RAM image of NV Index header
965         UINT64              *data            // IN: UINT64 pointer for counter or bits
966         )
967     {
968         // Validate that index has been written and is the right type
969         pAssert(   nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET
970                 && (   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
971                     || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET
972                    )
973                );
974
975         // bit and counter value is store in native format for TPM CPU.  So we directly
976         // copy the contents of NV to output data buffer
977         if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
978         {
979             UINT32      ramAddr;
980 
981             // Get data from RAM buffer
982             ramAddr = NvGetRAMIndexOffset(handle);
983             MemoryCopy(data, s_ramIndex + ramAddr, sizeof(*data), sizeof(*data));
984         }
985         else
986         {
987             UINT32      entityAddr;
988             entityAddr = NvFindHandle(handle);
989 
990             // Get data from NV
991             // Skip NV Index info, read data buffer
992             _plat__NvMemoryRead(
993                 entityAddr + sizeof(TPM_HANDLE) + sizeof(NV_INDEX),
994                 sizeof(UINT64), data);
995         }
996 
997         return;
998     }
```

9.4.7.10 NvWriteIndexInfo()

This function is called to queue the write of NV Index data to persistent memory.

This function requires that NV Index is defined.

Table 44

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

```
999     TPM_RC
1000    NvWriteIndexInfo(
1001        TPMI_RH_NV_INDEX     handle,         // IN: handle
1002        NV_INDEX            *nvIndex         // IN: NV Index info to be written
1003        )
1004    {
1005        UINT32      entryAddr;
1006        TPM_RC      result;
1007 
1008        // Get the starting offset for the index in the RAM image of NV
1009        entryAddr = NvFindHandle(handle);
1010        pAssert(entryAddr != 0);
1011 
1012        // Step over the link value
1013        entryAddr = entryAddr + sizeof(TPM_HANDLE);
1014 
1015        // If the index data is actually changed, then a write to NV is required
1016        if(_plat__NvIsDifferent(entryAddr, sizeof(NV_INDEX),nvIndex))
1017        {
1018            // Make sure that NV is available
1019            result = NvIsAvailable();
1020            if(result != TPM_RC_SUCCESS)
1021                return result;
1022            _plat__NvMemoryWrite(entryAddr, sizeof(NV_INDEX), nvIndex);
1023            g_updateNV = TRUE;
1024        }
1025        return TPM_RC_SUCCESS;
1026    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.4.7.11 NvWriteIndexData()

This function is used to write NV index data.

This function requires that the NV Index is defined, and the data is within the defined data range for the index.

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

```
1027    TPM_RC
1028    NvWriteIndexData(
1029        TPMI_RH_NV_INDEX     handle,         // IN: handle
1030        NV_INDEX            *nvIndex,        // IN: RAM copy of NV Index
1031        UINT32               offset,         // IN: offset of NV data
1032        UINT32               size,           // IN: size of NV data
1033        void                *data            // OUT: data buffer
1034        )
1035    {
1036        TPM_RC               result;
1037        // Validate that write falls within range of the index
1038        pAssert(nvIndex->publicArea.dataSize >= offset + size);
1039
1040        // Update TPMA_NV_WRITTEN bit if necessary
1041        if(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
1042        {
1043            nvIndex->publicArea.attributes.TPMA_NV_WRITTEN = SET;
1044            result = NvWriteIndexInfo(handle, nvIndex);
1045            if(result != TPM_RC_SUCCESS)
1046                return result;
1047        }
1048
1049        // Check to see if process for an orderly index is required.
1050        if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
1051        {
1052            UINT32      ramAddr;
1053
1054            // Write data to RAM buffer
1055            ramAddr = NvGetRAMIndexOffset(handle);
1056            MemoryCopy(s_ramIndex + ramAddr + offset, data, size,
1057                       sizeof(s_ramIndex) - ramAddr - offset);
1058
1059            // NV update does not happen for orderly index.  Have
1060            // to clear orderlyState to reflect that we have changed the
1061            // NV and an orderly shutdown is required. Only going to do this if we
1062            // are not processing a counter that has just rolled over
1063            if(g_updateNV == FALSE)
1064                g_clearOrderly = TRUE;
1065        }
1066        // Need to process this part if the Index isn't orderly or if it is
1067        // an orderly counter that just rolled over.
1068        if(g_updateNV || nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == CLEAR)
1069        {
1070            // Processing for an index with TPMA_NV_ORDERLY CLEAR
1071            UINT32      entryAddr = NvFindHandle(handle);
1072
1073            pAssert(entryAddr != 0);
1074
1075            // Offset into the index to the first byte of the data to be written
1076            entryAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
1077
1078            // If the data is actually changed, then a write to NV is required
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1079            if(_plat__NvIsDifferent(entryAddr, size, data))
1080            {
1081                // Make sure that NV is available
1082                result = NvIsAvailable();
1083                if(result != TPM_RC_SUCCESS)
1084                    return result;
1085                _plat__NvMemoryWrite(entryAddr, size, data);
1086                g_updateNV = TRUE;
1087            }
1088        }
1089        return TPM_RC_SUCCESS;
1090    }
```

9.4.7.12 NvGetName()

This function is used to compute the Name of an NV 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.

```
1091    UINT16
1092    NvGetName(
1093        TPMI_RH_NV_INDEX     handle,        // IN: handle of the index
1094        NAME                *name           // OUT: name of the index
1095        )
1096    {
1097        UINT16               dataSize, digestSize;
1098        NV_INDEX             nvIndex;
1099        BYTE                 marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
1100        BYTE                *buffer;
1101        HASH_STATE           hashState;
1102
1103        // Get NV public info
1104        NvGetIndexInfo(handle, &nvIndex);
1105
1106        // Marshal public area
1107        buffer = marshalBuffer;
1108        dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex.publicArea, &buffer, NULL);
1109
1110        // hash public area
1111        digestSize = CryptStartHash(nvIndex.publicArea.nameAlg, &hashState);
1112        CryptUpdateDigest(&hashState, dataSize, marshalBuffer);
1113
1114        // Complete digest leaving room for the nameAlg
1115        CryptCompleteHash(&hashState, digestSize, &((BYTE *)name)[2]);
1116
1117        // Include the nameAlg
1118        UINT16_TO_BYTE_ARRAY(nvIndex.publicArea.nameAlg, (BYTE *)name);
1119        return digestSize + 2;
1120    }
```

9.4.7.13 NvDefineIndex()

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

Table 45

Error Returns	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```
1121  TPM_RC
1122  NvDefineIndex(
1123      TPMS_NV_PUBLIC  *publicArea,    // IN: A template for an area to create.
1124      TPM2B_AUTH      *authValue      // IN: The initial authorization value
1125      )
1126  {
1127      // The buffer to be written to NV memory
1128      BYTE            nvBuffer[sizeof(TPM_HANDLE) + sizeof(NV_INDEX)];
1129
1130      NV_INDEX        *nvIndex;           // a pointer to the NV_INDEX data in
1131                                          //   nvBuffer
1132      UINT16          entrySize;          // size of entry
1133
1134      entrySize = sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + publicArea->dataSize;
1135
1136      // Check if we have enough space to create the NV Index
1137      // In this implementation, the only resource limitation is the available NV
1138      // space.  Other implementation may have other limitation on counter or on
1139      // NV slot
1140      if(!NvTestSpace(entrySize, TRUE)) return TPM_RC_NV_SPACE;
1141
1142      // if the index to be defined is RAM backed, check RAM space availability
1143      // as well
1144      if(publicArea->attributes.TPMA_NV_ORDERLY == SET
1145              && !NvTestRAMSpace(publicArea->dataSize))
1146          return TPM_RC_NV_SPACE;
1147
1148
1149      // Copy input value to nvBuffer
1150          // Copy handle
1151      * (TPM_HANDLE *) nvBuffer = publicArea->nvIndex;
1152
1153          // Copy NV_INDEX
1154      nvIndex = (NV_INDEX *) (nvBuffer + sizeof(TPM_HANDLE));
1155      nvIndex->publicArea = *publicArea;
1156      nvIndex->authValue = *authValue;
1157
1158      // Add index to NV memory
1159      NvAdd(entrySize, sizeof(TPM_HANDLE) + sizeof(NV_INDEX), nvBuffer);
1160
1161      // If the data of NV Index is RAM backed, add the data area in RAM as well
1162      if(publicArea->attributes.TPMA_NV_ORDERLY == SET)
1163          NvAddRAM(publicArea->nvIndex, publicArea->dataSize);
1164
1165      return TPM_RC_SUCCESS;
1166  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

9.4.7.14 NvAddEvictObject()

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

© ISO/IEC 2015 – All rights reserved

Table 46

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

```
1167  TPM_RC
1168  NvAddEvictObject(
1169      TPMI_DH_OBJECT   evictHandle,   // IN: new evict handle
1170      OBJECT          *object         // IN: object to be added
1171      )
1172  {
1173      // The buffer to be written to NV memory
1174      BYTE            nvBuffer[sizeof(TPM_HANDLE) + sizeof(OBJECT)];
1175
1176      OBJECT          *nvObject;          // a pointer to the OBJECT data in
1177                                          // nvBuffer
1178      UINT16          entrySize;          // size of entry
1179
1180      // evict handle type should match the object hierarchy
1181      pAssert(   (   NvIsPlatformPersistentHandle(evictHandle)
1182                  && object->attributes.ppsHierarchy == SET)
1183              || (   NvIsOwnerPersistentHandle(evictHandle)
1184                  && (   object->attributes.spsHierarchy == SET
1185                      || object->attributes.epsHierarchy == SET)));
1186
1187      // An evict needs 4 bytes of handle + sizeof OBJECT
1188      entrySize = sizeof(TPM_HANDLE) + sizeof(OBJECT);
1189
1190      // Check if we have enough space to add the evict object
1191      // An evict object needs 8 bytes in index table + sizeof OBJECT
1192      // In this implementation, the only resource limitation is the available NV
1193      // space.  Other implementation may have other limitation on evict object
1194      // handle space
1195      if(!NvTestSpace(entrySize, FALSE)) return TPM_RC_NV_SPACE;
1196
1197      // Allocate a new evict handle
1198      if(!NvIsUndefinedEvictHandle(evictHandle))
1199          return TPM_RC_NV_DEFINED;
1200
1201      // Copy evict object to nvBuffer
1202          // Copy handle
1203      * (TPM_HANDLE *) nvBuffer = evictHandle;
1204
1205          // Copy OBJECT
1206      nvObject = (OBJECT *) (nvBuffer + sizeof(TPM_HANDLE));
1207      *nvObject = *object;
1208
1209      // Set evict attribute and handle
1210      nvObject->attributes.evict = SET;
1211      nvObject->evictHandle = evictHandle;
1212
1213      // Add evict to NV memory
1214      NvAdd(entrySize, entrySize, nvBuffer);
1215
1216      return TPM_RC_SUCCESS;
1217
1218  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

9.4.7.15 NvDeleteEntity()

This function will delete a NV Index or an evict object.

 © ISO/IEC 2015 – All rights reserved

This function requires that the index/evict object has been defined.

```
1219  void
1220  NvDeleteEntity(
1221      TPM_HANDLE       handle         // IN: handle of entity to be deleted
1222      )
1223  {
1224      UINT32      entityAddr;     // pointer to entity
1225
1226      entityAddr = NvFindHandle(handle);
1227      pAssert(entityAddr != 0);
1228
1229      if(HandleGetType(handle) == TPM_HT_NV_INDEX)
1230      {
1231          NV_INDEX    nvIndex;
1232
1233          // Read the NV Index info
1234          _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
1235                              &nvIndex);
1236
1237          // If the entity to be deleted is a counter with the maximum counter
1238          // value, record it in NV memory
1239          if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
1240                  && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
1241          {
1242              UINT64      countValue;
1243              UINT64      maxCount;
1244              NvGetIntIndexData(handle, &nvIndex, &countValue);
1245              maxCount = NvReadMaxCount();
1246              if(countValue > maxCount)
1247                  NvWriteMaxCount(countValue);
1248          }
1249          // If the NV Index is RAM back, delete the RAM data as well
1250          if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
1251              NvDeleteRAM(handle);
1252      }
1253      NvDelete(entityAddr);
1254
1255      return;
1256
1257  }
```

9.4.7.16 NvFlushHierarchy()

This function will delete persistent objects belonging to the indicated If the storage hierarchy is selected, the function will also delete any NV Index define using *ownerAuth*.

```
1258  void
1259  NvFlushHierarchy(
1260      TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flushed.
1261      )
1262  {
1263      NV_ITER         iter = NV_ITER_INIT;
1264      UINT32          currentAddr;
1265
1266      while((currentAddr = NvNext(&iter)) != 0)
1267      {
1268          TPM_HANDLE      entityHandle;
1269
1270          // Read handle information.
1271          _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1272
1273          if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
1274          {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1275            // Handle NV Index
1276            NV_INDEX    nvIndex;
1277
1278            // If flush endorsement or platform hierarchy, no NV Index would be
1279            // flushed
1280            if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
1281                continue;
1282            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1283                                sizeof(NV_INDEX), &nvIndex);
1284
1285            // For storage hierarchy, flush OwnerCreated index
1286            if(  nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
1287            {
1288                // Delete the NV Index
1289                NvDelete(currentAddr);
1290
1291                // Re-iterate from beginning after a delete
1292                iter = NV_ITER_INIT;
1293
1294                // If the NV Index is RAM back, delete the RAM data as well
1295                if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
1296                    NvDeleteRAM(entityHandle);
1297            }
1298        }
1299        else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
1300        {
1301            OBJECT          object;
1302
1303            // Get evict object
1304            NvGetEvictObject(entityHandle, &object);
1305
1306            // If the evict object belongs to the hierarchy to be flushed
1307            if(   (    hierarchy == TPM_RH_PLATFORM
1308                   &&  object.attributes.ppsHierarchy == SET)
1309               || (    hierarchy == TPM_RH_OWNER
1310                   &&  object.attributes.spsHierarchy == SET)
1311               || (    hierarchy == TPM_RH_ENDORSEMENT
1312                   &&  object.attributes.epsHierarchy == SET)
1313               )
1314            {
1315                // Delete the evict object
1316                NvDelete(currentAddr);
1317
1318                // Re-iterate from beginning after a delete
1319                iter = NV_ITER_INIT;
1320            }
1321        }
1322        else
1323        {
1324            pAssert(FALSE);
1325        }
1326    }
1327
1328    return;
1329 }
```

9.4.7.17 NvSetGlobalLock()

This function is used to SET the TPMA_NV_WRITELOCKED attribute for all NV Indices that have TPMA_NV_GLOBALLOCK SET. This function is use by TPM2_NV_GlobalWriteLock().

```
1330 void
1331 NvSetGlobalLock(
1332     void
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
1333        )
1334    {
1335        NV_ITER         iter = NV_ITER_INIT;
1336        UINT32          currentAddr;
1337
1338        // Check all Indices
1339        while((currentAddr = NvNextIndex(&iter)) != 0)
1340        {
1341            NV_INDEX    nvIndex;
1342
1343            // Read the index data
1344            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1345                                sizeof(NV_INDEX), &nvIndex);
1346
1347            // See if it should be locked
1348            if(nvIndex.publicArea.attributes.TPMA_NV_GLOBALLOCK == SET)
1349            {
1350
1351                // if so, lock it
1352                nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED = SET;
1353
1354                _plat__NvMemoryWrite(currentAddr + sizeof(TPM_HANDLE),
1355                                     sizeof(NV_INDEX), &nvIndex);
1356                // Set the flag that a NV write happens
1357                g_updateNV = TRUE;
1358            }
1359        }
1360
1361        return;
1362
1363    }
```

9.4.7.18 InsertSort()

Sort a handle into handle list in ascending order. The total handle number in the list should not exceed MAX_CAP_HANDLES.

```
1364    static void
1365    InsertSort(
1366        TPML_HANDLE     *handleList,    // IN/OUT: sorted handle list
1367        UINT32           count,         // IN: maximum count in the handle list
1368        TPM_HANDLE       entityHandle   // IN: handle to be inserted
1369        )
1370    {
1371        UINT32          i, j;
1372        UINT32          originalCount;
1373
1374        // For a corner case that the maximum count is 0, do nothing
1375        if(count == 0) return;
1376
1377        // For empty list, add the handle at the beginning and return
1378        if(handleList->count == 0)
1379        {
1380            handleList->handle[0] = entityHandle;
1381            handleList->count++;
1382            return;
1383        }
1384
1385        // Check if the maximum of the list has been reached
1386        originalCount = handleList->count;
1387        if(originalCount < count)
1388            handleList->count++;
1389
1390        // Insert the handle to the list
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1391        for(i = 0; i < originalCount; i++)
1392        {
1393            if(handleList->handle[i] > entityHandle)
1394            {
1395                for(j = handleList->count - 1; j > i; j--)
1396                {
1397                    handleList->handle[j] = handleList->handle[j-1];
1398                }
1399                break;
1400            }
1401        }
1402
1403        // If a slot was found, insert the handle in this position
1404        if(i < originalCount || handleList->count > originalCount)
1405            handleList->handle[i] = entityHandle;
1406
1407        return;
1408    }
```

9.4.7.19 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 47

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
1409    TPMI_YES_NO
1410    NvCapGetPersistent(
1411        TPMI_DH_OBJECT   handle,        // IN: start handle
1412        UINT32           count,         // IN: maximum number of returned handles
1413        TPML_HANDLE     *handleList     // OUT: list of handle
1414        )
1415    {
1416        TPMI_YES_NO         more = NO;
1417        NV_ITER             iter = NV_ITER_INIT;
1418        UINT32              currentAddr;
1419
1420        pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1421
1422        // Initialize output handle list
1423        handleList->count = 0;
1424
1425        // The maximum count of handles we may return is MAX_CAP_HANDLES
1426        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1427
1428        while((currentAddr = NvNextEvict(&iter)) != 0)
1429        {
1430            TPM_HANDLE      entityHandle;
1431
1432            // Read handle information.
1433            _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1434
1435            // Ignore persistent handles that have values less than the input handle
1436            if(entityHandle < handle)
1437                continue;
1438
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1439            // if the handles in the list have reached the requested count, and there
1440            // are still handles need to be inserted, indicate that there are more.
1441            if(handleList->count == count)
1442                more = YES;
1443
1444            // A handle with a value larger than start handle is a candidate
1445            // for return. Insert sort it to the return list.  Insert sort algorithm
1446            // is chosen here for simplicity based on the assumption that the total
1447            // number of NV Indices is small.  For an implementation that may allow
1448            // large number of NV Indices, a more efficient sorting algorithm may be
1449            // used here.
1450            InsertSort(handleList, count, entityHandle);
1451
1452        }
1453        return more;
1454    }
```

9.4.7.20 NvCapGetIndex()

This function returns a list of handles of NV Indices, starting from *handle*. *Handle* must be in the range of NV Indices, but does not have to reference an existing NV Index.

Table 48

Return Value	Meaning
YES	if there are more handles to report
NO	all the available handles has been reported

```
1455    TPMI_YES_NO
1456    NvCapGetIndex(
1457        TPMI_DH_OBJECT   handle,        // IN: start handle
1458        UINT32           count,         // IN: maximum number of returned handles
1459        TPML_HANDLE     *handleList     // OUT: list of handle
1460        )
1461    {
1462        TPMI_YES_NO          more = NO;
1463        NV_ITER              iter = NV_ITER_INIT;
1464        UINT32               currentAddr;
1465
1466        pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1467
1468        // Initialize output handle list
1469        handleList->count = 0;
1470
1471        // The maximum count of handles we may return is MAX_CAP_HANDLES
1472        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1473
1474        while((currentAddr = NvNextIndex(&iter)) != 0)
1475        {
1476            TPM_HANDLE      entityHandle;
1477
1478            // Read handle information.
1479            _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1480
1481            // Ignore index handles that have values less than the 'handle'
1482            if(entityHandle < handle)
1483                continue;
1484
1485            // if the count of handles in the list has reached the requested count,
1486            // and there are still handles to report, set more.
1487            if(handleList->count == count)
1488                more = YES;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1489
1490            // A handle with a value larger than start handle is a candidate
1491            // for return. Insert sort it to the return list.  Insert sort algorithm
1492            // is chosen here for simplicity based on the assumption that the total
1493            // number of NV Indices is small.  For an implementation that may allow
1494            // large number of NV Indices, a more efficient sorting algorithm may be
1495            // used here.
1496            InsertSort(handleList, count, entityHandle);
1497        }
1498        return more;
1499    }
```

9.4.7.21 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```
1500    UINT32
1501    NvCapGetIndexNumber(
1502        void
1503        )
1504    {
1505        UINT32          num = 0;
1506        NV_ITER         iter = NV_ITER_INIT;
1507
1508        while(NvNextIndex(&iter) != 0) num++;
1509
1510        return num;
1511    }
```

9.4.7.22 NvCapGetPersistentNumber()

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

```
1512    UINT32
1513    NvCapGetPersistentNumber(
1514        void
1515        )
1516    {
1517        UINT32          num = 0;
1518        NV_ITER         iter = NV_ITER_INIT;
1519
1520        while(NvNextEvict(&iter) != 0) num++;
1521
1522        return num;
1523    }
```

9.4.7.23 NvCapGetPersistentAvail()

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

```
1524    UINT32
1525    NvCapGetPersistentAvail(
1526        void
1527        )
1528    {
1529        UINT32          availSpace;
1530        UINT32          objectSpace;
1531
1532        // Compute the available space in NV storage
1533        availSpace = NvGetFreeByte();
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
1534
1535        // Get the space needed to add a persistent object to NV storage
1536        objectSpace = NvGetEvictObjectSize();
1537
1538        return availSpace / objectSpace;
1539    }
```

9.4.7.24 NvCapGetCounterNumber()

Get the number of defined NV Indexes that have NV TPMA_NV_COUNTER attribute SET.

```
1540    UINT32
1541    NvCapGetCounterNumber(
1542        void
1543        )
1544    {
1545        NV_ITER         iter = NV_ITER_INIT;
1546        UINT32          currentAddr;
1547        UINT32          num = 0;
1548
1549        while((currentAddr = NvNextIndex(&iter)) != 0)
1550        {
1551            NV_INDEX    nvIndex;
1552
1553            // Get NV Index info
1554            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1555                                sizeof(NV_INDEX), &nvIndex);
1556            if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET) num++;
1557        }
1558
1559        return num;
1560    }
```

9.4.7.25 NvCapGetCounterAvail()

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

```
1561    UINT32
1562    NvCapGetCounterAvail(
1563        void
1564        )
1565    {
1566        UINT32          availNVSpace;
1567        UINT32          availRAMSpace;
1568        UINT32          counterNVSpace;
1569        UINT32          counterRAMSpace;
1570        UINT32          persistentNum = NvCapGetPersistentNumber();
1571
1572        // Get the available space in NV storage
1573        availNVSpace = NvGetFreeByte();
1574
1575        if (persistentNum < MIN_EVICT_OBJECTS)
1576        {
1577            // Some space have to be reserved for evict object. Adjust availNVSpace.
1578            UINT32      reserved = (MIN_EVICT_OBJECTS - persistentNum)
1579                                   * NvGetEvictObjectSize();
1580            if (reserved > availNVSpace)
1581                availNVSpace = 0;
1582            else
1583                availNVSpace -= reserved;
1584        }
1585
1586        // Get the space needed to add a counter index to NV storage
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
1587        counterNVSpace = NvGetCounterSize();
1588
1589        // Compute the available space in RAM
1590        availRAMSpace = RAM_INDEX_SPACE - s_ramIndexSize;
1591
1592        // Compute the space needed to add a counter index to RAM storage
1593        // It takes an size field, a handle and sizeof(UINT64) for counter data
1594        counterRAMSpace = sizeof(UINT32) + sizeof(TPM_HANDLE) + sizeof(UINT64);
1595
1596        // Return the min of counter number in NV and in RAM
1597        if(availNVSpace / counterNVSpace > availRAMSpace / counterRAMSpace)
1598            return availRAMSpace / counterRAMSpace;
1599        else
1600            return availNVSpace / counterNVSpace;
1601    }
```

9.5 Object.c

9.5.1 Introduction

This file contains the functions that manage the object store of the TPM.

9.5.2 Includes and Data Definitions

```
1   #define OBJECT_C
2   #include "InternalRoutines.h"
3   #include <Platform.h>
```

9.5.3 Functions

9.5.3.1 ObjectStartup()

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

```
4   void
5   ObjectStartup(
6       void
7       )
8   {
9       UINT32      i;
10
11      // object slots initialization
12      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
13      {
14          //Set the slot to not occupied
15          s_objects[i].occupied = FALSE;
16      }
17      return;
18  }
```

9.5.3.2 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().

```
19  void
20  ObjectCleanupEvict(
21      void
22      )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
{
    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
        if(s_objects[i].object.entity.attributes.evict == SET)
            s_objects[i].occupied = FALSE;
    }

    return;
}
```

9.5.3.3 ObjectIsPresent()

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 49

Return Value	Meaning
TRUE	if the handle references a loaded object
FALSE	if the handle is not an object handle, or it does not reference to a loaded object

```
BOOL
ObjectIsPresent(
    TPMI_DH_OBJECT   handle         // IN: handle to be checked
    )
{
    UINT32          slotIndex;      // index of object slot

    pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);

    // The index in the loaded object array is found by subtracting the first
    // object handle number from the input handle number. If the indicated
    // slot is occupied, then indicate that there is already is a loaded
    //  object associated with the handle.
    slotIndex = handle - TRANSIENT_FIRST;
    if(slotIndex >= MAX_LOADED_OBJECTS)
        return FALSE;

    return s_objects[slotIndex].occupied;
}
```

9.5.3.4 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 50

Return Value	Meaning
TRUE	object is an HMAC, hash, or event sequence object
FALSE	object is not an HMAC, hash, or event sequence object

```
57  BOOL
58  ObjectIsSequence(
59      OBJECT          *object         // IN: handle to be checked
60      )
61  {
62      pAssert (object != NULL);
63      if(   object->attributes.hmacSeq == SET
64         || object->attributes.hashSeq == SET
65         || object->attributes.eventSeq == SET)
66          return TRUE;
67      else
68          return FALSE;
69  }
```

9.5.3.5 ObjectGet()

This function is used to find the object structure associated with a handle.

This function requires that *handle* references a loaded object.

```
70  OBJECT*
71  ObjectGet(
72      TPMI_DH_OBJECT   handle          // IN: handle of the object
73      )
74  {
75      pAssert(   handle >= TRANSIENT_FIRST
76              && handle - TRANSIENT_FIRST < MAX_LOADED_OBJECTS);
77      pAssert(s_objects[handle - TRANSIENT_FIRST].occupied == TRUE);
78
79      // In this implementation, the handle is determined by the slot occupied by the
80      // object.
81      return &s_objects[handle - TRANSIENT_FIRST].object.entity;
82  }
```

9.5.3.6 ObjectGetName()

This function is used to access the Name of the object. In this implementation, the Name is computed when the object is loaded and is saved in the internal representation of the object. This function copies the Name data from the object into the buffer at *name* and returns the number of octets copied.

This function requires that *handle* references a loaded object.

```
83  UINT16
84  ObjectGetName(
85      TPMI_DH_OBJECT   handle,         // IN: handle of the object
86      NAME            *name            // OUT: name of the object
87      )
88  {
89      OBJECT      *object = ObjectGet(handle);
90      if(object->publicArea.nameAlg == TPM_ALG_NULL)
91          return 0;
92
93      // Copy the Name data to the output
94      MemoryCopy(name, object->name.t.name, object->name.t.size, sizeof(NAME));
95      return  object->name.t.size;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
}
```

9.5.3.7 ObjectGetNameAlg()

This function is used to get the Name algorithm of a object.

This function requires that *handle* references a loaded object.

```
TPMI_ALG_HASH
ObjectGetNameAlg(
    TPMI_DH_OBJECT   handle         // IN: handle of the object
    )
{
    OBJECT              *object = ObjectGet(handle);

    return object->publicArea.nameAlg;
}
```

9.5.3.8 ObjectGetQualifiedName()

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.

```
void
ObjectGetQualifiedName(
    TPMI_DH_OBJECT   handle,        // IN: handle of the object
    TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
    )
{
    OBJECT      *object = ObjectGet(handle);
    if(object->publicArea.nameAlg == TPM_ALG_NULL)
        qualifiedName->t.size = 0;
    else
        // Copy the name
        *qualifiedName = object->qualifiedName;

    return;
}
```

9.5.3.9 ObjectDataGetHierarchy()

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

```
TPMI_RH_HIERARCHY
ObjectDataGetHierarchy(
    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)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
    {
        return TPM_RH_PLATFORM;
    }
    else
    {
        return TPM_RH_NULL;
    }

}
```

9.5.3.10 ObjectGetHierarchy()

This function returns the handle of the hierarchy to which a handle belongs. This function is similar to ObjectDataGetHierarchy() but this routine takes a handle but ObjectDataGetHierarchy() takes a pointer to an object.

This function requires that *handle* references a loaded object.

```
TPMI_RH_HIERARCHY
ObjectGetHierarchy(
    TPMI_DH_OBJECT   handle         // IN :object handle
    )
{
    OBJECT          *object = ObjectGet(handle);

    return ObjectDataGetHierarchy(object);
}
```

9.5.3.11 ObjectAllocateSlot()

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

Table 51

Return Value	Meaning
TRUE	allocate success
FALSE	do not have free slot

```
static BOOL
ObjectAllocateSlot(
    TPMI_DH_OBJECT  *handle,        // OUT: handle of allocated object
    OBJECT          **object        // OUT: points to the allocated object
    )
{
    UINT32          i;

    // find an unoccupied handle slot
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        if(!s_objects[i].occupied)          // If found a free slot
        {
            // Mark the slot as occupied
            s_objects[i].occupied = TRUE;
            break;
        }
    }
    // If we reach the end of object slot without finding a free one, return
    // error.
    if(i == MAX_LOADED_OBJECTS) return FALSE;

```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
175         *handle = i + TRANSIENT_FIRST;
176         *object = &s_objects[i].object.entity;
177
178         // Initialize the object attributes
179         MemorySet(&((*object)->attributes), 0, sizeof(OBJECT_ATTRIBUTES));
180
181         return TRUE;
182     }
```

9.5.3.12 ObjectLoad()

This function loads an object into an internal object structure. If an error is returned, the internal state is unchanged.

Table 52

Error Returns	Meaning
TPM_RC_BINDING	if the public and sensitive parts of the object are not matched
TPM_RC_KEY	if the parameters in the public area of the object are not consistent
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object
TPM_RC_TYPE	the public and private parts are not the same type

```
183     TPM_RC
184     ObjectLoad(
185         TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy to which the object belongs
186         TPMT_PUBLIC         *publicArea,     // IN: public area
187         TPMT_SENSITIVE      *sensitive,      // IN: sensitive area (may be null)
188         TPM2B_NAME          *name,           // IN: object's name (may be null)
189         TPM_HANDLE           parentHandle,   // IN: handle of parent
190         BOOL                 skipChecks,     // IN: flag to indicate if it is OK to skip
191                                              //     consistency checks.
192         TPMI_DH_OBJECT      *handle          // OUT: object handle
193         )
194     {
195         OBJECT               *object = NULL;
196         OBJECT               *parent = NULL;
197         TPM_RC                result = TPM_RC_SUCCESS;
198         TPM2B_NAME            parentQN;         // Parent qualified name
199
200         // Try to allocate a slot for new object
201         if(!ObjectAllocateSlot(handle, &object))
202             return TPM_RC_OBJECT_MEMORY;
203
204         // Initialize public
205         object->publicArea = *publicArea;
206         if(sensitive != NULL)
207            object->sensitive = *sensitive;
208
209         // Are the consistency checks needed
210         if(!skipChecks)
211         {
212             // Check if key size matches
213             if(!CryptObjectIsPublicConsistent(&object->publicArea))
214             {
215                 result = TPM_RC_KEY;
216                 goto ErrorExit;
217             }
218             if(sensitive != NULL)
219             {
220                 // Check if public type matches sensitive type
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
221             result = CryptObjectPublicPrivateMatch(object);
222             if(result != TPM_RC_SUCCESS)
223                 goto ErrorExit;
224         }
225     }
226     object->attributes.publicOnly = (sensitive == NULL);
227
228     // If 'name' is NULL, then there is nothing left to do for this
229     // object as it has no qualified name and it is not a member of any
230     // hierarchy and it is temporary
231     if(name == NULL || name->t.size == 0)
232     {
233         object->qualifiedName.t.size = 0;
234         object->name.t.size = 0;
235         object->attributes.temporary = SET;
236         return TPM_RC_SUCCESS;
237     }
238     // If parent handle is a permanent handle, it is a primary or temporary
239     // object
240     if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
241     {
242         // initialize QN
243         parentQN.t.size = 4;
244
245         // for a primary key, parent qualified name is the handle of hierarchy
246         UINT32_TO_BYTE_ARRAY(parentHandle, parentQN.t.name);
247     }
248     else
249     {
250         // Get hierarchy and qualified name of parent
251         ObjectGetQualifiedName(parentHandle, &parentQN);
252
253         // Check for stClear object
254         parent = ObjectGet(parentHandle);
255         if(   publicArea->objectAttributes.stClear == SET
256            || parent->attributes.stClear == SET)
257             object->attributes.stClear = SET;
258
259     }
260     object->name = *name;
261
262     // Compute object qualified name
263     ObjectComputeQualifiedName(&parentQN, publicArea->nameAlg,
264                                name, &object->qualifiedName);
265
266     // Any object in TPM_RH_NULL hierarchy is temporary
267     if(hierarchy == TPM_RH_NULL)
268     {
269         object->attributes.temporary = SET;
270     }
271     else if(parentQN.t.size == sizeof(TPM_HANDLE))
272     {
273         // Otherwise, if the size of parent's qualified name is the size of a
274         // handle, this object is a primary object
275         object->attributes.primary = SET;
276     }
277     switch(hierarchy)
278     {
279         case TPM_RH_PLATFORM:
280             object->attributes.ppsHierarchy = SET;
281             break;
282         case TPM_RH_OWNER:
283             object->attributes.spsHierarchy = SET;
284             break;
285         case TPM_RH_ENDORSEMENT:
286             object->attributes.epsHierarchy = SET;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
            break;
        case TPM_RH_NULL:
            break;
        default:
            pAssert(FALSE);
            break;
    }
    return TPM_RC_SUCCESS;

ErrorExit:
    ObjectFlush(*handle);
    return result;
}
```

9.5.3.13 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 BOOL
AllocateSequenceSlot(
    TPM_HANDLE      *newHandle,     // OUT: receives the allocated handle
    HASH_OBJECT     **object,       // OUT: receives pointer to allocated object
    TPM2B_AUTH      *auth           // IN: the authValue for the slot
    )
{
    OBJECT              *objectHash;        // the hash as an object

    if(!ObjectAllocateSlot(newHandle, &objectHash))
        return FALSE;

    *object = (HASH_OBJECT *)objectHash;

    // Validate that the proper location of the hash state data relative to the
    // object state data.
    pAssert(&((*object)->auth) == &objectHash->publicArea.authPolicy);

    // Set the common values that a sequence object shares with an ordinary 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;

    // Clear the attributes
    MemorySet(&((*object)->objectAttributes), 0, sizeof(TPMA_OBJECT));

    // A sequence object is DA exempt.
    (*object)->objectAttributes.noDA = SET;

    if(auth != NULL)
    {
        MemoryRemoveTrailingZeros(auth);
        (*object)->auth = *auth;
    }
    else
        (*object)->auth.t.size = 0;
    return TRUE;
}
```

9.5.3.14 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

Table 53

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

```
340 TPM_RC
341 ObjectCreateHMACSequence(
342     TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
343     TPM_HANDLE       handle,         // IN: the handle associated with sequence
344                                      //     object
345     TPM2B_AUTH      *auth,           // IN: authValue
346     TPMI_DH_OBJECT  *newHandle       // OUT: HMAC sequence object handle
347     )
348 {
349     HASH_OBJECT         *hmacObject;
350     OBJECT              *keyObject;
351
352     // Try to allocate a slot for new object
353     if(!AllocateSequenceSlot(newHandle, &hmacObject, auth))
354         return TPM_RC_OBJECT_MEMORY;
355
356     // Set HMAC sequence bit
357     hmacObject->attributes.hmacSeq = SET;
358
359     // Get pointer to the HMAC key object
360     keyObject = ObjectGet(handle);
361
362     CryptStartHMACSequence2B(hashAlg, &keyObject->sensitive.sensitive.bits.b,
363                              &hmacObject->state.hmacState);
364
365     return TPM_RC_SUCCESS;
366 }
```

9.5.3.15 ObjectCreateHashSequence()

This function creates a hash sequence object.

Table 54

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

```
367 TPM_RC
368 ObjectCreateHashSequence(
369     TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
370     TPM2B_AUTH      *auth,           // IN: authValue
371     TPMI_DH_OBJECT  *newHandle       // OUT: sequence object handle
372     )
373 {
374     HASH_OBJECT         *hashObject;
375
376     // Try to allocate a slot for new object
377     if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
378         return TPM_RC_OBJECT_MEMORY;
379
380     // Set hash sequence bit
381     hashObject->attributes.hashSeq = SET;
382
383     // Start hash for hash sequence
384     CryptStartHashSequence(hashAlg, &hashObject->state.hashState[0]);
385
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
386         return TPM_RC_SUCCESS;
387     }
```

9.5.3.16 ObjectCreateEventSequence()

This function creates an event sequence object.

Table 55

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

```
388     TPM_RC
389     ObjectCreateEventSequence(
390         TPM2B_AUTH      *auth,          // IN: authValue
391         TPMI_DH_OBJECT  *newHandle      // OUT: sequence object handle
392         )
393     {
394         HASH_OBJECT         *hashObject;
395         UINT32               count;
396         TPM_ALG_ID           hash;
397
398         // Try to allocate a slot for new object
399         if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
400             return TPM_RC_OBJECT_MEMORY;
401
402         // Set the event sequence attribute
403         hashObject->attributes.eventSeq = SET;
404
405
406         // Initialize hash states for each implemented PCR algorithms
407         for(count = 0; (hash = CryptGetHashAlgByIndex(count)) != TPM_ALG_NULL; count++)
408         {
409             // If this is a _TPM_Init or _TPM_HashStart, the sequence object will
410             // not leave the TPM so it doesn't need the sequence handling
411             if(auth == NULL)
412                 CryptStartHash(hash, &hashObject->state.hashState[count]);
413             else
414                 CryptStartHashSequence(hash, &hashObject->state.hashState[count]);
415         }
416         return TPM_RC_SUCCESS;
417     }
```

9.5.3.17 ObjectTerminateEvent()

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

```
418     void
419     ObjectTerminateEvent(
420         void
421         )
422     {
423         HASH_OBJECT         *hashObject;
424         int                  count;
425         BYTE                 buffer[MAX_DIGEST_SIZE];
426         hashObject = (HASH_OBJECT *)ObjectGet(g_DRTMHandle);
427
428         // Don't assume that this is a proper sequence object
429         if(hashObject->attributes.eventSeq)
430         {
431             // If it is, close any open hash contexts. This is done in case
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
        // the crypto implementation has some context values that need to be
        // cleaned up (hygiene).
        //
        for(count = 0; CryptGetHashAlgByIndex(count) != TPM_ALG_NULL; count++)
        {
            CryptCompleteHash(&hashObject->state.hashState[count], 0, buffer);
        }
        // Flush sequence object
        ObjectFlush(g_DRTMHandle);
    }

    g_DRTMHandle = TPM_RH_UNASSIGNED;
}
```

9.5.3.18 ObjectContextLoad()

This function loads an object from a saved object context.

Table 56

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

```
TPM_RC
ObjectContextLoad(
    OBJECT          *object,        // IN: object structure from saved context
    TPMI_DH_OBJECT  *handle         // OUT: object handle
    )
{
    OBJECT      *newObject;

    // Try to allocate a slot for new object
    if(!ObjectAllocateSlot(handle, &newObject))
        return TPM_RC_OBJECT_MEMORY;

    // Copy input object data to internal structure
    *newObject = *object;

    return TPM_RC_SUCCESS;
}
```

9.5.3.19 ObjectFlush()

This function frees an object slot.

This function requires that the object is loaded.

```
void
ObjectFlush(
    TPMI_DH_OBJECT   handle         // IN: handle to be freed
    )
{
    UINT32      index = handle - TRANSIENT_FIRST;
    pAssert(ObjectIsPresent(handle));

    // Mark the handle slot as unoccupied
    s_objects[index].occupied = FALSE;

    // With no attributes
    MemorySet((BYTE*)&(s_objects[index].object.entity.attributes),
              0, sizeof(OBJECT_ATTRIBUTES));
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
476         return;
477     }
```

9.5.3.20 ObjectFlushHierarchy()

This function is called to flush all the loaded transient objects associated with a hierarchy when the hierarchy is disabled.

```
478     void
479     ObjectFlushHierarchy(
480         TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flush
481         )
482     {
483         UINT16          i;
484
485         // iterate object slots
486         for(i = 0; i < MAX_LOADED_OBJECTS; i++)
487         {
488             if(s_objects[i].occupied)          // If found an occupied slot
489             {
490                 switch(hierarchy)
491                 {
492                     case TPM_RH_PLATFORM:
493                         if(s_objects[i].object.entity.attributes.ppsHierarchy == SET)
494                             s_objects[i].occupied = FALSE;
495                         break;
496                     case TPM_RH_OWNER:
497                         if(s_objects[i].object.entity.attributes.spsHierarchy == SET)
498                             s_objects[i].occupied = FALSE;
499                         break;
500                     case TPM_RH_ENDORSEMENT:
501                         if(s_objects[i].object.entity.attributes.epsHierarchy == SET)
502                             s_objects[i].occupied = FALSE;
503                         break;
504                     default:
505                         pAssert(FALSE);
506                         break;
507                 }
508             }
509         }
510
511         return;
512
513     }
```

9.5.3.21 ObjectLoadEvict()

This function loads a persistent object into a transient object slot.

This function requires that *handle* is associated with a persistent object.

Table 57

Error Returns	Meaning
TPM_RC_HANDLE	the persistent object does not exist or the associated hierarchy is disabled.
TPM_RC_OBJECT_MEMORY	no object slot

```
514     TPM_RC
515     ObjectLoadEvict(
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
516         TPM_HANDLE      *handle,        // IN:OUT: evict object handle.  If success, it
517                                         // will be replace by the loaded object handle
518         TPM_CC           commandCode    // IN: the command being processed
519         )
520     {
521         TPM_RC          result;
522         TPM_HANDLE      evictHandle = *handle;   // Save the evict handle
523         OBJECT          *object;
524
525         // If this is an index that references a persistent object created by
526         // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
527         if(*handle >=  PLATFORM_PERSISTENT)
528         {
529             // belongs to platform
530             if(g_phEnable == CLEAR)
531                 return TPM_RC_HANDLE;
532         }
533         // belongs to owner
534         else if(gc.shEnable == CLEAR)
535             return TPM_RC_HANDLE;
536
537         // Try to allocate a slot for an object
538         if(!ObjectAllocateSlot(handle, &object))
539             return TPM_RC_OBJECT_MEMORY;
540
541         // Copy persistent object to transient object slot.  A TPM_RC_HANDLE
542         // may be returned at this point. This will mark the slot as containing
543         // a transient object so that it will be flushed at the end of the
544         // command
545         result = NvGetEvictObject(evictHandle, object);
546
547         // Bail out if this failed
548         if(result != TPM_RC_SUCCESS)
549             return result;
550
551         // check the object to see if it is in the endorsement hierarchy
552         // if it is and this is not a TPM2_EvictControl() command, indicate
553         // that the hierarchy is disabled.
554         // If the associated hierarchy is disabled, make it look like the
555         // handle is not defined
556         if(    ObjectDataGetHierarchy(object) == TPM_RH_ENDORSEMENT
557             && gc.ehEnable == CLEAR
558             && commandCode != TPM_CC_EvictControl
559            )
560             return TPM_RC_HANDLE;
561
562         return result;
563     }
```

9.5.3.22 ObjectComputeName()

This function computes the Name of an object from its public area.

```
564     void
565     ObjectComputeName(
566         TPMT_PUBLIC     *publicArea,    // IN: public area of an object
567         TPM2B_NAME      *name           // OUT: name of the object
568         )
569     {
570         TPM2B_PUBLIC         marshalBuffer;
571         BYTE                *buffer;            // auxiliary marshal buffer pointer
572         HASH_STATE           hashState;         // hash state
573
574         // if the nameAlg is NULL then there is no name.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
575         if(publicArea->nameAlg == TPM_ALG_NULL)
576         {
577             name->t.size = 0;
578             return;
579         }
580         // Start hash stack
581         name->t.size = CryptStartHash(publicArea->nameAlg, &hashState);
582
583         // Marshal the public area into its canonical form
584         buffer = marshalBuffer.b.buffer;
585
586         marshalBuffer.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
587
588         // Adding public area
589         CryptUpdateDigest2B(&hashState, &marshalBuffer.b);
590
591         // Complete hash leaving room for the name algorithm
592         CryptCompleteHash(&hashState, name->t.size, &name->t.name[2]);
593
594         // set the nameAlg
595         UINT16_TO_BYTE_ARRAY(publicArea->nameAlg, name->t.name);
596         name->t.size += 2;
597         return;
598     }
```

9.5.3.23 ObjectComputeQualifiedName()

This function computes the qualified name of an object.

```
599     void
600     ObjectComputeQualifiedName(
601         TPM2B_NAME      *parentQN,      // IN: parent's qualified name
602         TPM_ALG_ID       nameAlg,       // IN: name hash
603         TPM2B_NAME      *name,          // IN: name of the object
604         TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
605         )
606     {
607         HASH_STATE      hashState;  // hash state
608
609         //      QN_A = hash_A (QN of parent || NAME_A)
610
611         // Start hash
612         qualifiedName->t.size = CryptStartHash(nameAlg, &hashState);
613
614         // Add parent's qualified name
615         CryptUpdateDigest2B(&hashState, &parentQN->b);
616
617         // Add self name
618         CryptUpdateDigest2B(&hashState, &name->b);
619
620         // Complete hash leaving room for the name algorithm
621         CryptCompleteHash(&hashState, qualifiedName->t.size,
622                           &qualifiedName->t.name[2]);
623         UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
624         qualifiedName->t.size += 2;
625         return;
626     }
```

9.5.3.24 ObjectDataIsStorage()

This function determines if a public area has the attributes associated with a storage key. A storage key is an asymmetric object that has its *restricted* and *decrypt* attributes SET, and *sign* CLEAR.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 58

Return Value	Meaning
TRUE	if the object is a storage key
FALSE	if the object is not a storage key

```
627  BOOL
628  ObjectDataIsStorage(
629      TPMT_PUBLIC     *publicArea     // IN: public area of the object
630      )
631  {
632      if(   CryptIsAsymAlgorithm(publicArea->type)            // must be asymmetric,
633         && publicArea->objectAttributes.restricted == SET    // restricted,
634         && publicArea->objectAttributes.decrypt == SET       // decryption key
635         && publicArea->objectAttributes.sign == CLEAR        // can not be sign key
636        )
637          return TRUE;
638      else
639          return FALSE;
640  }
```

9.5.3.25 ObjectIsStorage()

This function determines if an object has the attributes associated with a storage key. A storage key is an asymmetric object that has its *restricted* and *decrypt* attributes SET, and *sign* CLEAR.

Table 59

Return Value	Meaning
TRUE	if the object is a storage key
FALSE	if the object is not a storage key

```
641  BOOL
642  ObjectIsStorage(
643      TPMI_DH_OBJECT   handle          // IN: object handle
644      )
645  {
646      OBJECT           *object = ObjectGet(handle);
647      return ObjectDataIsStorage(&object->publicArea);
648  }
```

9.5.3.26 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.

Table 60

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
649  TPMI_YES_NO
650  ObjectCapGetLoaded(
651      TPMI_DH_OBJECT   handle,         // IN: start handle
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
652         UINT32           count,         // IN: count of returned handles
653         TPML_HANDLE     *handleList     // OUT: list of handle
654         )
655     {
656         TPMI_YES_NO          more = NO;
657         UINT32               i;
658
659         pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
660
661         // Initialize output handle list
662         handleList->count = 0;
663
664         // The maximum count of handles we may return is MAX_CAP_HANDLES
665         if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
666
667         // Iterate object slots to get loaded object handles
668         for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
669         {
670             if(s_objects[i].occupied == TRUE)
671             {
672                 // A valid transient object can not be the copy of a persistent object
673                 pAssert(s_objects[i].object.entity.attributes.evict == CLEAR);
674
675                 if(handleList->count < count)
676                 {
677                     // If we have not filled up the return list, add this object
678                     // handle to it
679                     handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
680                     handleList->count++;
681                 }
682                 else
683                 {
684                     // If the return list is full but we still have loaded object
685                     // available, report this and stop iterating
686                     more = YES;
687                     break;
688                 }
689             }
690         }
691
692         return more;
693     }
```

9.5.3.27 ObjectCapGetTransientAvail()

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

```
694     UINT32
695     ObjectCapGetTransientAvail(
696         void
697         )
698     {
699         UINT32      i;
700         UINT32      num = 0;
701
702         // Iterate object slot to get the number of unoccupied slots
703         for(i = 0; i < MAX_LOADED_OBJECTS; i++)
704         {
705             if(s_objects[i].occupied == FALSE) num++;
706         }
707
708         return num;
709     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.6 PCR.c

9.6.1 Introduction

This function contains the functions needed for PCR access and manipulation.

This implementation uses a static allocation for the PCR. The amount of memory is allocated based on the number of PCR in the implementation and the number of implemented hash algorithms. This is not the expected implementation. PCR SPACE DEFINITIONS.

In the definitions below, the *g_hashPcrMap* is a bit array that indicates which of the PCR are implemented. The *g_hashPcr* array is an array of digests. In this implementation, the space is allocated whether the PCR is implemented or not.

9.6.2 Includes, Defines, and Data Definitions

```
#define PCR_C
#include "InternalRoutines.h"
#include <Platform.h>
```

The initial value of PCR attributes. The value of these fields should be consistent with platform specific specifications. In this implementation, we assume the total number of implemented PCR is 24.

```
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},
    {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, App specific
    {0, 0x0F, 0x1F}         // PCR 24, testing policy
};
```

9.6.3 Functions

9.6.3.1 PCRBelongsAuthGroup()

This function indicates if a PCR belongs to a group that requires an *authValue* in order to modify the PCR. If it does, *groupIndex* is set to value of the group index. This feature of PCR is decided by the platform specification.

Table 61

Return Value	Meaning
TRUE:	PCR belongs an auth group
FALSE:	PCR does not belong an auth group

```
BOOL
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
PCRBelongsAuthGroup(
    TPMI_DH_PCR      handle,         // IN: handle of PCR
    UINT32          *groupIndex      // OUT: group index if PCR belongs a
                                     //      group that allows authValue.  If PCR
                                     //      does not belong to an auth group,
                                     //      the value in this parameter is
                                     //      invalid
)
{
#if NUM_AUTHVALUE_PCR_GROUP > 0
    // Platform specification determines to which auth group a PCR belongs (if
    // any). In this implementation, we assume there is only
    // one auth group which contains PCR[20-22].  If the platform specification
    // requires differently, the implementation should be changed accordingly
    if(handle >= 20 && handle <= 22)
    {
        *groupIndex = 0;
        return TRUE;
    }

#endif
    return FALSE;
}
```

9.6.3.2 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 62

Return Value	Meaning
TRUE:	PCR belongs a policy group
FALSE:	PCR does not belong a policy group

```
BOOL
PCRBelongsPolicyGroup(
    TPMI_DH_PCR      handle,         // IN: handle of PCR
    UINT32          *groupIndex      // OUT: group index if PCR belongs a group that
                                     //     allows policy.  If PCR does not belong to
                                     //     a policy group, the value in this
                                     //     parameter is invalid
    )
{
#if NUM_POLICY_PCR_GROUP > 0
    // Platform specification decides if a PCR belongs to a policy group and
    // belongs to which group.  In this implementation, we assume there is only
    // one policy group which contains PCR20-22.  If the platform specification
    // requires differently, the implementation should be changed accordingly
    if(handle >= 20 && handle <= 22)
    {
        *groupIndex = 0;
        return TRUE;
    }
#endif
    return FALSE;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.6.3.3 PCRBelongsTCBGroup()

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

Table 63

Return Value	Meaning
TRUE:	PCR belongs to TCB group
FALSE:	PCR does not belong to TCB group

```
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;
}
```

9.6.3.4 PCRPolicyIsAvailable()

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

Table 64

Return Value	Meaning
TRUE	the PCR should be authorized by policy
FALSE	the PCR does not allow policy

```
BOOL
PCRPolicyIsAvailable(
    TPMI_DH_PCR      handle          // IN: PCR handle
    )
{
    UINT32           groupIndex;

    return PCRBelongsPolicyGroup(handle, &groupIndex);
}
```

9.6.3.5 PCRGetAuthValue()

This function is used to access the *authValue* of a PCR. If PCR does not belong to an *authValue* group, an Empty Auth will be returned.

```
void
PCRGetAuthValue(
    TPMI_DH_PCR      handle,         // IN: PCR handle
    TPM2B_AUTH      *auth            // OUT: authValue of PCR
    )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
{
    UINT32      groupIndex;

    if(PCRBelongsAuthGroup(handle, &groupIndex))
    {
        *auth = gc.pcrAuthValues.auth[groupIndex];
    }
    else
    {
        auth->t.size = 0;
    }

    return;
}
```

9.6.3.6 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.

```
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;
    }
}
```

9.6.3.7 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.

```
void
PCRSimStart(
    void
    )
{
    UINT32  i;
    for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
    {
        gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
        gp.pcrPolicies.policy[i].t.size = 0;
    }

    for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
    {
        gc.pcrAuthValues.auth[i].t.size = 0;
    }

```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
148         // We need to give an initial configuration on allocated PCR before
149         // receiving any TPM2_PCR_Allocate command to change this configuration
150         // When the simulation environment starts, we allocate all the PCRs
151         for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
152                 gp.pcrAllocated.count++)
153         {
154             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
155                 = CryptGetHashAlgByIndex(gp.pcrAllocated.count);
156
157             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
158                 = PCR_SELECT_MAX;
159             for(i = 0; i < PCR_SELECT_MAX; i++)
160                 gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
161                     = 0xFF;
162         }
163
164         // Store the initial configuration to NV
165         NvWriteReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
166         NvWriteReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
167
168         return;
169     }
```

9.6.3.8 GetSavedPcrPointer()

This function returns the address of an array of state saved PCR based on the hash algorithm.

Table 65

Return Value	Meaning
NULL	no such algorithm
not NULL	pointer to the 0th byte of the 0th PCR

```
170     static BYTE *
171     GetSavedPcrPointer (
172         TPM_ALG_ID       alg,           // IN: algorithm for bank
173         UINT32           pcrIndex       // IN: PCR index in PCR_SAVE
174         )
175     {
176         switch(alg)
177         {
178     #ifdef TPM_ALG_SHA1
179         case TPM_ALG_SHA1:
180             return gc.pcrSave.sha1[pcrIndex];
181             break;
182     #endif
183     #ifdef TPM_ALG_SHA256
184         case TPM_ALG_SHA256:
185             return gc.pcrSave.sha256[pcrIndex];
186             break;
187     #endif
188     #ifdef TPM_ALG_SHA384
189         case TPM_ALG_SHA384:
190             return gc.pcrSave.sha384[pcrIndex];
191             break;
192     #endif
193
194     #ifdef TPM_ALG_SHA512
195         case TPM_ALG_SHA512:
196             return gc.pcrSave.sha512[pcrIndex];
197             break;
198     #endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
#ifdef TPM_ALG_SM3_256
    case TPM_ALG_SM3_256:
        return gc.pcrSave.sm3_256[pcrIndex];
        break;
#endif
    default:
        FAIL(FATAL_ERROR_INTERNAL);
    }
    //return NULL; // Can't be reached
}
```

9.6.3.9 PcrIsAllocated()

This function indicates if a PCR number for the particular hash algorithm is allocated.

Table 66

Return Value	Meaning
FALSE	PCR is not allocated
TRUE	PCR is allocated

```
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;
}
```

9.6.3.10 GetPcrPointer()

This function returns the address of an array of PCR based on the hash algorithm.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 67

Return Value	Meaning
NULL	no such algorithm
not NULL	pointer to the 0th byte of the 0th PCR

```
236   static BYTE *
237   GetPcrPointer (
238       TPM_ALG_ID       alg,           // IN: algorithm for bank
239       UINT32           pcrNumber      // IN: PCR number
240       )
241   {
242       static BYTE     *pcr = NULL;
243
244       if(!PcrIsAllocated(pcrNumber, alg))
245           return NULL;
246
247       switch(alg)
248       {
249   #ifdef TPM_ALG_SHA1
250       case TPM_ALG_SHA1:
251           pcr = s_pcrs[pcrNumber].sha1Pcr;
252           break;
253   #endif
254   #ifdef TPM_ALG_SHA256
255       case TPM_ALG_SHA256:
256           pcr = s_pcrs[pcrNumber].sha256Pcr;
257           break;
258   #endif
259   #ifdef TPM_ALG_SHA384
260       case TPM_ALG_SHA384:
261           pcr = s_pcrs[pcrNumber].sha384Pcr;
262           break;
263   #endif
264   #ifdef TPM_ALG_SHA512
265       case TPM_ALG_SHA512:
266           pcr = s_pcrs[pcrNumber].sha512Pcr;
267           break;
268   #endif
269   #ifdef TPM_ALG_SM3_256
270       case TPM_ALG_SM3_256:
271           pcr = s_pcrs[pcrNumber].sm3_256Pcr;
272           break;
273   #endif
274       default:
275           pAssert(FALSE);
276           break;
277       }
278
279       return pcr;
280   }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

9.6.3.11 IsPcrSelected()

This function indicates if an indicated PCR number is selected by the bit map in *selection*.

© ISO/IEC 2015 – All rights reserved

Table 68

Return Value	Meaning
FALSE	PCR is not selected
TRUE	PCR is selected

```
281  static BOOL
282  IsPcrSelected (
283      UINT32               pcr,           // IN: The number of the PCR
284      TPMS_PCR_SELECTION  *selection      // IN: The selection structure
285      )
286  {
287      BOOL                 selected = FALSE;
288      if(   pcr < IMPLEMENTATION_PCR
289         && ((selection->pcrSelect[pcr/8]) & (1 << (pcr % 8))) != 0)
290          selected = TRUE;
291
292      return selected;
293  }
```

9.6.3.12 FilterPcr()

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

```
294  static void
295  FilterPcr(
296      TPMS_PCR_SELECTION  *selection      // IN: input PCR selection
297      )
298  {
299      UINT32     i;
300      TPMS_PCR_SELECTION      *allocated = NULL;
301
302      // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
303      for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
304          selection->pcrSelect[i] = 0;
305
306      // Find the internal configuration for the bank
307      for(i = 0; i < gp.pcrAllocated.count; i++)
308      {
309          if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
310          {
311              allocated = &gp.pcrAllocated.pcrSelections[i];
312              break;
313          }
314      }
315
316      for (i = 0; i < selection->sizeofSelect; i++)
317      {
318          if(allocated == NULL)
319          {
320              // If the required bank does not exist, clear input selection
321              selection->pcrSelect[i] = 0;
322          }
323          else
324              selection->pcrSelect[i] &= allocated->pcrSelect[i];
325      }
326
327      return;
328  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.6.3.13 PcrDrtm()

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

```
329  void
330  PcrDrtm(
331      const TPMI_DH_PCR        pcrHandle,     // IN: the index of the PCR to be
332                                              //     modified
333      const TPMI_ALG_HASH      hash,          // IN: the bank identifier
334      const TPM2B_DIGEST      *digest         // IN: the digest to modify the PCR
335      )
336  {
337      BYTE        *pcrData = GetPcrPointer(hash, pcrHandle);
338
339      if(pcrData != NULL)
340      {
341          // Rest the PCR to zeros
342          MemorySet(pcrData, 0, digest->t.size);
343
344          // if the TPM has not started, then set the PCR to 0...04 and then extend
345          if(!TPMIsStarted())
346          {
347              pcrData[digest->t.size - 1] = 4;
348          }
349          // Now, extend the value
350          PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
351      }
352  }
```

9.6.3.14 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```
353  void
354  PCRStartup(
355      STARTUP_TYPE     type           // IN: startup type
356      )
357  {
358      UINT32              pcr, j;
359      UINT32              saveIndex = 0;
360
361      g_pcrReConfig = FALSE;
362
363      if(type != SU_RESUME)
364      {
365          // PCR generation counter is cleared at TPM_RESET and TPM_RESTART
366          gr.pcrCounter = 0;
367      }
368
369      // Initialize/Restore PCR values
370      for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
371      {
372          BOOL        incrSaveIndex = FALSE;
373
374          // If this is the H-CRTM PCR and we are not doing a resume and we
375          // had an H-CRTM event, then we don't change this PCR
376          if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
377              continue;
378
379          // Iterate each hash algorithm bank
380          for(j = 0; j < gp.pcrAllocated.count; j++)
381          {
382              TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
383              BYTE            *pcrData = GetPcrPointer(hash, pcr);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
            UINT16          pcrSize = CryptGetHashDigestSize(hash);

            if(pcrData != NULL)
            {
                if(type == SU_RESUME && s_initAttributes[pcr].stateSave == SET)
                {
                    // Restore saved PCR value
                    BYTE    *pcrSavedData;
                    pcrSavedData = GetSavedPcrPointer(
                                        gp.pcrAllocated.pcrSelections[j].hash,
                                        saveIndex);
                    MemoryCopy(pcrData, pcrSavedData, pcrSize, pcrSize);
                    incrSaveIndex = TRUE;
                }
                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(incrSaveIndex == TRUE)
            saveIndex++;
    }

    // Reset authValues
    if(type != SU_RESUME)
    {
        for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
        {
            gc.pcrAuthValues.auth[j].t.size = 0;
        }
    }

}
```

9.6.3.15 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++)
    {
        // Iterate each hash algorithm bank
        for(j = 0; j < gp.pcrAllocated.count; j++)
        {
            BYTE    *pcrData;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
443             UINT32  pcrSize;
444
445             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
446
447             if(pcrData != NULL)
448             {
449                 pcrSize
450                     = CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
451
452                 if(s_initAttributes[pcr].stateSave == SET)
453                 {
454                     // Restore saved PCR value
455                     BYTE    *pcrSavedData;
456                     pcrSavedData
457                         = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
458                                              saveIndex++);
459                     MemoryCopy(pcrSavedData, pcrData, pcrSize, pcrSize);
460                 }
461             }
462         }
463     }
464
465     return;
466 }
```

9.6.3.16 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 69

Return Value	Meaning
TRUE	PCR is state saved
FALSE	PCR is not state saved

```
467 BOOL
468 PCRIsStateSaved(
469     TPMI_DH_PCR      handle          // IN: PCR handle to be extended
470     )
471 {
472     UINT32           pcr = handle - PCR_FIRST;
473
474     if(s_initAttributes[pcr].stateSave == SET)
475         return TRUE;
476     else
477         return FALSE;
478 }
```

9.6.3.17 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 70

Return Value	Meaning
TRUE	TPM2_PCR_Reset() is allowed
FALSE	TPM2_PCR_Reset() is not allowed

```
479  BOOL
480  PCRIsResetAllowed(
481      TPMI_DH_PCR      handle         // IN: PCR handle to be extended
482      )
483  {
484      UINT8               commandLocality;
485      UINT8               localityBits = 1;
486      UINT32              pcr = handle - PCR_FIRST;
487
488      // Check for the locality
489      commandLocality = _plat__LocalityGet();
490
491  #ifdef DRTM_PCR
492      // For a TPM that does DRTM, Reset is not allowed at locality 4
493      if(commandLocality == 4)
494          return FALSE;
495  #endif
496
497      localityBits = localityBits << commandLocality;
498      if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
499          return FALSE;
500      else
501          return TRUE;
502
503  }
```

9.6.3.18 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.

```
504  void
505  PCRChanged(
506      TPM_HANDLE       pcrHandle      // IN: the handle of the PCR that changed.
507      )
508  {
509      // For the reference implementation, the only change that does not cause
510      // increment is a change to a PCR in the TCB group.
511      if(!PCRBelongsTCBGroup(pcrHandle))
512          gr.pcrCounter++;
513  }
```

9.6.3.19 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 71

Return Value	Meaning
TRUE	extend is allowed
FALSE	extend is not allowed

```
514  BOOL
515  PCRIsExtendAllowed(
516      TPMI_DH_PCR      handle         // IN: PCR handle to be extended
517      )
518  {
519      UINT8               commandLocality;
520      UINT8               localityBits = 1;
521      UINT32              pcr = handle - PCR_FIRST;
522
523      // Check for the locality
524      commandLocality = _plat__LocalityGet();
525      localityBits = localityBits << commandLocality;
526      if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
527          return FALSE;
528      else
529          return TRUE;
530
531  }
```

9.6.3.20 PCRExtend()

This function is used to extend a PCR in a specific bank.

```
532  void
533  PCRExtend(
534      TPMI_DH_PCR      handle,         // IN: PCR handle to be extended
535      TPMI_ALG_HASH    hash,           // IN: hash algorithm of PCR
536      UINT32           size,           // IN: size of data to be extended
537      BYTE            *data            // IN: data to be extended
538      )
539  {
540      UINT32               pcr = handle - PCR_FIRST;
541      BYTE                *pcrData;
542      HASH_STATE           hashState;
543      UINT16               pcrSize;
544
545      pcrData = GetPcrPointer(hash, pcr);
546
547      // Extend PCR if it is allocated
548      if(pcrData != NULL)
549      {
550          pcrSize = CryptGetHashDigestSize(hash);
551          CryptStartHash(hash, &hashState);
552          CryptUpdateDigest(&hashState, pcrSize, pcrData);
553          CryptUpdateDigest(&hashState, size, data);
554          CryptCompleteHash(&hashState, pcrSize, pcrData);
555
556          // If PCR does not belong to TCB group, increment PCR counter
557          if(!PCRBelongsTCBGroup(handle))
558              gr.pcrCounter++;
559      }
560
561      return;
562  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.6.3.21 PCRComputeCurrentDigest()

This function computes the digest of the selected PCR.

As a side-effect, *selection* is modified so that only the implemented PCR will have their bits still set.

```
563  void
564  PCRComputeCurrentDigest(
565      TPMI_ALG_HASH        hashAlg,       // IN: hash algorithm to compute digest
566      TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
567                                          //     output)
568      TPM2B_DIGEST        *digest         // OUT: digest
569      )
570  {
571      HASH_STATE               hashState;
572      TPMS_PCR_SELECTION      *select;
573      BYTE                    *pcrData;   // will point to a digest
574      UINT32                   pcrSize;
575      UINT32                   pcr;
576      UINT32                   i;
577
578      // Initialize the hash
579      digest->t.size = CryptStartHash(hashAlg, &hashState);
580      pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
581
582      // Iterate through the list of PCR selection structures
583      for(i = 0; i < selection->count; i++)
584      {
585          // Point to the current selection
586          select = &selection->pcrSelections[i]; // Point to the current selection
587          FilterPcr(select);      // Clear out the bits for unimplemented PCR
588
589          // Need the size of each digest
590          pcrSize = CryptGetHashDigestSize(selection->pcrSelections[i].hash);
591
592          // Iterate through the selection
593          for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
594          {
595              if(IsPcrSelected(pcr, select))         // Is this PCR selected
596              {
597                  // Get pointer to the digest data for the bank
598                  pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
599                  pAssert(pcrData != NULL);
600                  CryptUpdateDigest(&hashState, pcrSize, pcrData);  // add to digest
601              }
602          }
603      }
604      // Complete hash stack
605      CryptCompleteHash2B(&hashState, &digest->b);
606
607      return;
608  }
```

9.6.3.22 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.

```
609  void
610  PCRRead(
611      TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
612                                          //     output)
613      TPML_DIGEST         *digest,        // OUT: digest
614      UINT32              *pcrCounter     // OUT: the current value of PCR generation
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
                                        //     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

        // Iterate through the selection
        for (pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
        {
            if(IsPcrSelected(pcr, select))         // Is this PCR selected
            {
                // Check if number of digest exceed upper bound
                if(digest->count > 7)
                {
                    // Clear rest of the current select bitmap
                    while(   pcr < IMPLEMENTATION_PCR
                             // do not round up!
                          && (pcr / 8) < select->sizeofSelect)
                    {
                        // do not round up!
                        select->pcrSelect[pcr/8] &= (BYTE) ~(1 << (pcr % 8));
                        pcr++;
                    }
                    // Exit inner loop
                    break;;
                }
                // Need the size of each digest
                digest->digests[digest->count].t.size =
                    CryptGetHashDigestSize(selection->pcrSelections[i].hash);

                // Get pointer to the digest data for the bank
                pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
                pAssert(pcrData != NULL);
                // Add to the data to digest
                MemoryCopy(digest->digests[digest->count].t.buffer,
                           pcrData,
                           digest->digests[digest->count].t.size,
                           digest->digests[digest->count].t.size);
                digest->count++;
            }
        }
        // If we exit inner loop because we have exceed the output upper bound
        if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
        {
            // Clear rest of the selection
            while(i < selection->count)
            {
                MemorySet(selection->pcrSelections[i].pcrSelect, 0,
                          selection->pcrSelections[i].sizeofSelect);
                i++;
            }
            // exit outer loop
            break;
        }
    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
681
682         *pcrCounter = gr.pcrCounter;
683
684         return;
685     }
```

9.6.3.23 PcrWrite()

This function is used by _TPM_Hash_End() to set a PCR to the computed hash of the H-CRTM event.

```
686     void
687     PcrWrite(
688         TPMI_DH_PCR      handle,        // IN: PCR handle to be extended
689         TPMI_ALG_HASH    hash,          // IN: hash algorithm of PCR
690         TPM2B_DIGEST    *digest         // IN: the new value
691         )
692     {
693         UINT32              pcr = handle - PCR_FIRST;
694         BYTE               *pcrData;
695
696         // Copy value to the PCR if it is allocated
697         pcrData = GetPcrPointer(hash, pcr);
698         if(pcrData != NULL)
699         {
700             MemoryCopy(pcrData, digest->t.buffer, digest->t.size, digest->t.size); ;
701         }
702
703         return;
704     }
```

9.6.3.24 PCRAllocate()

This function is used to change the PCR allocation.

Table 72

Error Returns	Meaning
TPM_RC_SUCCESS	allocate success
TPM_RC_NO_RESULTS	allocate failed
TPM_RC_PCR	improper allocation

```
705     TPM_RC
706     PCRAllocate(
707         TPML_PCR_SELECTION  *allocate,      // IN: required allocation
708         UINT32              *maxPCR,        // OUT: Maximum number of PCR
709         UINT32              *sizeNeeded,    // OUT: required space
710         UINT32              *sizeAvailable  // OUT: available space
711         )
712     {
713         UINT32                  i, j, k;
714         TPML_PCR_SELECTION      newAllocate;
715         // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
716         BOOL                    pcrHcrtm = FALSE;
717         BOOL                    pcrDrtm = FALSE;
718
719         // Create the expected new PCR allocation based on the existing allocation
720         // and the new input:
721         //  1. if a PCR bank does not appear in the new allocation, the existing
722         //     allocation of this PCR bank will be preserved.
723         //  2. if a PCR bank appears multiple times in the new allocation, only the
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
724         //     last one will be in effect.
725         newAllocate = gp.pcrAllocated;
726         for(i = 0; i < allocate->count; i++)
727         {
728             for(j = 0; j < newAllocate.count; j++)
729             {
730                 // If hash matches, the new allocation covers the old allocation
731                 // for this particular bank.
732                 // The assumption is the initial PCR allocation (from manufacture)
733                 // has all the supported hash algorithms with an assigned bank
734                 // (possibly empty).  So there must be a match for any new bank
735                 // allocation from the input.
736                 if(newAllocate.pcrSelections[j].hash ==
737                     allocate->pcrSelections[i].hash)
738                 {
739                     newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
740                     break;
741                 }
742             }
743             // The j loop must exit with a match.
744             pAssert(j < newAllocate.count);
745         }
746
747         // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
748         *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
749         if(*maxPCR > IMPLEMENTATION_PCR)
750             *maxPCR = IMPLEMENTATION_PCR;
751
752         // Compute required size for allocation
753         *sizeNeeded = 0;
754         for(i = 0; i < newAllocate.count; i++)
755         {
756             UINT32      digestSize
757                 = CryptGetHashDigestSize(newAllocate.pcrSelections[i].hash);
758 #if defined(DRTM_PCR)
759             // Make sure that we end up with at least one DRTM PCR
760 #   define PCR_DRTM  (PCR_FIRST + DRTM_PCR)    // for cosmetics
761             pcrDrtm =    pcrDrtm || TEST_BIT(PCR_DRTM, newAllocate.pcrSelections[i]);
762 #else   // if DRTM PCR is not required, indicate that the allocation is OK
763             pcrDrtm = TRUE;
764 #endif
765
766 #if defined(HCRTM_PCR)
767             // and one HCRTM PCR (since this is usually PCR 0...)
768 #   define PCR_HCRTM (PCR_FIRST + HCRTM_PCR)
769             pcrHcrtm =    pcrDrtm || TEST_BIT(PCR_HCRTM, newAllocate.pcrSelections[i]);
770 #else
771             pcrHcrtm = TRUE;
772 #endif
773             for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
774             {
775                 BYTE        mask = 1;
776                 for(k = 0; k < 8; k++)
777                 {
778                     if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
779                         *sizeNeeded += digestSize;
780                     mask = mask << 1;
781                 }
782             }
783         }
784
785         if(!pcrDrtm || !pcrHcrtm)
786             return TPM_RC_PCR;
787
788
789         // In this particular implementation, we always have enough space to
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
790         // allocate PCR.  Different implementation may return a sizeAvailable less
791         // than the sizeNeed.
792         *sizeAvailable = sizeof(s_pcrs);
793
794         // Save the required allocation to NV.  Note that after NV is written, the
795         // PCR allocation in NV is no longer consistent with the RAM data
796         // gp.pcrAllocated.  The NV version reflect the allocate after next
797         // TPM_RESET, while the RAM version reflects the current allocation
798         NvWriteReserved(NV_PCR_ALLOCATED, &newAllocate);
799
800         return TPM_RC_SUCCESS;
801
802     }
```

9.6.3.25 PCRSetValue()

This function is used to set the designated PCR in all banks to an initial value. The initial value is signed and will be sign extended into the entire PCR.

```
803     void
804     PCRSetValue(
805         TPM_HANDLE       handle,        // IN: the handle of the PCR to set
806         INT8             initialValue   // IN: the value to set
807         )
808     {
809         int              i;
810         UINT32           pcr = handle - PCR_FIRST;
811         TPMI_ALG_HASH    hash;
812         UINT16           digestSize;
813         BYTE            *pcrData;
814
815         // Iterate supported PCR bank algorithms to reset
816         for(i = 0; i < HASH_COUNT; i++)
817         {
818             hash = CryptGetHashAlgByIndex(i);
819             // Prevent runaway
820             if(hash == TPM_ALG_NULL)
821                 break;
822
823             // Get a pointer to the data
824             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
825
826             // If the PCR is allocated
827             if(pcrData != NULL)
828             {
829                 // And the size of the digest
830                 digestSize = CryptGetHashDigestSize(hash);
831
832                 // Set the LSO to the input value
833                 pcrData[digestSize - 1] = initialValue;
834
835                 // Sign extend
836                 if(initialValue >= 0)
837                     MemorySet(pcrData, 0, digestSize - 1);
838                 else
839                     MemorySet(pcrData, -1, digestSize - 1);
840             }
841         }
842     }
```

9.6.3.26 PCRResetDynamics

This function is used to reset a dynamic PCR to 0. This function is used in DRTM sequence.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
void
PCRResetDynamics(
    void
    )
{
    UINT32              pcr, i;

    // Initialize PCR values
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        // Iterate each hash algorithm bank
        for(i = 0; i < gp.pcrAllocated.count; i++)
        {
            BYTE    *pcrData;
            UINT32  pcrSize;

            pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);

            if(pcrData != NULL)
            {
                pcrSize =
                    CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[i].hash);

                // Reset PCR
                // Any PCR can be reset by locality 4 should be reset to 0
                if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
                    MemorySet(pcrData, 0, pcrSize);
            }
        }
    }
    return;
}
```

9.6.3.27 PCRCapGetAllocation()

This function is used to get the current allocation of PCR banks.

Table 73

Return Value	Meaning
YES:	if the return count is 0
NO:	if the return count is not 0

```
TPMI_YES_NO
PCRCapGetAllocation(
    UINT32              count,          // IN: count of return
    TPML_PCR_SELECTION  *pcrSelection   // OUT: PCR allocation list
    )
{
    if(count == 0)
    {
        pcrSelection->count = 0;
        return YES;
    }
    else
    {
        *pcrSelection = gp.pcrAllocated;
        return NO;
    }
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.6.3.28 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```
892  static void
893  PCRSetSelectBit(
894      UINT32           pcr,           // IN: PCR number
895      BYTE            *bitmap         // OUT: bit map to be set
896      )
897  {
898      bitmap[pcr / 8] |= (1 << (pcr % 8));
899      return;
900  }
```

9.6.3.29 PCRGetProperty()

This function returns the selected PCR property.

Table 74

Return Value	Meaning
TRUE	the property type is implemented
FALSE	the property type is not implemented

```
901  static BOOL
902  PCRGetProperty(
903      TPM_PT_PCR               property,
904      TPMS_TAGGED_PCR_SELECT  *select
905      )
906  {
907      UINT32              pcr;
908      UINT32              groupIndex;
909
910      select->tag = property;
911      // Always set the bitmap to be the size of all PCR
912      select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
913
914      // Initialize bitmap
915      MemorySet(select->pcrSelect, 0, select->sizeofSelect);
916
917      // Collecting properties
918      for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
919      {
920          switch(property)
921          {
922              case TPM_PT_PCR_SAVE:
923                  if(s_initAttributes[pcr].stateSave == SET)
924                      PCRSetSelectBit(pcr, select->pcrSelect);
925                  break;
926              case TPM_PT_PCR_EXTEND_L0:
927                  if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
928                      PCRSetSelectBit(pcr, select->pcrSelect);
929                  break;
930              case TPM_PT_PCR_RESET_L0:
931                  if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
932                      PCRSetSelectBit(pcr, select->pcrSelect);
933                  break;
934              case TPM_PT_PCR_EXTEND_L1:
935                  if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
936                      PCRSetSelectBit(pcr, select->pcrSelect);
937                  break;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
938                 case TPM_PT_PCR_RESET_L1:
939                     if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
940                         PCRSetSelectBit(pcr, select->pcrSelect);
941                     break;
942                 case TPM_PT_PCR_EXTEND_L2:
943                     if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
944                         PCRSetSelectBit(pcr, select->pcrSelect);
945                     break;
946                 case TPM_PT_PCR_RESET_L2:
947                     if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
948                         PCRSetSelectBit(pcr, select->pcrSelect);
949                     break;
950                 case TPM_PT_PCR_EXTEND_L3:
951                     if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
952                         PCRSetSelectBit(pcr, select->pcrSelect);
953                     break;
954                 case TPM_PT_PCR_RESET_L3:
955                     if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
956                         PCRSetSelectBit(pcr, select->pcrSelect);
957                     break;
958                 case TPM_PT_PCR_EXTEND_L4:
959                     if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
960                         PCRSetSelectBit(pcr, select->pcrSelect);
961                     break;
962                 case TPM_PT_PCR_RESET_L4:
963                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
964                         PCRSetSelectBit(pcr, select->pcrSelect);
965                     break;
966                 case TPM_PT_PCR_DRTM_RESET:
967                     // DRTM reset PCRs are the PCR reset by locality 4
968                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
969                         PCRSetSelectBit(pcr, select->pcrSelect);
970                     break;
971 #if NUM_POLICY_PCR_GROUP > 0
972                 case TPM_PT_PCR_POLICY:
973                     if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
974                         PCRSetSelectBit(pcr, select->pcrSelect);
975                     break;
976 #endif
977 #if NUM_AUTHVALUE_PCR_GROUP > 0
978                 case TPM_PT_PCR_AUTH:
979                     if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
980                         PCRSetSelectBit(pcr, select->pcrSelect);
981                     break;
982 #endif
983 #if ENABLE_PCR_NO_INCREMENT == YES
984                 case TPM_PT_PCR_NO_INCREMENT:
985                     if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
986                         PCRSetSelectBit(pcr, select->pcrSelect);
987                     break;
988 #endif
989                 default:
990                     // If property is not supported, stop scanning PCR attributes
991                     // and return.
992                     return FALSE;
993                     break;
994             }
995         }
996         return TRUE;
997 }
```

9.6.3.30 PCRCapGetProperties()

This function returns a list of PCR properties starting at *property*.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

Table 75

Return Value	Meaning
YES:	if no more property is available
NO:	if there are more properties not reported

```
TPMI_YES_NO
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
    )
{
    TPMI_YES_NO     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;

    // TPM_PT_PCR_FIRST is defined as 0 in spec.  It ensures that property
    // value would never be less than TPM_PT_PCR_FIRST
    pAssert(TPM_PT_PCR_FIRST == 0);

    // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
    // implemented on the TPM.
    for(i = property; i <= TPM_PT_PCR_LAST; i++)
    {
        if(select->count < count)
        {
            // If we have not filled up the return list, add more properties to it
            if(PCRGetProperty(i, &select->pcrProperty[select->count]))
                // only increment if the property is implemented
            select->count++;
        }
        else
        {
            // If the return list is full but we still have properties
            // available, report this and stop iterating.
            more = YES;
            break;
        }
    }
    return more;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

9.6.3.31 PCRCapGetHandles()

This function is used to get a list of handles of PCR, started from *handle*. If *handle* exceeds the maximum PCR handle range, an empty list will be returned and the return value will be NO.

Table 76

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
TPMI_YES_NO
```

```
1040    PCRCapGetHandles(
1041        TPMI_DH_PCR      handle,        // IN: start handle
1042        UINT32           count,         // IN: count of returned handles
1043        TPML_HANDLE     *handleList     // OUT: list of handle
1044        )
1045    {
1046        TPMI_YES_NO     more = NO;
1047        UINT32          i;
1048
1049        pAssert(HandleGetType(handle) == TPM_HT_PCR);
1050
1051        // Initialize output handle list
1052        handleList->count = 0;
1053
1054        // The maximum count of handles we may return is MAX_CAP_HANDLES
1055        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1056
1057        // Iterate PCR handle range
1058        for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1059        {
1060            if(handleList->count < count)
1061            {
1062                // If we have not filled up the return list, add this PCR
1063                // handle to it
1064                handleList->handle[handleList->count] = i + PCR_FIRST;
1065                handleList->count++;
1066            }
1067            else
1068            {
1069                // If the return list is full but we still have PCR handle
1070                // available, report this and stop iterating
1071                more = YES;
1072                break;
1073            }
1074        }
1075        return more;
1076    }
```

9.7 PP.c

9.7.1 Introduction

This file contains the functions that support the physical presence operations of the TPM.

9.7.2 Includes

```
1   #include "InternalRoutines.h"
```

9.7.3 Functions

9.7.3.1 PhysicalPresencePreInstall_Init()

This function is used to initialize the array of commands that 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.

```
2   void
3   PhysicalPresencePreInstall_Init(
4       void
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
    )
{
    // Clear all the PP commands
    MemorySet(&gp.ppList, 0,
              ((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7) / 8);

    // TPM_CC_PP_Commands always requires PP
    if(CommandIsImplemented(TPM_CC_PP_Commands))
        PhysicalPresenceCommandSet(TPM_CC_PP_Commands);

    // Write PP list to NV
    NvWriteReserved(NV_PP_LIST, &gp.ppList);

    return;
}
```

9.7.3.2 PhysicalPresenceCommandSet()

This function is used to indicate a command that requires PP confirmation.

```
void
PhysicalPresenceCommandSet(
    TPM_CC          commandCode     // IN: command code
    )
{
    UINT32      bitPos;

    // Assume command is implemented.  It should be checked before this
    // function is called
    pAssert(CommandIsImplemented(commandCode));

    // If the command is not a PP command, ignore it
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return;

    bitPos = commandCode - TPM_CC_PP_FIRST;

    // Set bit
    gp.ppList[bitPos/8] |= 1 << (bitPos % 8);

    return;
}
```

9.7.3.3 PhysicalPresenceCommandClear()

This function is used to indicate a command that no longer requires PP confirmation.

```
void
PhysicalPresenceCommandClear(
    TPM_CC          commandCode     // IN: command code
    )
{
    UINT32      bitPos;

    // Assume command is implemented.  It should be checked before this
    // function is called
    pAssert(CommandIsImplemented(commandCode));

    // If the command is not a PP command, ignore it
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return;

    // if the input code is TPM_CC_PP_Commands, it can not be cleared
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
58      if(commandCode == TPM_CC_PP_Commands)
59          return;
60
61      bitPos = commandCode - TPM_CC_PP_FIRST;
62
63      // Set bit
64      gp.ppList[bitPos/8] |= (1 << (bitPos % 8));
65      // Flip it to off
66      gp.ppList[bitPos/8] ^= (1 << (bitPos % 8));
67
68      return;
69  }
```

9.7.3.4 PhysicalPresenceIsRequired()

This function indicates if PP confirmation is required for a command.

Table 77

Return Value	Meaning
TRUE	if physical presence is required
FALSE	if physical presence is not required

```
70  BOOL
71  PhysicalPresenceIsRequired(
72      TPM_CC           commandCode    // IN: command code
73      )
74  {
75      UINT32      bitPos;
76
77      // if the input commandCode is not a PP command, return FALSE
78      if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
79          return FALSE;
80
81      bitPos = commandCode - TPM_CC_PP_FIRST;
82
83      // Check the bit map.  If the bit is SET, PP authorization is required
84      return ((gp.ppList[bitPos/8] & (1 << (bitPos % 8))) != 0);
85
86  }
```

9.7.3.5 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 78

Return Value	Meaning
YES	if there are more command codes available
NO	all the available command codes have been returned

```
87  TPMI_YES_NO
88  PhysicalPresenceCapGetCCList(
89      TPM_CC           commandCode,   // IN: start command code
90      UINT32           count,         // IN: count of returned TPM_CC
91      TPML_CC         *commandList    // OUT: list of TPM_CC
92      )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    // 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(i = commandCode; i <= TPM_CC_PP_LAST; i++)
    {
        if(PhysicalPresenceIsRequired(i))
        {
            if(commandList->count < count)
            {
                // If we have not filled up the return list, add this command
                // code to it
                commandList->commandCodes[commandList->count] = i;
                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;
}
```

9.8 Session.c

9.8.1 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 TMP), 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

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_GetCapabiltiy(), 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*.

9.8.2 Includes, Defines, and Local Variables

```
1   #define SESSION_C
2   #include "InternalRoutines.h"
3   #include "Platform.h"
4   #include "SessionProcess_fp.h"
```

9.8.3 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*.

EXAMPLE 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.

```
5   static void
6   ContextIdSetOldest(
7       void
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
    )
{
    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.
}
```

9.8.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```
void
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.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
67              if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
68                  gr.contextArray[i] = 0;
69          }
70          // Find the oldest session in context ID data and set it in
71          // s_oldestSavedSession
72          ContextIdSetOldest();
73      }
74      else
75      {
76          // For STARTUP_CLEAR, clear out the contextArray
77          for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
78              gr.contextArray[i] = 0;
79
80          // reset the context counter
81          gr.contextCounter = MAX_LOADED_SESSIONS + 1;
82
83          // Initialize oldest saved session
84          s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
85      }
86      return;
87  }
```

9.8.5 Access Functions

9.8.5.1 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 79

Return Value	Meaning
TRUE	if session is loaded
FALSE	if it is not loaded

```
88  BOOL
89  SessionIsLoaded(
90      TPM_HANDLE          handle          // IN: session handle
91      )
92  {
93      pAssert(    HandleGetType(handle) == TPM_HT_POLICY_SESSION
94              || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
95
96      handle = handle & HR_HANDLE_MASK;
97
98      // if out of range of possible active session, or not assigned to a loaded
99      // session return false
100     if(   handle >= MAX_ACTIVE_SESSIONS
101        || gr.contextArray[handle] == 0
102        || gr.contextArray[handle] > MAX_LOADED_SESSIONS
103       )
104         return FALSE;
105
106     return TRUE;
107 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

9.8.5.2 SessionIsSaved()

This function test a session handle references a saved session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE An password authorization does not have a session.

This function requires that the handle be a valid session handle.

Table 80

Return Value	Meaning
TRUE	if session is saved
FALSE	if it is not saved

```
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;
}
```

9.8.5.3 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 81

Return Value	Meaning
TRUE	if PCR value is current
FALSE	if PCR value is not current

```
BOOL
SessionPCRValueIsCurrent(
    TPMI_SH_POLICY   handle         // IN: session handle
    )
{
    SESSION             *session;

    pAssert(SessionIsLoaded(handle));

    session = SessionGet(handle);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
137         if(   session->pcrCounter != 0
138            && session->pcrCounter != gr.pcrCounter
139           )
140             return FALSE;
141         else
142             return TRUE;
143     }
```

9.8.5.4 SessionGet()

This function returns a pointer to the session object associated with a session handle.

The function requires that the session is loaded.

```
144     SESSION *
145     SessionGet(
146         TPM_HANDLE       handle         // IN: session handle
147         )
148     {
149         CONTEXT_SLOT    sessionIndex;
150
151         pAssert(   HandleGetType(handle) == TPM_HT_POLICY_SESSION
152                 || HandleGetType(handle) == TPM_HT_HMAC_SESSION
153                );
154
155         pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);
156
157         // get the contents of the session array.  Because session is loaded, we
158         // should always get a valid sessionIndex
159         sessionIndex = gr.contextArray[handle & HR_HANDLE_MASK] - 1;
160
161         pAssert(sessionIndex < MAX_LOADED_SESSIONS);
162
163         return &s_sessions[sessionIndex].session;
164     }
```

9.8.6 Utility Functions

9.8.6.1 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 82

return type	TPM_RC
TPM_RC_SUCCESS	context ID was assigned
TPM_RC_CONTEXT_GAP	can't assign a new *contextID* until the oldest saved session context is recycled
TPM_RC_SESSION_HANDLE	there is no slot available in the context array for tracking of this session context

```
165     static TPM_RC
166     ContextIdSessionCreate (
167         TPM_HANDLE      *handle,        // OUT: receives the assigned handle. This will
168                                         //     be an index that must be adjusted by the
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
169                                     //     caller according to the type of the
170                                     //     session created
171     UINT32          sessionIndex    // IN: The session context array entry that will
172                                     //     be occupied by the created session
173     )
174 {
175
176     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
177
178     // check to see if creating the context is safe
179     // Is this going to be an assignment for the last session context
180     // array entry?  If so, then there will be no room to recycle the
181     // oldest context if needed.  If the gap is not at maximum, then
182     // it will be possible to save a context if it becomes necessary.
183     if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
184        && s_freeSessionSlots == 1)
185     {
186         // See if the gap is at maximum
187         if(   (CONTEXT_SLOT)gr.contextCounter
188            == gr.contextArray[s_oldestSavedSession])
189
190             // Note: if this is being used on a TPM.combined, this return
191             //       code should be transformed to an appropriate
192             //       ISO/IEC 11889 (first edition) error code for this case.
193             return TPM_RC_CONTEXT_GAP;
194     }
195
196     // Find an unoccupied entry in the contextArray
197     for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
198     {
199         if(gr.contextArray[*handle] == 0)
200         {
201             // indicate that the session associated with this handle
202             // references a loaded session
203             gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex+1);
204             return TPM_RC_SUCCESS;
205         }
206     }
207     return TPM_RC_SESSION_HANDLES;
208 }
```

9.8.6.2 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 83

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	need to recycle sessions
TPM_RC_SESSION_HANDLE	active session space is full
TPM_RC_SESSION_MEMORY	loaded session space is full

```
209 TPM_RC
210 SessionCreate(
211     TPM_SE          sessionType,    // IN: the session type
212     TPMI_ALG_HASH    authHash,      // IN: the hash algorithm
213     TPM2B_NONCE     *nonceCaller,   // IN: initial nonceCaller
214     TPMT_SYM_DEF    *symmetric,     // IN: the symmetric algorithm
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
215         TPMI_DH_ENTITY   bind,          // IN: the bind object
216         TPM2B_DATA      *seed,          // IN: seed data
217         TPM_HANDLE      *sessionHandle  // OUT: the session handle
218         )
219     {
220         TPM_RC               result = TPM_RC_SUCCESS;
221         CONTEXT_SLOT         slotIndex;
222         SESSION             *session = NULL;
223
224         pAssert(   sessionType == TPM_SE_HMAC
225                 || sessionType == TPM_SE_POLICY
226                 || sessionType == TPM_SE_TRIAL);
227
228         // If there are no open spots in the session array, then no point in searching
229         if(s_freeSessionSlots == 0)
230             return TPM_RC_SESSION_MEMORY;
231
232         // Find a space for loading a session
233         for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
234         {
235             // Is this available?
236             if(s_sessions[slotIndex].occupied == FALSE)
237             {
238                 session = &s_sessions[slotIndex].session;
239                 break;
240             }
241         }
242         // if no spot found, then this is an internal error
243         pAssert (slotIndex < MAX_LOADED_SESSIONS);
244
245         // Call context ID function to get a handle. TPM_RC_SESSION_HANDLE may be
246         // returned from ContextIdHandelAssign()
247         result = ContextIdSessionCreate(sessionHandle, slotIndex);
248         if(result != TPM_RC_SUCCESS)
249             return result;
250
251         //*** Only return from this point on is TPM_RC_SUCCESS
252
253         // Can now indicate that the session array entry is occupied.
254         s_freeSessionSlots--;
255         s_sessions[slotIndex].occupied = TRUE;
256
257         // Initialize the session data
258         MemorySet(session, 0, sizeof(SESSION));
259
260         // Initialize internal session data
261         session->authHashAlg = authHash;
262         // Initialize session type
263         if(sessionType == TPM_SE_HMAC)
264         {
265             *sessionHandle += HMAC_SESSION_FIRST;
266
267         }
268         else
269         {
270             *sessionHandle += POLICY_SESSION_FIRST;
271
272             // For TPM_SE_POLICY or TPM_SE_TRIAL
273             session->attributes.isPolicy = SET;
274             if(sessionType == TPM_SE_TRIAL)
275                 session->attributes.isTrialPolicy = SET;
276
277             // Initialize policy session data
278             SessionInitPolicyData(session);
279         }
280         // Create initial session nonce
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
281         session->nonceTPM.t.size = nonceCaller->t.size;
282         CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
283 
284         // Set up session parameter encryption algorithm
285         session->symmetric = *symmetric;
286 
287         // If there is a bind object or a session secret, then need to compute
288         // a sessionKey.
289         if(bind != TPM_RH_NULL || seed->t.size != 0)
290         {
291             // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
292             //                       nonceCaller, bits)
293             // The HMAC key for generating the sessionSecret can be the concatenation
294             // of an authorization value and a seed value
295             // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
296             // label values.
297 
298             TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
299             TPM2B_KEY            key;
300 
301             UINT16               hashSize;      // The size of the hash used by the
302                                                 // session crated by this command
303             TPM2B_AUTH  entityAuth;             // The authValue of the entity
304                                                 // associated with HMAC session
305 
306             // Get hash size, which is also the length of sessionKey
307             hashSize = CryptGetHashDigestSize(session->authHashAlg);
308 
309             // Get authValue of associated entity
310             entityAuth.t.size = EntityGetAuthValue(bind, &entityAuth.t.buffer);
311 
312             // Concatenate authValue and seed
313             pAssert(entityAuth.t.size + seed->t.size <= <K>sizeof(key.t.buffer));
314             MemoryCopy2B(&key.b, &entityAuth.b, sizeof(key.t.buffer));
315             MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));
316 
317             session->sessionKey.t.size = hashSize;
318 
319             // Compute the session key
320             // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
321             // label values.
322             KDFa(session->authHashAlg, &key.b, "ATH", &session->nonceTPM.b,
323                  &nonceCaller->b, hashSize * 8, session->sessionKey.t.buffer, NULL);
324         }
325 
326         // Copy the name of the entity that the HMAC session is bound to
327         // Policy session is not bound to an entity
328         if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
329         {
330             session->attributes.isBound = SET;
331             SessionComputeBoundEntity(bind, &session->u1.boundEntity);
332         }
333         // If there is a bind object and it is subject to DA, then use of this session
334         // is subject to DA regardless of how it is used.
335         session->attributes.isDaBound =    (bind != TPM_RH_NULL)
336                                         && (IsDAExempted(bind) == FALSE);
337 
338         // If the session is bound, then check to see if it is bound to lockoutAuth
339         session->attributes.isLockoutBound =    (session->attributes.isDaBound  == SET)
340                                              && (bind == TPM_RH_LOCKOUT);
341         return TPM_RC_SUCCESS;
342 
343     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.8.6.3 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 84

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	a *contextID* could not be assigned.
TPM_RC_TOO_MANY_CONTEXTS	the counter maxed out

```
344  TPM_RC
345  SessionContextSave (
346      TPM_HANDLE           handle,        // IN: session handle
347      CONTEXT_COUNTER     *contextID      // OUT: assigned contextID
348      )
349  {
350      UINT32                      contextIndex;
351      CONTEXT_SLOT                slotIndex;
352
353      pAssert(SessionIsLoaded(handle));
354
355      // check to see if the gap is already maxed out
356      // Need to have a saved session
357      if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
358            // if the oldest saved session has the same value as the low bits
359            // of the contextCounter, then the GAP is maxed out.
360         && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
361          return TPM_RC_CONTEXT_GAP;
362
363      // if the caller wants the context counter, set it
364      if(contextID != NULL)
365          *contextID = gr.contextCounter;
366
367      pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);
368
369      contextIndex = handle & HR_HANDLE_MASK;
370
371      // Extract the session slot number referenced by the contextArray
372      // because we are going to overwrite this with the low order
373      // contextID value.
374      slotIndex = gr.contextArray[contextIndex] - 1;
375
376      // Set the contextID for the contextArray
377      gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
378
379      // Increment the counter
380      gr.contextCounter++;
381
382      // In the unlikely event that the 64-bit context counter rolls over...
383      if(gr.contextCounter == 0)
384      {
385          // back it up
386          gr.contextCounter--;
387          // return an error
388          return TPM_RC_TOO_MANY_CONTEXTS;
389      }
390      // if the low-order bits wrapped, need to advance the value to skip over
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
391         // the values used to indicate that a session is loaded
392         if(((CONTEXT_SLOT)gr.contextCounter) == 0)
393             gr.contextCounter += MAX_LOADED_SESSIONS + 1;
394
395         // If no other sessions are saved, this is now the oldest.
396         if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
397             s_oldestSavedSession = contextIndex;
398
399         // Mark the session slot as unoccupied
400         s_sessions[slotIndex].occupied = FALSE;
401
402         // and indicate that there is an additional open slot
403         s_freeSessionSlots++;
404
405         return TPM_RC_SUCCESS;
406     }
```

9.8.6.4 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.

Table 85

Error Returns	Meaning
TPM_RC_SESSION_MEMORY	no free session slots
TPM_RC_CONTEXT_GAP	the gap count is maximum and this is not the oldest saved context

```
407     TPM_RC
408     SessionContextLoad(
409         SESSION         *session,       // IN: session structure from saved context
410         TPM_HANDLE      *handle         // IN/OUT: session handle
411         )
412     {
413         UINT32              contextIndex;
414         CONTEXT_SLOT        slotIndex;
415
416         pAssert(   HandleGetType(*handle) == TPM_HT_POLICY_SESSION
417                 || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
418
419         // Don't bother looking if no openings
420         if(s_freeSessionSlots == 0)
421             return TPM_RC_SESSION_MEMORY;
422
423         // Find a free session slot to load the session
424         for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
425             if(s_sessions[slotIndex].occupied == FALSE) break;
426
427         // if no spot found, then this is an internal error
428         pAssert (slotIndex < MAX_LOADED_SESSIONS);
429
430         contextIndex = *handle & HR_HANDLE_MASK;   // extract the index
431
432         // If there is only one slot left, and the gap is at maximum, the only session
433         // context that we can safely load is the oldest one.
434         if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
435            && s_freeSessionSlots == 1
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
436         && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
437         && contextIndex != s_oldestSavedSession
438        )
439          return TPM_RC_CONTEXT_GAP;
440
441     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
442
443     // set the contextArray value to point to the session slot where
444     // the context is loaded
445     gr.contextArray[contextIndex] = slotIndex + 1;
446
447     // if this was the oldest context, find the new oldest
448     if(contextIndex == s_oldestSavedSession)
449         ContextIdSetOldest();
450
451     // Copy session data to session slot
452     s_sessions[slotIndex].session = *session;
453
454     // Set session slot as occupied
455     s_sessions[slotIndex].occupied = TRUE;
456
457     // Reduce the number of open spots
458     s_freeSessionSlots--;
459
460     return TPM_RC_SUCCESS;
461 }
```

9.8.6.5 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.

```
462 void
463 SessionFlush(
464     TPM_HANDLE       handle          // IN: loaded or saved session handle
465     )
466 {
467     CONTEXT_SLOT         slotIndex;
468     UINT32               contextIndex;   // Index into contextArray
469
470     pAssert(   (   HandleGetType(handle) == TPM_HT_POLICY_SESSION
471                 || HandleGetType(handle) == TPM_HT_HMAC_SESSION
472                )
473             && (SessionIsLoaded(handle)  || SessionIsSaved(handle))
474            );
475
476     // Flush context ID of this session
477     // Convert handle to an index into the contextArray
478     contextIndex = handle & HR_HANDLE_MASK;
479
480     pAssert(contextIndex < <K>sizeof(gr.contextArray)/sizeof(gr.contextArray[0]));
481
482     // Get the current contents of the array
483     slotIndex = gr.contextArray[contextIndex];
484
485     // Mark context array entry as available
486     gr.contextArray[contextIndex] = 0;
487
488     // Is this a saved session being flushed
489     if(slotIndex > MAX_LOADED_SESSIONS)
490     {
491         // Flushing the oldest session?
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
492             if(contextIndex == s_oldestSavedSession)
493                 // If so, find a new value for oldest.
494                 ContextIdSetOldest();
495         }
496         else
497         {
498             // Adjust slot index to point to session array index
499             slotIndex -= 1;
500 
501             // Free session array index
502             s_sessions[slotIndex].occupied = FALSE;
503             s_freeSessionSlots++;
504         }
505 
506         return;
507     }
```

9.8.6.6 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.

```
508     void
509     SessionComputeBoundEntity(
510         TPMI_DH_ENTITY   entityHandle,  // IN: handle of entity
511         TPM2B_NAME      *bind           // OUT: binding value
512         )
513     {
514         TPM2B_AUTH          auth;
515         INT16               overlap;
516 
517         // Get name
518         bind->t.size = EntityGetName(entityHandle, &bind->t.name);
519 
520     //    // The bound value of a reserved handle is the handle itself
521     //    if(bind->t.size == sizeof(TPM_HANDLE)) return;
522 
523         // For all the other entities, concatenate the auth value to the name.
524         // Get a local copy of the auth value because some overlapping
525         // may be necessary.
526         auth.t.size = EntityGetAuthValue(entityHandle, &auth.t.buffer);
527         pAssert(auth.t.size <= <K>sizeof(TPMU_HA));
528 
529         // Figure out if there will be any overlap
530         overlap = bind->t.size + auth.t.size - sizeof(bind->t.name);
531 
532         // There is overlap if the combined sizes are greater than will fit
533         if(overlap > 0)
534         {
535             // The overlap area is at the end of the Name
536             BYTE    *result = &bind->t.name[bind->t.size - overlap];
537             int     i;
538 
539             // XOR the auth value into the Name for the overlap area
540             for(i = 0; i < overlap; i++)
541                 result[i] ^= auth.t.buffer[i];
542         }
543         else
544         {
545             // There is no overlap
546             overlap = 0;
547         }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
548         //copy the remainder of the authData to the end of the name
549         MemoryCopy(&bind->t.name[bind->t.size], &auth.t.buffer[overlap],
550                    auth.t.size - overlap, sizeof(bind->t.name) - bind->t.size);
551
552         // Increase the size of the bind data by the size of the auth - the overlap
553         bind->t.size += auth.t.size-overlap;
554
555         return;
556     }
```

9.8.6.7 SessionInitPolicyData()

This function initializes the portions of the session policy data that are not set by the allocation of a session.

```
557     void
558     SessionInitPolicyData(
559         SESSION         *session        // IN: session handle
560         )
561     {
562         // Initialize start time
563         session->startTime = go.clock;
564
565         // Initialize policyDigest.  policyDigest is initialized with a string of 0 of
566         // session algorithm digest size. Since the policy already contains all zeros
567         // it is only necessary to set the size
568         session->u2.policyDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
569         return;
570     }
```

9.8.6.8 SessionResetPolicyData()

This function is used to reset the policy data without changing the nonce or the start time of the session.

```
571     void
572     SessionResetPolicyData(
573         SESSION         *session        // IN: the session to reset
574         )
575     {
576         session->commandCode = 0;       // No command
577
578         // No locality selected
579         MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
580
581         // The cpHash size to zero
582         session->u1.cpHash.b.size = 0;
583
584         // No timeout
585         session->timeOut = 0;
586
587         // Reset the pcrCounter
588         session->pcrCounter = 0;
589
590         // Reset the policy hash
591         MemorySet(&session->u2.policyDigest.t.buffer, 0,
592                   session->u2.policyDigest.t.size);
593
594         // Reset the session attributes
595         MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
596
597         // set the policy attribute
598         session->attributes.isPolicy = SET;
599     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

9.8.6.9 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 86

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
600  TPMI_YES_NO
601  SessionCapGetLoaded(
602      TPMI_SH_POLICY   handle,        // IN: start handle
603      UINT32           count,         // IN: count of returned handles
604      TPML_HANDLE     *handleList     // OUT: list of handle
605      )
606  {
607      TPMI_YES_NO     more = NO;
608      UINT32          i;
609
610      pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
611
612      // Initialize output handle list
613      handleList->count = 0;
614
615      // The maximum count of handles we may return is MAX_CAP_HANDLES
616      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
617
618      // Iterate session context ID slots to get loaded session handles
619      for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
620      {
621          // If session is active
622          if(gr.contextArray[i] != 0)
623          {
624              // If session is loaded
625              if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
626              {
627                  if(handleList->count < count)
628                  {
629                      SESSION         *session;
630
631                      // If we have not filled up the return list, add this
632                      // session handle to it
633                      // assume that this is going to be an HMAC session
634                      handle = i + HMAC_SESSION_FIRST;
635                      session = SessionGet(handle);
636                      if(session->attributes.isPolicy)
637                          handle = i + POLICY_SESSION_FIRST;
638                      handleList->handle[handleList->count] = handle;
639                      handleList->count++;
640                  }
641                  else
642                  {
643                      // If the return list is full but we still have loaded object
644                      // available, report this and stop iterating
645                      more = YES;
646                      break;
647                  }
648              }
649          }
650      }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
651
652         return more;
653
654     }
```

9.8.6.10 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 87

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
655     TPMI_YES_NO
656     SessionCapGetSaved(
657         TPMI_SH_HMAC     handle,        // IN: start handle
658         UINT32           count,         // IN: count of returned handles
659         TPML_HANDLE     *handleList     // OUT: list of handle
660         )
661     {
662         TPMI_YES_NO     more = NO;
663         UINT32          i;
664
665         pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);
666
667         // Initialize output handle list
668         handleList->count = 0;
669
670         // The maximum count of handles we may return is MAX_CAP_HANDLES
671         if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
672
673         // Iterate session context ID slots to get loaded session handles
674         for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
675         {
676             // If session is active
677             if(gr.contextArray[i] != 0)
678             {
679                 // If session is saved
680                 if (gr.contextArray[i] > MAX_LOADED_SESSIONS)
681                 {
682                     if(handleList->count < count)
683                     {
684                         // If we have not filled up the return list, add this
685                         // session handle to it
686                         handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
687                         handleList->count++;
688                     }
689                     else
690                     {
691                         // If the return list is full but we still have loaded object
692                         // available, report this and stop iterating
693                         more = YES;
694                         break;
695                     }
696                 }
697             }
698         }
699
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
700         return more;
701
702     }
```

9.8.6.11 SessionCapGetLoadedNumber()

This function return the number of authorization sessions currently loaded into TPM RAM.

```
703     UINT32
704     SessionCapGetLoadedNumber(
705         void
706         )
707     {
708         return MAX_LOADED_SESSIONS - s_freeSessionSlots;
709     }
```

9.8.6.12 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 might just be an estimate. The only constraint for the estimate is, if it is one or more, then at least one session needs to be loadable.

```
710     UINT32
711     SessionCapGetLoadedAvail(
712         void
713         )
714     {
715         return s_freeSessionSlots;
716     }
```

9.8.6.13 SessionCapGetActiveNumber()

This function returns the number of active authorization sessions currently being tracked by the TPM.

```
717     UINT32
718     SessionCapGetActiveNumber(
719         void
720         )
721     {
722         UINT32              i;
723         UINT32              num = 0;
724
725         // Iterate the context array to find the number of non-zero slots
726         for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
727         {
728             if(gr.contextArray[i] != 0) num++;
729         }
730
731         return num;
732     }
```

9.8.6.14 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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
UINT32
SessionCapGetActiveAvail(
    void
    )
{
    UINT32              i;
    UINT32              num = 0;

    // Iterate the context array to find the number of zero slots
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        if(gr.contextArray[i] == 0) num++;
    }

    return num;
}
```

9.9 Time.c

9.9.1 Introduction

This file contains the functions relating to the TPM's time functions including the interface to the implementation-specific time functions.

9.9.2 Includes

```
#include "InternalRoutines.h"
#include "Platform.h"
```

9.9.3 Functions

9.9.3.1 TimePowerOn()

This function initialize time info at _TPM_Init().

```
void
TimePowerOn(
    void
    )
{
    TPM_SU          orderlyShutDown;

    // Read orderly data info from NV memory
    NvReadReserved(NV_ORDERLY_DATA, &go);

    // Read orderly shut down state flag
    NvReadReserved(NV_ORDERLY, &orderlyShutDown);

    // 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(orderlyShutDown == SHUTDOWN_NONE)
        go.clockSafe= NO;
    else
        go.clockSafe = YES;

    // Set the initial state of the DRBG
    CryptDrbgGetPutState(PUT_STATE);

    // Clear time since TPM power on
    g_time = 0;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
28
29      return;
30  }
```

9.9.3.2 TimeStartup()

This function updates the *resetCount* and *restartCount* components of TPMS_CLOCK_INFO structure at TPM2_Startup().

```
31  void
32  TimeStartup(
33      STARTUP_TYPE     type           // IN: start up type
34      )
35  {
36      if(type == SU_RESUME)
37      {
38          // Resume sequence
39          gr.restartCount++;
40      }
41      else
42      {
43          if(type == SU_RESTART)
44          {
45              // Hibernate sequence
46              gr.clearCount++;
47              gr.restartCount++;
48          }
49          else
50          {
51              // Reset sequence
52              // Increase resetCount
53              gp.resetCount++;
54
55              // Write resetCount to NV
56              NvWriteReserved(NV_RESET_COUNT, &gp.resetCount);
57              gp.totalResetCount++;
58
59              // We do not expect the total reset counter overflow during the life
60              // time of TPM.  If it ever happens, TPM will be put to failure mode
61              // and there is no way to recover it.
62              // The reason that there is no recovery is that we don't increment
63              // the NV totalResetCount when incrementing would make it 0. When the
64              // TPM starts up again, the old value of totalResetCount will be read
65              // and we will get right back to here with the increment failing.
66              if(gp.totalResetCount == 0)
67                  FAIL(FATAL_ERROR_INTERNAL);
68
69
70              // Write total reset counter to NV
71              NvWriteReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
72
73              // Reset restartCount
74              gr.restartCount = 0;
75          }
76      }
77
78      return;
79  }
```

9.9.3.3 TimeUpdateToCurrent()

This function updates the *Time* and *Clock* in the global TPMS_TIME_INFO structure.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

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 implementations does not do rate limiting. If the implementation does do rate limiting, then the *Clock* update should not be inhibited even when doing rather limiting.

```
void
TimeUpdateToCurrent(
    void
    )
{
    UINT64          oldClock;
    UINT64          elapsed;
#define CLOCK_UPDATE_MASK  ((1ULL << NV_CLOCK_UPDATE_INTERVAL)- 1)

    // Can't update time during the dark interval or when rate limiting.
    if(NvIsAvailable() != TPM_RC_SUCCESS)
        return;

    // Save the old clock value
    oldClock = go.clock;

    // Update the time info to current
    elapsed = _plat__ClockTimeElapsed();
    go.clock += elapsed;
    g_time += elapsed;

    // Check to see if the update has caused a need for an nvClock update
    // CLOCK_UPDATE_MASK is measured by second, while the value in go.clock is
    // recorded by millisecond.  Align the clock value to second before the bit
    // operations
    if( ((go.clock/1000) | CLOCK_UPDATE_MASK)
            > ((oldClock/1000) | CLOCK_UPDATE_MASK))
    {
        // Going to update the time state so the safe flag
        // should be set
        go.clockSafe = YES;

        // Get the DRBG state before updating orderly data
        CryptDrbgGetPutState(GET_STATE);

        NvWriteReserved(NV_ORDERLY_DATA, &go);
    }

    // Call self healing logic for dictionary attack parameters
    DASelfHeal();

    return;
}
```

9.9.3.4 TimeSetAdjustRate()

This function is used to perform rate adjustment on *Time* and *Clock*.

```
void
TimeSetAdjustRate(
    TPM_CLOCK_ADJUST     adjust         // IN: adjust constant
    )
{
    switch(adjust)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
129         {
130             case TPM_CLOCK_COARSE_SLOWER:
131                 _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
132                 break;
133             case TPM_CLOCK_COARSE_FASTER:
134                 _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
135                 break;
136             case TPM_CLOCK_MEDIUM_SLOWER:
137                 _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
138                 break;
139             case TPM_CLOCK_MEDIUM_FASTER:
140                 _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
141                 break;
142             case TPM_CLOCK_FINE_SLOWER:
143                 _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
144                 break;
145             case TPM_CLOCK_FINE_FASTER:
146                 _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
147                 break;
148             case TPM_CLOCK_NO_CHANGE:
149                 break;
150             default:
151                 pAssert(FALSE);
152                 break;
153         }
154
155         return;
156     }
```

9.9.3.5 TimeGetRange()

This function is used to access TPMS_TIME_INFO. The TPMS_TIME_INFO structure is treaded as an array of bytes, and a byte offset and length determine what bytes are returned.

Table 88

Error Returns	Meaning
TPM_RC_RANGE	invalid data range

```
157     TPM_RC
158     TimeGetRange(
159         UINT16          offset,         // IN: offset in TPMS_TIME_INFO
160         UINT16          size,           // IN: size of data
161         TIME_INFO       *dataBuffer     // OUT: result buffer
162         )
163     {
164         TPMS_TIME_INFO      timeInfo;
165         UINT16              infoSize;
166         BYTE                infoData[sizeof(TPMS_TIME_INFO)];
167         BYTE                *buffer;
168
169         // Fill TPMS_TIME_INFO structure
170         timeInfo.time = g_time;
171         TimeFillInfo(&timeInfo.clockInfo);
172
173         // Marshal TPMS_TIME_INFO to canonical form
174         buffer = infoData;
175         infoSize = TPMS_TIME_INFO_Marshal(&timeInfo, &buffer, NULL);
176
177         // Check if the input range is valid
178         if(offset + size > infoSize) return TPM_RC_RANGE;
179
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
180         // Copy info data to output buffer
181         MemoryCopy(dataBuffer, infoData + offset, size, sizeof(TIME_INFO));
182 
183         return TPM_RC_SUCCESS;
184     }
```

9.9.3.6 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```
185     void
186     TimeFillInfo(
187         TPMS_CLOCK_INFO     *clockInfo
188         )
189     {
190         clockInfo->clock = go.clock;
191         clockInfo->resetCount = gp.resetCount;
192         clockInfo->restartCount = gr.restartCount;
193 
194         // If NV is not available, clock stopped advancing and the value reported is
195         // not "safe".
196         if(NvIsAvailable() == TPM_RC_SUCCESS)
197             clockInfo->safe = go.clockSafe;
198         else
199             clockInfo->safe = NO;
200 
201         return;
202     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10 Support

10.1 AlgorithmCap.c

10.1.1 Description

This file contains the algorithm property definitions for the algorithms and the code for the TPM2_GetCapability() to return the algorithm properties.

10.1.2 Includes and Defines

```
#include "InternalRoutines.h"
typedef struct
{
    TPM_ALG_ID          algID;
    TPMA_ALGORITHM      attributes;
} ALGORITHM;
static const ALGORITHM    s_algorithms[] =
{
#ifdef TPM_ALG_RSA
    {TPM_ALG_RSA,           {1, 0, 0, 1, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA1
    {TPM_ALG_SHA1,          {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_HMAC
    {TPM_ALG_HMAC,          {0, 0, 1, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_AES
    {TPM_ALG_AES,           {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_MGF1
    {TPM_ALG_MGF1,          {0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif

    {TPM_ALG_KEYEDHASH,     {0, 0, 1, 1, 0, 1, 1, 0, 0}},

#ifdef TPM_ALG_XOR
    {TPM_ALG_XOR,           {0, 1, 1, 0, 0, 0, 0, 0, 0}},
#endif

#ifdef TPM_ALG_SHA256
    {TPM_ALG_SHA256,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA384
    {TPM_ALG_SHA384,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA512
    {TPM_ALG_SHA512,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_WHIRLPOOL512
    {TPM_ALG_WHIRLPOOL512,  {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SM3_256
    {TPM_ALG_SM3_256,       {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SM4
    {TPM_ALG_SM4,          {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_RSASSA
    {TPM_ALG_RSASSA,        {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
#ifdef TPM_ALG_RSAES
    {TPM_ALG_RSAES,         {1, 0, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_RSAPSS
    {TPM_ALG_RSAPSS,        {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_OAEP
    {TPM_ALG_OAEP,          {1, 0, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_ECDSA
    {TPM_ALG_ECDSA,         {1, 0, 0, 0, 0, 1, 0, 1, 0}},
#endif
#ifdef TPM_ALG_ECDH
    {TPM_ALG_ECDH,          {1, 0, 0, 0, 0, 0, 0, 1, 0}},
#endif
#ifdef TPM_ALG_ECDAA
    {TPM_ALG_ECDAA,         {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_ECSCHNORR
    {TPM_ALG_ECSCHNORR,     {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_KDF1_SP800_56a
    {TPM_ALG_KDF1_SP800_56a,{0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif
#ifdef TPM_ALG_KDF2
    {TPM_ALG_KDF2,          {0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif
#ifdef TPM_ALG_KDF1_SP800_108
    {TPM_ALG_KDF1_SP800_108,{0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif
#ifdef TPM_ALG_ECC
    {TPM_ALG_ECC,           {1, 0, 0, 1, 0, 0, 0, 0, 0}},
#endif

    {TPM_ALG_SYMCIPHER,     {0, 0, 0, 1, 0, 0, 0, 0, 0}},

#ifdef TPM_ALG_CAMELLIA
    {TPM_ALG_CAMELLIA,      {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_CTR
    {TPM_ALG_CTR,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_OFB
    {TPM_ALG_OFB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_CBC
    {TPM_ALG_CBC,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_CFB
    {TPM_ALG_CFB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_ECB
    {TPM_ALG_ECB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
};
```

10.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2_GetCapability() to return a list of the implemented algorithms.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 89

Return Value	Meaning
YES	more algorithms to report
NO	no more algorithms to report

```
107   TPMI_YES_NO
108   AlgorithmCapGetImplemented(
109       TPM_ALG_ID                  algID,      // IN: the starting algorithm ID
110       UINT32                      count,      // IN: count of returned algorithms
111       TPML_ALG_PROPERTY           *algList    // OUT: algorithm list
112   )
113   {
114       TPMI_YES_NO     more = NO;
115       UINT32          i;
116       UINT32          algNum;
117
118       // initialize output algorithm list
119       algList->count = 0;
120
121       // The maximum count of algorithms we may return is MAX_CAP_ALGS.
122       if(count > MAX_CAP_ALGS)
123           count = MAX_CAP_ALGS;
124
125       // Compute how many algorithms are defined in s_algorithms array.
126       algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
127
128       // Scan the implemented algorithm list to see if there is a match to 'algID'.
129       for(i = 0; i < algNum; i++)
130       {
131           // If algID is less than the starting algorithm ID, skip it
132           if(s_algorithms[i].algID < algID)
133               continue;
134           if(algList->count < count)
135           {
136               // If we have not filled up the return list, add more algorithms
137               // to it
138               algList->algProperties[algList->count].alg = s_algorithms[i].algID;
139               algList->algProperties[algList->count].algProperties =
140                   s_algorithms[i].attributes;
141               algList->count++;
142           }
143           else
144           {
145               // If the return list is full but we still have algorithms
146               // available, report this and stop scanning.
147               more = YES;
148               break;
149           }
150
151       }
152
153       return more;
154
155   }
156   LIB_EXPORT
157   void
158   AlgorithmGetImplementedVector(
159       ALGORITHM_VECTOR    *implemented    // OUT: the implemented bits are SET
160       )
161   {
162       int                     index;
163
164       // Nothing implemented until we say it is
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
165         MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));
166
167         for(index =  (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
168             index >= 0;
169             index--)
170                 SET_BIT(s_algorithms[index].algID, *implemented);
171         return;
172     }
```

10.2 Bits.c

10.2.1 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.

10.2.2 Includes

```
1   #include "InternalRoutines.h"
```

10.2.3 Functions

10.2.3.1 BitIsSet()

This function is used to check the setting of a bit in an array of bits.

Table 90

Return Value	Meaning
TRUE	bit is set
FALSE	bit is not set

```
2   BOOL
3   BitIsSet(
4       unsigned int        bitNum,         // IN: number of the bit in 'bArray'
5       BYTE                *bArray,        // IN: array containing the bits
6       unsigned int        arraySize       // IN: size in bytes of 'bArray'
7       )
8   {
9       pAssert(arraySize > (bitNum >> 3));
10      return((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
11  }
```

10.2.3.2 BitSet()

This function will set the indicated bit in *bArray*.

```
12  void
13  BitSet(
14      unsigned int        bitNum,         // IN: number of the bit in 'bArray'
15      BYTE                *bArray,        // IN: array containing the bits
16      unsigned int        arraySize       // IN: size in bytes of 'bArray'
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
    )
{
    pAssert(arraySize > bitNum/8);
    bArray[bitNum >> 3] |= (1 << (bitNum & 7));
}
```

10.2.3.3 BitClear()

This function will clear the indicated bit in *bArray*.

```
void
BitClear(
    unsigned int     bitNum,        // IN: number of the bit in 'bArray'.
    BYTE            *bArray,        // IN: array containing the bits
    unsigned int     arraySize      // IN: size in bytes of 'bArray'
    )
{
    pAssert(arraySize > bitNum/8);
    bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
}
```

10.3 CommandAttributeData.c

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

```
static const TPMA_CC    s_ccAttr [] = {
        {0x011f, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_UndefineSpaceSpecial
        {0x0120, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_EvictControl
        {0x0121, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_HierarchyControl
        {0x0122, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_UndefineSpace
        {0x0123, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
        {0x0124, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_ChangeEPS
        {0x0125, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_ChangePPS
        {0x0126, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_Clear
        {0x0127, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClearControl
        {0x0128, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClockSet
        {0x0129, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_HierarchyChangeAuth
        {0x012a, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_DefineSpace
        {0x012b, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Allocate
        {0x012c, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_SetAuthPolicy
        {0x012d, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PP_Commands
        {0x012e, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetPrimaryPolicy
        {0x012f, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_FieldUpgradeStart
        {0x0130, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClockRateAdjust
        {0x0131, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_CreatePrimary
        {0x0132, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_GlobalWriteLock
        {0x0133, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_GetCommandAuditDigest
        {0x0134, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Increment
        {0x0135, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_SetBits
        {0x0136, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Extend
        {0x0137, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Write
        {0x0138, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_WriteLock
        {0x0139, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_DictionaryAttackLockReset
        {0x013a, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_DictionaryAttackParameters
        {0x013b, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_ChangeAuth
        {0x013c, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Event
        {0x013d, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Reset
        {0x013e, 0, 0, 0, 1, 1, 0, 0, 0},     // TPM_CC_SequenceComplete
        {0x013f, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetAlgorithmSet
        {0x0140, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetCommandCodeAuditStatus
        {0x0141, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_FieldUpgradeData
        {0x0142, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_IncrementalSelfTest
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
38          {0x0143, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_SelfTest
39          {0x0144, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_Startup
40          {0x0145, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_Shutdown
41          {0x0146, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_StirRandom
42          {0x0147, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_ActivateCredential
43          {0x0148, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Certify
44          {0x0149, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_PolicyNV
45          {0x014a, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_CertifyCreation
46          {0x014b, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Duplicate
47          {0x014c, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_GetTime
48          {0x014d, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_GetSessionAuditDigest
49          {0x014e, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Read
50          {0x014f, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_ReadLock
51          {0x0150, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_ObjectChangeAuth
52          {0x0151, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_PolicySecret
53          {0x0152, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Rewrap
54          {0x0153, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Create
55          {0x0154, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ECDH_ZGen
56          {0x0155, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_HMAC
57          {0x0156, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Import
58          {0x0157, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_Load
59          {0x0158, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Quote
60          {0x0159, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_RSA_Decrypt
61          {0x015a, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
62          {0x015b, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_HMAC_Start
63          {0x015c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_SequenceUpdate
64          {0x015d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Sign
65          {0x015e, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Unseal
66          {0x015f, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
67          {0x0160, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_PolicySigned
68          {0x0161, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_ContextLoad
69          {0x0162, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ContextSave
70          {0x0163, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ECDH_KeyGen
71          {0x0164, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_EncryptDecrypt
72          {0x0165, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_FlushContext
73          {0x0166, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
74          {0x0167, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_LoadExternal
75          {0x0168, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_MakeCredential
76          {0x0169, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_ReadPublic
77          {0x016a, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyAuthorize
78          {0x016b, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyAuthValue
79          {0x016c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCommandCode
80          {0x016d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCounterTimer
81          {0x016e, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCpHash
82          {0x016f, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyLocality
83          {0x0170, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyNameHash
84          {0x0171, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyOR
85          {0x0172, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyTicket
86          {0x0173, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ReadPublic
87          {0x0174, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_RSA_Encrypt
88          {0x0175, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
89          {0x0176, 0, 0, 0, 0, 2, 1, 0, 0},     // TPM_CC_StartAuthSession
90          {0x0177, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_VerifySignature
91          {0x0178, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_ECC_Parameters
92          {0x0179, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_FirmwareRead
93          {0x017a, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetCapability
94          {0x017b, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetRandom
95          {0x017c, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetTestResult
96          {0x017d, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_Hash
97          {0x017e, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_PCR_Read
98          {0x017f, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPCR
99          {0x0180, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyRestart
100         {0x0181, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_ReadClock
101         {0x0182, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Extend
102         {0x0183, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_SetAuthValue
103         {0x0184, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_NV_Certify
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
104             {0x0185, 0, 1, 0, 1, 2, 0, 0, 0},     // TPM_CC_EventSequenceComplete
105             {0x0186, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_HashSequenceStart
106             {0x0187, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPhysicalPresence
107             {0x0188, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyDuplicationSelect
108             {0x0189, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyGetDigest
109             {0x018a, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_TestParms
110             {0x018b, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Commit
111             {0x018c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPassword
112             {0x018d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ZGen_2Phase
113             {0x018e, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_EC_Ephemeral
114             {0x018f, 0, 0, 0, 0, 1, 0, 0, 0}      // TPM_CC_PolicyNvWritten
115     };
116     typedef  UINT16              _ATTR_;
117     #define  NOT_IMPLEMENTED    (_ATTR_)(0)
118     #define  ENCRYPT_2          (_ATTR_)(1 <<  0)
119     #define  ENCRYPT_4          (_ATTR_)(1 <<  1)
120     #define  DECRYPT_2          (_ATTR_)(1 <<  2)
121     #define  DECRYPT_4          (_ATTR_)(1 <<  3)
122     #define  HANDLE_1_USER      (_ATTR_)(1 <<  4)
123     #define  HANDLE_1_ADMIN     (_ATTR_)(1 <<  5)
124     #define  HANDLE_1_DUP       (_ATTR_)(1 <<  6)
125     #define  HANDLE_2_USER      (_ATTR_)(1 <<  7)
126     #define  PP_COMMAND         (_ATTR_)(1 <<  8)
127     #define  IS_IMPLEMENTED     (_ATTR_)(1 <<  9)
128     #define  NO_SESSIONS        (_ATTR_)(1 << 10)
129     #define  NV_COMMAND         (_ATTR_)(1 << 11)
130     #define  PP_REQUIRED        (_ATTR_)(1 << 12)
131     #define  R_HANDLE           (_ATTR_)(1 << 13)
```

This is the command code attribute structure.

```
132     typedef UINT16 COMMAND_ATTRIBUTES;
133     static const COMMAND_ATTRIBUTES    s_commandAttributes [] = {
134         (_ATTR_)(CC_NV_UndefineSpaceSpecial                *
        (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),              // 0x011f
135         (_ATTR_)(CC_EvictControl                           *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0120
136         (_ATTR_)(CC_HierarchyControl                       *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0121
137         (_ATTR_)(CC_NV_UndefineSpace                       *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0122
138         (_ATTR_)                                  (NOT_IMPLEMENTED),
        // 0x0123 - Not assigned
139         (_ATTR_)(CC_ChangeEPS                              *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0124
140         (_ATTR_)(CC_ChangePPS                              *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0125
141         (_ATTR_)(CC_Clear                                  *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0126
142         (_ATTR_)(CC_ClearControl                           *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0127
143         (_ATTR_)(CC_ClockSet                               *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0128
144         (_ATTR_)(CC_HierarchyChangeAuth                    *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                   // 0x0129
145         (_ATTR_)(CC_NV_DefineSpace                         *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                   // 0x012a
146         (_ATTR_)(CC_PCR_Allocate                           *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x012b
147         (_ATTR_)(CC_PCR_SetAuthPolicy                      *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                   // 0x012c
148         (_ATTR_)(CC_PP_Commands                            *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),                            // 0x012d
149         (_ATTR_)(CC_SetPrimaryPolicy                       *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                   // 0x012e
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
150       (_ATTR_)(CC_FieldUpgradeStart          *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),                  // 0x012f
151       (_ATTR_)(CC_ClockRateAdjust            *
      (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0130
152       (_ATTR_)(CC_CreatePrimary              *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)), // 0x0131
153       (_ATTR_)(CC_NV_GlobalWriteLock         *
      (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0132
154       (_ATTR_)(CC_GetCommandAuditDigest      *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),      // 0x0133
155       (_ATTR_)(CC_NV_Increment               * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x0134
156       (_ATTR_)(CC_NV_SetBits                 * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x0135
157       (_ATTR_)(CC_NV_Extend                  *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x0136
158       (_ATTR_)(CC_NV_Write                   *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x0137
159       (_ATTR_)(CC_NV_WriteLock               * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x0138
160       (_ATTR_)(CC_DictionaryAttackLockReset  * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x0139
161       (_ATTR_)(CC_DictionaryAttackParameters * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x013a
162       (_ATTR_)(CC_NV_ChangeAuth              *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),                             // 0x013b
163       (_ATTR_)(CC_PCR_Event                  *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x013c
164       (_ATTR_)(CC_PCR_Reset                  * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x013d
165       (_ATTR_)(CC_SequenceComplete           *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x013e
166       (_ATTR_)(CC_SetAlgorithmSet            * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x013f
167       (_ATTR_)(CC_SetCommandCodeAuditStatus  *
      (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0140
168       (_ATTR_)(CC_FieldUpgradeData           * (IS_IMPLEMENTED+DECRYPT_2)),
      // 0x0141
169       (_ATTR_)(CC_IncrementalSelfTest        * (IS_IMPLEMENTED)),
      // 0x0142
170       (_ATTR_)(CC_SelfTest                   * (IS_IMPLEMENTED)),
      // 0x0143
171       (_ATTR_)(CC_Startup                    * (IS_IMPLEMENTED+NO_SESSIONS)),
      // 0x0144
172       (_ATTR_)(CC_Shutdown                   * (IS_IMPLEMENTED)),
      // 0x0145
173       (_ATTR_)(CC_StirRandom                 * (IS_IMPLEMENTED+DECRYPT_2)),
      // 0x0146
174       (_ATTR_)(CC_ActivateCredential         *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),     // 0x0147
175       (_ATTR_)(CC_Certify                    *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),     // 0x0148
176       (_ATTR_)(CC_PolicyNV                   *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x0149
177       (_ATTR_)(CC_CertifyCreation            *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x014a
178       (_ATTR_)(CC_Duplicate                  *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),                     // 0x014b
179       (_ATTR_)(CC_GetTime                    *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),      // 0x014c
180       (_ATTR_)(CC_GetSessionAuditDigest      *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),      // 0x014d
181       (_ATTR_)(CC_NV_Read                    *
      (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                              // 0x014e
182       (_ATTR_)(CC_NV_ReadLock                * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x014f
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
183         (_ATTR_)(CC_ObjectChangeAuth            *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),                   // 0x0150
184         (_ATTR_)(CC_PolicySecret                *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0151
185         (_ATTR_)(CC_Rewrap                      *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0152
186         (_ATTR_)(CC_Create                      *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0153
187         (_ATTR_)(CC_ECDH_ZGen                   *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0154
188         (_ATTR_)(CC_HMAC                        *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0155
189         (_ATTR_)(CC_Import                      *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0156
190         (_ATTR_)(CC_Load                        *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),           // 0x0157
191         (_ATTR_)(CC_Quote                       *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0158
192         (_ATTR_)(CC_RSA_Decrypt                 *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0159
193         (_ATTR_)                                  (NOT_IMPLEMENTED),
        // 0x015a - Not assigned
194         (_ATTR_)(CC_HMAC_Start                  *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),                     // 0x015b
195         (_ATTR_)(CC_SequenceUpdate              *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x015c
196         (_ATTR_)(CC_Sign                        *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x015d
197         (_ATTR_)(CC_Unseal                      *
        (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                              // 0x015e
198         (_ATTR_)                                  (NOT_IMPLEMENTED),
        // 0x015f - Not assigned
199         (_ATTR_)(CC_PolicySigned                * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
        // 0x0160
200         (_ATTR_)(CC_ContextLoad                 * (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
        // 0x0161
201         (_ATTR_)(CC_ContextSave                 * (IS_IMPLEMENTED+NO_SESSIONS)),
        // 0x0162
202         (_ATTR_)(CC_ECDH_KeyGen                 * (IS_IMPLEMENTED+ENCRYPT_2)),
        // 0x0163
203         (_ATTR_)(CC_EncryptDecrypt              *
        (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                              // 0x0164
204         (_ATTR_)(CC_FlushContext                * (IS_IMPLEMENTED+NO_SESSIONS)),
        // 0x0165
205         (_ATTR_)                                  (NOT_IMPLEMENTED),
        // 0x0166 - Not assigned
206         (_ATTR_)(CC_LoadExternal                *
        (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),                         // 0x0167
207         (_ATTR_)(CC_MakeCredential              * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
        // 0x0168
208         (_ATTR_)(CC_NV_ReadPublic               * (IS_IMPLEMENTED+ENCRYPT_2)),
        // 0x0169
209         (_ATTR_)(CC_PolicyAuthorize             * (IS_IMPLEMENTED+DECRYPT_2)),
        // 0x016a
210         (_ATTR_)(CC_PolicyAuthValue             * (IS_IMPLEMENTED)),
        // 0x016b
211         (_ATTR_)(CC_PolicyCommandCode           * (IS_IMPLEMENTED)),
        // 0x016c
212         (_ATTR_)(CC_PolicyCounterTimer          * (IS_IMPLEMENTED+DECRYPT_2)),
        // 0x016d
213         (_ATTR_)(CC_PolicyCpHash                * (IS_IMPLEMENTED+DECRYPT_2)),
        // 0x016e
214         (_ATTR_)(CC_PolicyLocality              * (IS_IMPLEMENTED)),
        // 0x016f
215         (_ATTR_)(CC_PolicyNameHash              * (IS_IMPLEMENTED+DECRYPT_2)),
        // 0x0170
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
216       (_ATTR_)(CC_PolicyOR                  * (IS_IMPLEMENTED)),
      // 0x0171
217       (_ATTR_)(CC_PolicyTicket              * (IS_IMPLEMENTED+DECRYPT_2)),
      // 0x0172
218       (_ATTR_)(CC_ReadPublic                * (IS_IMPLEMENTED+ENCRYPT_2)),
      // 0x0173
219       (_ATTR_)(CC_RSA_Encrypt               * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
      // 0x0174
220       (_ATTR_)                                (NOT_IMPLEMENTED),
      // 0x0175 - Not assigned
221       (_ATTR_)(CC_StartAuthSession          *
      (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),                          // 0x0176
222       (_ATTR_)(CC_VerifySignature           * (IS_IMPLEMENTED+DECRYPT_2)),
      // 0x0177
223       (_ATTR_)(CC_ECC_Parameters            * (IS_IMPLEMENTED)),
      // 0x0178
224       (_ATTR_)(CC_FirmwareRead              * (IS_IMPLEMENTED+ENCRYPT_2)),
      // 0x0179
225       (_ATTR_)(CC_GetCapability             * (IS_IMPLEMENTED)),
      // 0x017a
226       (_ATTR_)(CC_GetRandom                 * (IS_IMPLEMENTED+ENCRYPT_2)),
      // 0x017b
227       (_ATTR_)(CC_GetTestResult             * (IS_IMPLEMENTED+ENCRYPT_2)),
      // 0x017c
228       (_ATTR_)(CC_Hash                      * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
      // 0x017d
229       (_ATTR_)(CC_PCR_Read                  * (IS_IMPLEMENTED)),
      // 0x017e
230       (_ATTR_)(CC_PolicyPCR                 * (IS_IMPLEMENTED+DECRYPT_2)),
      // 0x017f
231       (_ATTR_)(CC_PolicyRestart             * (IS_IMPLEMENTED)),
      // 0x0180
232       (_ATTR_)(CC_ReadClock                 * (IS_IMPLEMENTED+NO_SESSIONS)),
      // 0x0181
233       (_ATTR_)(CC_PCR_Extend                * (IS_IMPLEMENTED+HANDLE_1_USER)),
      // 0x0182
234       (_ATTR_)(CC_PCR_SetAuthValue          *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                               // 0x0183
235       (_ATTR_)(CC_NV_Certify                *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),       // 0x0184
236       (_ATTR_)(CC_EventSequenceComplete     *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),                 // 0x0185
237       (_ATTR_)(CC_HashSequenceStart         * (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
      // 0x0186
238       (_ATTR_)(CC_PolicyPhysicalPresence    * (IS_IMPLEMENTED)),
      // 0x0187
239       (_ATTR_)(CC_PolicyDuplicationSelect   * (IS_IMPLEMENTED+DECRYPT_2)),
      // 0x0188
240       (_ATTR_)(CC_PolicyGetDigest           * (IS_IMPLEMENTED+ENCRYPT_2)),
      // 0x0189
241       (_ATTR_)(CC_TestParms                 * (IS_IMPLEMENTED)),
      // 0x018a
242       (_ATTR_)(CC_Commit                    *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                     // 0x018b
243       (_ATTR_)(CC_PolicyPassword            * (IS_IMPLEMENTED)),
      // 0x018c
244       (_ATTR_)(CC_ZGen_2Phase               *
      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                     // 0x018d
245       (_ATTR_)(CC_EC_Ephemeral              * (IS_IMPLEMENTED+ENCRYPT_2)),
      // 0x018e
246       (_ATTR_)(CC_PolicyNvWritten           * (IS_IMPLEMENTED))
      // 0x018f
247   };
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.4 CommandCodeAttributes.c

10.4.1 Introduction

This file contains the functions for testing various command properties.

10.4.2 Includes and Defines

```
#include    "Tpm.h"
#include    "InternalRoutines.h"
typedef UINT16          ATTRIBUTE_TYPE;
```

The following file is produced from the command tables in ISO/IEC 11889-3. It defines the attributes for each of the commands.

NOTE This file is currently produced by an automated process. Files produced from ISO/IEC 11889-2 or ISO/IEC 11889-3 tables through automated processes are not included in ISO/IEC 11889 so that there is no ambiguity about the information in ISO/IEC 11889-2 or ISO/IEC 11889-3 tables being the normative definition.

```
#include    "CommandAttributeData.c"
```

10.4.3 Command Attribute Functions

10.4.3.1 CommandAuthRole()

This function returns the authorization role required of a handle.

Table 91

Return Value	Meaning
AUTH_NONE	no authorization is required
AUTH_USER	user role authorization is required
AUTH_ADMIN	admin role authorization is required
AUTH_DUP	duplication role authorization is required

```
AUTH_ROLE
CommandAuthRole(
    TPM_CC      commandCode,        // IN: command code
    UINT32      handleIndex         // IN: handle index (zero based)
    )
{
    if(handleIndex > 1)
        return AUTH_NONE;
    if(handleIndex == 0) {
        ATTRIBUTE_TYPE  properties = s_commandAttributes[commandCode - TPM_CC_FIRST];
        if(properties & HANDLE_1_USER) return AUTH_USER;
        if(properties & HANDLE_1_ADMIN) return AUTH_ADMIN;
        if(properties & HANDLE_1_DUP) return AUTH_DUP;
        return AUTH_NONE;
    }
    if(s_commandAttributes[commandCode - TPM_CC_FIRST] & HANDLE_2_USER) return
AUTH_USER;
    return AUTH_NONE;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.4.3.2 CommandIsImplemented()

This function indicates if a command is implemented.

Table 92

Return Value	Meaning
TRUE	if the command is implemented
FALSE	if the command is not implemented

```
BOOL
CommandIsImplemented(
    TPM_CC            commandCode    // IN: command code
    )
{
    if(commandCode < TPM_CC_FIRST || commandCode > TPM_CC_LAST)
        return FALSE;
    if((s_commandAttributes[commandCode - TPM_CC_FIRST] & IS_IMPLEMENTED))
        return TRUE;
    else
        return FALSE;
}
```

10.4.3.3 CommandGetAttribute()

return a TPMA_CC structure for the given command code

```
TPMA_CC
CommandGetAttribute(
    TPM_CC            commandCode    // IN: command code
    )
{
    UINT32      size = sizeof(s_ccAttr) / sizeof(s_ccAttr[0]);
    UINT32      i;
    for(i = 0; i < size; i++) {
        if(s_ccAttr[i].commandIndex == (UINT16) commandCode)
            return s_ccAttr[i];
    }

    // This function should be called in the way that the command code
    // attribute is available.
    FAIL(FATAL_ERROR_INTERNAL);
}
```

10.4.3.4 EncryptSize()

This function returns the size of the decrypt size field. This function returns 0 if encryption is not allowed.

Table 93

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
int
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
52  EncryptSize(
53      TPM_CC            commandCode     // IN: commandCode
54      )
55  {
56      COMMAND_ATTRIBUTES  ca = s_commandAttributes[commandCode - TPM_CC_FIRST];
57      if(ca & ENCRYPT_2)
58          return 2;
59      if(ca & ENCRYPT_4)
60          return 4;
61      return 0;
62  }
```

10.4.3.5 DecryptSize()

This function returns the size of the decrypt size field. This function returns 0 if decryption is not allowed.

Table 94

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
63  int
64  DecryptSize(
65      TPM_CC            commandCode     // IN: commandCode
66      )
67  {
68      COMMAND_ATTRIBUTES  ca = s_commandAttributes[commandCode - TPM_CC_FIRST];
69
70      if(ca & DECRYPT_2)
71          return 2;
72      if(ca & DECRYPT_4)
73          return 4;
74      return 0;
75  }
```

10.4.3.6 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 95

Return Value	Meaning
TRUE	session is allowed with this command
FALSE	session is not allowed with this command

```
76  BOOL
77  IsSessionAllowed(
78      TPM_CC            commandCode     // IN: the command to be checked
79      )
80  {
81      if(s_commandAttributes[commandCode - TPM_CC_FIRST] & NO_SESSIONS)
82          return FALSE;
83      else
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
84          return TRUE;
85      }
```

10.4.3.7 IsHandleInResponse()

```
86      BOOL
87      IsHandleInResponse(
88          TPM_CC          commandCode
89          )
90      {
91          if(s_commandAttributes[commandCode - TPM_CC_FIRST] & R_HANDLE)
92              return TRUE;
93          else
94              return FALSE;
95      }
```

10.4.3.8 IsWriteOperation()

Checks to see if an operation will write to NV memory.

```
96      BOOL
97      IsWriteOperation(
98          TPM_CC          command        // IN: Command to check
99          )
100     {
101         switch (command)
102         {
103             case TPM_CC_NV_Write:
104             case TPM_CC_NV_Increment:
105             case TPM_CC_NV_SetBits:
106             case TPM_CC_NV_Extend:
107             // Nv write lock counts as a write operation for authorization purposes.
108             // We check to see if the NV is write locked before we do the authorization
109             // If it is locked, we fail the command early.
110             case TPM_CC_NV_WriteLock:
111                 return TRUE;
112             default:
113                 break;
114         }
115         return FALSE;
116     }
```

10.4.3.9 IsReadOperation()

Checks to see if an operation will write to NV memory.

```
117     BOOL
118     IsReadOperation(
119         TPM_CC          command        // IN: Command to check
120         )
121     {
122         switch (command)
123         {
124             case TPM_CC_NV_Read:
125             case TPM_CC_PolicyNV:
126             case TPM_CC_NV_Certify:
127             // Nv read lock counts as a read operation for authorization purposes.
128             // We check to see if the NV is read locked before we do the authorization
129             // If it is locked, we fail the command early.
130             case TPM_CC_NV_ReadLock:
131                 return TRUE;
132             default:
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
133                 break;
134         }
135         return FALSE;
136     }
```

10.4.3.10 CommandCapGetCCList()

This function returns a list of implemented commands and command attributes starting from the command in *commandCode*.

Table 96

Return Value	Meaning
YES	more command attributes are available
NO	no more command attributes are available

```
137     TPMI_YES_NO
138     CommandCapGetCCList(
139         TPM_CC            commandCode,   // IN: start command code
140         UINT32            count,         // IN: maximum count for number of entries in
141                                          //     'commandList'
142         TPML_CCA         *commandList    // OUT: list of TPMA_CC
143         )
144     {
145         TPMI_YES_NO       more = NO;
146         UINT32            i;
147
148         // initialize output handle list count
149         commandList->count = 0;
150
151         // The maximum count of commands that may be return is MAX_CAP_CC.
152         if(count > MAX_CAP_CC) count = MAX_CAP_CC;
153
154         // If the command code is smaller than TPM_CC_FIRST, start from TPM_CC_FIRST
155         if(commandCode < TPM_CC_FIRST) commandCode = TPM_CC_FIRST;
156
157         // Collect command attributes
158         for(i = commandCode; i <= TPM_CC_LAST; i++)
159         {
160             if(CommandIsImplemented(i))
161             {
162                 if(commandList->count < count)
163                 {
164                     // If the list is not full, add the attributes for this command.
165                     commandList->commandAttributes[commandList->count]
166                         = CommandGetAttribute(i);
167                     commandList->count++;
168                 }
169                 else
170                 {
171                     // If the list is full but there are more commands to report,
172                     // indicate this and return.
173                     more = YES;
174                     break;
175                 }
176             }
177         }
178         return more;
179     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.5 DRTM.c

10.5.1 Description

This file contains functions that simulate the DRTM events.

10.5.2 Includes

```
#include    "InternalRoutines.h"
```

10.5.3 Functions

10.5.3.1 Signal_Hash_Start()

This function interfaces between the platform code and _TPM_Hash_Start().

```
LIB_EXPORT void
Signal_Hash_Start(
    void
    )
{
    _TPM_Hash_Start();
    return;
}
```

10.5.3.2 Signal_Hash_Data()

This function interfaces between the platform code and _TPM_Hash_Data().

```
LIB_EXPORT void
Signal_Hash_Data(
    unsigned int     size,
    unsigned char   *buffer
    )
{
    _TPM_Hash_Data(size, buffer);
    return;
}
```

10.5.3.3 Signal_Hash_End()

This function interfaces between the platform code and _TPM_Hash_End().

```
LIB_EXPORT void
Signal_Hash_End(
    void
    )
{
    _TPM_Hash_End();
    return;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

10.6 Entity.c

10.6.1 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.

10.6.2 Includes

```
#include "InternalRoutines.h"
```

10.6.3 Functions

10.6.3.1 EntityGetLoadStatus()

This function will indicate if the entity associated with a handle is present in TPM memory. If the handle is a persistent object handle, and the object exists, the persistent object is moved from NV memory into a RAM object slot and the persistent handle is replaced with the transient object handle for the slot.

Table 97

Error Returns	Meaning
TPM_RC_HANDLE	handle type does not match
TPM_RC_REFERENCE_H0	entity is not present
TPM_RC_HIERARCHY	entity belongs to a disabled hierarchy
TPM_RC_OBJECT_MEMORY	handle is an evict object but there is no space to load it to RAM

```
TPM_RC
EntityGetLoadStatus(
    TPM_HANDLE      *handle,        // IN/OUT: handle of the entity
    TPM_CC           commandCode    // IN: the commmandCode
    )
{
    TPM_RC          result = TPM_RC_SUCCESS;

    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:
#endif
                case TPM_RH_ENDORSEMENT:
                    if(!gc.ehEnable)
                        result = TPM_RC_HIERARCHY;
                    break;
                case TPM_RH_PLATFORM:
                    if(!g_phEnable)
                        result = TPM_RC_HIERARCHY;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
32                      break;
33                      // null handle, PW session handle and lockout
34                      // handle are always available
35                  case TPM_RH_NULL:
36                  case TPM_RS_PW:
37                  case TPM_RH_LOCKOUT:
38                      break;
39                  default:
40                      // handling of the manufacture_specific handles
41                      if(     ((TPM_RH)*handle >= TPM_RH_AUTH_00)
42                          &&  ((TPM_RH)*handle <= TPM_RH_AUTH_FF))
43                          // use the value that would have been returned from
44                          // unmarshaling if it did the handle filtering
45                              result = TPM_RC_VALUE;
46                      else
47                          pAssert(FALSE);
48                      break;
49              }
50              break;
51          case TPM_HT_TRANSIENT:
52              // For a transient object, check if the handle is associated
53              // with a loaded object.
54              if(!ObjectIsPresent(*handle))
55                  result = TPM_RC_REFERENCE_H0;
56              break;
57          case TPM_HT_PERSISTENT:
58              // Persistent object
59              // Copy the persistent object to RAM and replace the handle with the
60              // handle of the assigned slot.  A TPM_RC_OBJECT_MEMORY,
61              // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
62              // ObjectLoadEvict()
63              result = ObjectLoadEvict(handle, commandCode);
64              break;
65          case TPM_HT_HMAC_SESSION:
66              // For an HMAC session, see if the session is loaded
67              // and if the session in the session slot is actually
68              // an HMAC session.
69              if(SessionIsLoaded(*handle))
70              {
71                  SESSION         *session;
72                  session = SessionGet(*handle);
73                  // Check if the session is a HMAC session
74                  if(session->attributes.isPolicy == SET)
75                      result = TPM_RC_HANDLE;
76              }
77              else
78                  result = TPM_RC_REFERENCE_H0;
79              break;
80          case TPM_HT_POLICY_SESSION:
81              // For a policy session, see if the session is loaded
82              // and if the session in the session slot is actually
83              // a policy session.
84              if(SessionIsLoaded(*handle))
85              {
86                  SESSION             *session;
87                  session = SessionGet(*handle);
88                  // Check if the session is a policy session
89                  if(session->attributes.isPolicy == CLEAR)
90                      result = TPM_RC_HANDLE;
91              }
92              else
93                  result = TPM_RC_REFERENCE_H0;
94              break;
95          case TPM_HT_NV_INDEX:
96              // For an NV Index, use the platform-specific routine
97              // to search the IN Index space.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
98              result = NvIndexIsAccessible(*handle, commandCode);
99              break;
100         case TPM_HT_PCR:
101             // Any PCR handle that is unmarshaled successfully referenced
102             // a PCR that is defined.
103             break;
104         default:
105             // Any other handle type is a defect in the unmarshaling code.
106             pAssert(FALSE);
107             break;
108     }
109     return result;
110 }
```

10.6.3.2 EntityGetAuthValue()

This function is used to access the *authValue* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authValue* should have been verified by IsAuthValueAvailable().

This function copies the authorization value of the entity to *auth*.

Return value is the number of octets copied to *auth*.

```
111 UINT16
112 EntityGetAuthValue(
113     TPMI_DH_ENTITY   handle,        // IN: handle of entity
114     AUTH_VALUE      *auth           // OUT: authValue of the entity
115     )
116 {
117     TPM2B_AUTH       authValue = {0};
118
119     switch(HandleGetType(handle))
120     {
121         case TPM_HT_PERMANENT:
122             switch(handle)
123             {
124                 case TPM_RH_OWNER:
125                     // ownerAuth for TPM_RH_OWNER
126                     authValue = gp.ownerAuth;
127                     break;
128                 case TPM_RH_ENDORSEMENT:
129                     // endorsementAuth for TPM_RH_ENDORSEMENT
130                     authValue = gp.endorsementAuth;
131                     break;
132                 case TPM_RH_PLATFORM:
133                     // platformAuth for TPM_RH_PLATFORM
134                     authValue = gc.platformAuth;
135                     break;
136                 case TPM_RH_LOCKOUT:
137                     // lockoutAuth for TPM_RH_LOCKOUT
138                     authValue = gp.lockoutAuth;
139                     break;
140                 case TPM_RH_NULL:
141                     // nullAuth for TPM_RH_NULL. Return 0 directly here
142                     return 0;
143                     break;
144 #ifdef  VENDOR_PERMANENT
145                 case VENDOR_PERMANENT:
146                     // vendor auth value
147                     authValue = g_platformUniqueDetails;
148 #endif
149                 default:
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
150                     // If any other permanent handle is present it is
151                     // a code defect.
152                     pAssert(FALSE);
153                     break;
154                 }
155                 break;
156             case TPM_HT_TRANSIENT:
157                 // authValue for an object
158                 // A persistent object would have been copied into RAM
159                 // and would have an transient object handle here.
160                 {
161                     OBJECT          *object;
162                     object = ObjectGet(handle);
163                     // special handling if this is a sequence object
164                     if(ObjectIsSequence(object))
165                     {
166                         authValue = ((HASH_OBJECT *)object)->auth;
167                     }
168                     else
169                     {
170                         // Auth value is available only when the private portion of
171                         // the object is loaded.  The check should be made before
172                         // this function is called
173                         pAssert(object->attributes.publicOnly == CLEAR);
174                         authValue = object->sensitive.authValue;
175                     }
176                 }
177                 break;
178             case TPM_HT_NV_INDEX:
179                 // authValue for an NV index
180                 {
181                     NV_INDEX        nvIndex;
182                     NvGetIndexInfo(handle, &nvIndex);
183                     authValue = nvIndex.authValue;
184                 }
185                 break;
186             case TPM_HT_PCR:
187                 // authValue for PCR
188                 PCRGetAuthValue(handle, &authValue);
189                 break;
190             default:
191                 // If any other handle type is present here, then there is a defect
192                 // in the unmarshaling code.
193                 pAssert(FALSE);
194                 break;
195         }
196
197         // Copy the authValue
198         pAssert(authValue.t.size <= <K>sizeof(authValue.t.buffer));
199         MemoryCopy(auth, authValue.t.buffer, authValue.t.size, sizeof(TPMU_HA));
200
201         return authValue.t.size;
202     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

10.6.3.3 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.

 © ISO/IEC 2015 – All rights reserved

```
203  TPMI_ALG_HASH
204  EntityGetAuthPolicy(
205      TPMI_DH_ENTITY   handle,        // IN: handle of entity
206      TPM2B_DIGEST    *authPolicy     // OUT: authPolicy of the entity
207      )
208  {
209      TPMI_ALG_HASH       hashAlg = TPM_ALG_NULL;
210
211      switch(HandleGetType(handle))
212      {
213          case TPM_HT_PERMANENT:
214              switch(handle)
215              {
216                  case TPM_RH_OWNER:
217                      // ownerPolicy for TPM_RH_OWNER
218                      *authPolicy = gp.ownerPolicy;
219                      hashAlg = gp.ownerAlg;
220                      break;
221                  case TPM_RH_ENDORSEMENT:
222                      // endorsementPolicy for TPM_RH_ENDORSEMENT
223                      *authPolicy = gp.endorsementPolicy;
224                      hashAlg = gp.endorsementAlg;
225                      break;
226                  case TPM_RH_PLATFORM:
227                      // platformPolicy for TPM_RH_PLATFORM
228                      *authPolicy = gc.platformPolicy;
229                      hashAlg = gc.platformAlg;
230                      break;
231                  case TPM_RH_LOCKOUT:
232                      // lockoutPolicy for TPM_RH_LOCKOUT
233                      *authPolicy = gp.lockoutPolicy;
234                      hashAlg = gp.lockoutAlg;
235                      break;
236                  default:
237                      // If any other permanent handle is present it is
238                      // a code defect.
239                      pAssert(FALSE);
240                      break;
241              }
242              break;
243          case TPM_HT_TRANSIENT:
244              // authPolicy for an object
245              {
246                  OBJECT *object = ObjectGet(handle);
247                  *authPolicy = object->publicArea.authPolicy;
248                  hashAlg = object->publicArea.nameAlg;
249              }
250              break;
251          case TPM_HT_NV_INDEX:
252              // authPolicy for a NV index
253              {
254                  NV_INDEX        nvIndex;
255                  NvGetIndexInfo(handle, &nvIndex);
256                  *authPolicy = nvIndex.publicArea.authPolicy;
257                  hashAlg = nvIndex.publicArea.nameAlg;
258              }
259              break;
260          case TPM_HT_PCR:
261              // authPolicy for a PCR
262              hashAlg = PCRGetAuthPolicy(handle, authPolicy);
263              break;
264          default:
265              // If any other handle type is present it is a code defect.
266              pAssert(FALSE);
267              break;
268      }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
269         return hashAlg;
270     }
```

10.6.3.4 EntityGetName()

This function returns the Name associated with a handle. It will set *name* to the Name and return the size of the Name string.

```
271     UINT16
272     EntityGetName(
273         TPMI_DH_ENTITY   handle,        // IN: handle of entity
274         NAME            *name           // OUT: name of entity
275         )
276     {
277         UINT16          nameSize;
278
279         switch(HandleGetType(handle))
280         {
281             case TPM_HT_TRANSIENT:
282                 // Name for an object
283                 nameSize = ObjectGetName(handle, name);
284                 break;
285             case TPM_HT_NV_INDEX:
286                 // Name for a NV index
287                 nameSize = NvGetName(handle, name);
288                 break;
289             default:
290                 // For all other types, the handle is the Name
291                 nameSize = TPM_HANDLE_Marshal(&handle, (BYTE **)&name, NULL);
292                 break;
293         }
294         return nameSize;
295     }
```

10.6.3.5 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. All other handles belong to the platform hierarchy. or an NV Index.

```
296     TPMI_RH_HIERARCHY
297     EntityGetHierarchy(
298         TPMI_DH_ENTITY   handle          // IN :handle of entity
299         )
300     {
301         TPMI_RH_HIERARCHY       hierarcy = TPM_RH_NULL;
302
303         switch(HandleGetType(handle))
304         {
305             case TPM_HT_PERMANENT:
306                 // hierarchy for a permanent handle
307                 switch(handle)
308                 {
309                     case TPM_RH_PLATFORM:
310                     case TPM_RH_ENDORSEMENT:
311                     case TPM_RH_NULL:
312                         hierarcy = handle;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
313                     break;
314                 // all other permanent handles are associated with the owner
315                 // hierarchy. (should only be TPM_RH_OWNER and TMP_RH_LOCKOUT)
316                 default:
317                     hierarchy = TPM_RH_OWNER;
318                     break;
319             }
320             break;
321         case TPM_HT_NV_INDEX:
322             // hierarchy for NV index
323             {
324                 NV_INDEX        nvIndex;
325                 NvGetIndexInfo(handle, &nvIndex);
326                 // If only the platform can delete the index, then it is
327                 // considered to be in the platform hierarchy, otherwise it
328                 // is in the owner hierarchy.
329                 if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
330                     hierarchy = TPM_RH_PLATFORM;
331                 else
332                     hierarchy = TPM_RH_OWNER;
333             }
334             break;
335         case TPM_HT_TRANSIENT:
336             // hierarchy for an object
337             {
338                 OBJECT          *object;
339                 object = ObjectGet(handle);
340                 if(object->attributes.ppsHierarchy)
341                 {
342                     hierarchy = TPM_RH_PLATFORM;
343                 }
344                 else if(object->attributes.epsHierarchy)
345                 {
346                     hierarchy = TPM_RH_ENDORSEMENT;
347                 }
348                 else if(object->attributes.spsHierarchy)
349                 {
350                     hierarchy = TPM_RH_OWNER;
351                 }
352 
353             }
354             break;
355         case TPM_HT_PCR:
356             hierarchy = TPM_RH_OWNER;
357             break;
358         default:
359             pAssert(0);
360             break;
361     }
362     // this is unreachable but it provides a return value for the default
363     // case which makes the complier happy
364     return hierarchy;
365 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

10.7 Global.c

10.7.1 Description

This file will instance the TPM variables that are not stack allocated. The descriptions for these variables is in Global.h.

© ISO/IEC 2015 – All rights reserved

10.7.2 Includes and Defines

```
#define GLOBAL_C
#include "InternalRoutines.h"
```

10.7.3 Global Data Values

These values are visible across multiple modules.

```
BOOL                    g_phEnable;
const UINT16            g_rcIndex[15] = {TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,
                                         TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,
                                         TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,
                                         TPM_RC_D, TPM_RC_E, TPM_RC_F
                                        };
TPM_HANDLE              g_exclusiveAuditSession;
UINT64                  g_time;
BOOL                    g_pcrReConfig;
TPMI_DH_OBJECT          g_DRTMHandle;
BOOL                    g_DrtmPreStartup;
BOOL                    g_clearOrderly;
TPM_SU                  g_prevOrderlyState;
BOOL                    g_updateNV;
BOOL                    g_nvOk;
TPM2B_AUTH              g_platformUniqueDetails;
STATE_CLEAR_DATA        gc;
STATE_RESET_DATA        gr;
PERSISTENT_DATA         gp;
ORDERLY_DATA            go;
```

10.7.4 Private Values

10.7.4.1 SessionProcess.c

```
#ifndef __IGNORE_STATE__        // DO NOT DEFINE THIS VALUE
```

These values do not need to be retained between commands.

```
TPM_HANDLE              s_sessionHandles[MAX_SESSION_NUM];
TPMA_SESSION            s_attributes[MAX_SESSION_NUM];
TPM_HANDLE              s_associatedHandles[MAX_SESSION_NUM];
TPM2B_NONCE             s_nonceCaller[MAX_SESSION_NUM];
TPM2B_AUTH              s_inputAuthValues[MAX_SESSION_NUM];
UINT32                  s_encryptSessionIndex;
UINT32                  s_decryptSessionIndex;
UINT32                  s_auditSessionIndex;
TPM2B_DIGEST            s_cpHashForAudit;
UINT32                  s_sessionNum;
#endif // __IGNORE_STATE__
BOOL                    s_DAPendingOnNV;
#ifdef TPM_CC_GetCommandAuditDigest
TPM2B_DIGEST            s_cpHashForCommandAudit;
#endif
```

10.7.4.2 DA.c

```
UINT64                  s_selfHealTimer;
UINT64                  s_lockoutTimer;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.7.4.3 NV.c

```
41  UINT32              s_reservedAddr[NV_RESERVE_LAST];
42  UINT32              s_reservedSize[NV_RESERVE_LAST];
43  UINT32              s_ramIndexSize;
44  BYTE                s_ramIndex[RAM_INDEX_SPACE];
45  UINT32              s_ramIndexSizeAddr;
46  UINT32              s_ramIndexAddr;
47  UINT32              s_maxCountAddr;
48  UINT32              s_evictNvStart;
49  UINT32              s_evictNvEnd;
50  TPM_RC              s_NvStatus;
```

10.7.4.4 Object.c

```
51  OBJECT_SLOT         s_objects[MAX_LOADED_OBJECTS];
```

10.7.4.5 PCR.c

```
52  PCR                 s_pcrs[IMPLEMENTATION_PCR];
```

10.7.4.6 Session.c

```
53  SESSION_SLOT        s_sessions[MAX_LOADED_SESSIONS];
54  UINT32              s_oldestSavedSession;
55  int                 s_freeSessionSlots;
```

10.7.4.7 Manufacture.c

```
56  BOOL                g_manufactured = FALSE;
```

10.7.4.8 Power.c

```
57  BOOL                s_initialized = FALSE;
```

10.7.4.9 MemoryLib.c

The *s_actionOutputBuffer* should not be modifiable by the host system until the TPM has returned a response code. The *s_actionOutputBuffer* should not be accessible until response parameter encryption, if any, is complete. This memory is not used between commands.

```
58  #ifndef __IGNORE_STATE__        // DO NOT DEFINE THIS VALUE
59  UINT32   s_actionInputBuffer[1024];          // action input buffer
60  UINT32   s_actionOutputBuffer[1024];         // action output buffer
61  BYTE     s_responseBuffer[MAX_RESPONSE_SIZE];// response buffer
62  #endif
```

10.7.4.10 SelfTest.c

Define these values here if the AlgorithmTests() project is not used

```
63  #ifndef SELF_TEST
64  ALGORITHM_VECTOR    g_implementedAlgorithms;
65  ALGORITHM_VECTOR    g_toTest;
66  #endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.7.4.11 TpmFail.c

```
67   jmp_buf             g_jumpBuffer;
68   BOOL                g_forceFailureMode;
69   BOOL                g_inFailureMode;
70   UINT32              s_failFunction;
71   UINT32              s_failLine;
72   UINT32              s_failCode;
```

10.8 Handle.c

10.8.1 Description

This file contains the functions that return the type of a handle.

10.8.2 Includes

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

10.8.3 Functions

10.8.3.1 HandleGetType()

This function returns the type of a handle which is the MSO of the handle.

```
 3   TPM_HT
 4   HandleGetType(
 5       TPM_HANDLE       handle         // IN: a handle to be checked
 6       )
 7   {
 8       // return the upper bytes of input data
 9       return (TPM_HT) ((handle & HR_RANGE_MASK) >> HR_SHIFT);
10   }
```

10.8.3.2 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:

```
11   TPM_HANDLE
12   NextPermanentHandle(
13       TPM_HANDLE       inHandle       // IN: the handle to check
14       )
15   {
16       TPM_HANDLE      retVal = 0;
17
18       switch (inHandle)
19       {
20           case TPM_RH_OWNER:
21           case TPM_RH_NULL:
22           case TPM_RS_PW:
23           case TPM_RH_LOCKOUT:
24           case TPM_RH_ENDORSEMENT:
25           case TPM_RH_PLATFORM:
26           case TPM_RH_PLATFORM_NV:
27   #ifdef  VENDOR_PERMANENT
28           case VENDOR_PERMANENT:
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
#endif
            retVal = inHandle;
            break;
        default:
            break;
    }
    return 0;
}
```

10.8.3.3 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 98

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
TPMI_YES_NO
PermanentCapGetHandles(
    TPM_HANDLE       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_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 = NextPermanentHandle(handle);
            i != 0; i = NextPermanentHandle(i+1))
    {
        if(handleList->count < count)
        {
            // If we have not filled up the return list, add this permanent
            // handle to it
            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;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.9 Locality.c

10.9.1 Includes

```
#include "InternalRoutines.h"
```

10.9.2 LocalityGetAttributes()

This function will convert a locality expressed as an integer into TPMA_LOCALITY form.

The function returns the locality attribute.

```
TPMA_LOCALITY
LocalityGetAttributes(
    UINT8            locality       // IN: locality value
    )
{
    TPMA_LOCALITY            locality_attributes;
    BYTE                    *localityAsByte = (BYTE *)&locality_attributes;

    MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
    switch(locality)
    {
        case 0:
            locality_attributes.TPM_LOC_ZERO = SET;
            break;
        case 1:
            locality_attributes.TPM_LOC_ONE = SET;
            break;
        case 2:
            locality_attributes.TPM_LOC_TWO = SET;
            break;
        case 3:
            locality_attributes.TPM_LOC_THREE = SET;
            break;
        case 4:
            locality_attributes.TPM_LOC_FOUR = SET;
            break;
        default:
            pAssert(locality < 256 && locality > 31);
            *localityAsByte = locality;
            break;
    }
    return locality_attributes;
}
```

10.10 Manufacture.c

10.10.1 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.

10.10.2 Includes and Data Definitions

```
#define MANUFACTURE_C
#include "InternalRoutines.h"
#include "Global.h"
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.10.3 Functions

10.10.3.1 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 99

Return Value	Meaning
0	success
1	manufacturing process previously performed

```
4   LIB_EXPORT int
5   TPM_Manufacture(
6       BOOL            firstTime       // IN: indicates if this is the first call from
7                                       //     main()
8       )
9   {
10      TPM_SU          orderlyShutdown;
11      UINT64          totalResetCount = 0;
12
13      // If TPM has been manufactured, return indication.
14      if(!firstTime && g_manufactured)
15          return 1;
16
17      // initialize crypto units
18      //CryptInitUnits();
19
20      //
21      s_selfHealTimer = 0;
22      s_lockoutTimer = 0;
23      s_DAPendingOnNV = FALSE;
24
25
26      // initialize NV
27      NvInit();
28
29  #ifdef _DRBG_STATE_SAVE
30      // Initialize the drbg. This needs to come before the install
31      // of the hierarchies
32      if(!_cpri__Startup())               // Have to start the crypto units first
33          FAIL(FATAL_ERROR_INTERNAL);
34      _cpri__DrbgGetPutState(PUT_STATE, 0, NULL);
35  #endif
36
37      // default configuration for PCR
38      PCRSimStart();
39
40      // initialize pre-installed hierarchy data
41      // This should happen after NV is initialized because hierarchy data is
42      // stored in NV.
43      HierarchyPreInstall_Init();
44
45      // initialize dictionary attack parameters
46      DAPreInstall_Init();
47
48      // initialize PP list
49      PhysicalPresencePreInstall_Init();
50
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
51      // initialize command audit list
52      CommandAuditPreInstall_Init();
53
54      // first start up is required to be Startup(CLEAR)
55      orderlyShutdown = TPM_SU_CLEAR;
56      NvWriteReserved(NV_ORDERLY, &orderlyShutdown);
57
58      // initialize the firmware version
59      gp.firmwareV1 = FIRMWARE_V1;
60  #ifdef FIRMWARE_V2
61      gp.firmwareV2 = FIRMWARE_V2;
62  #else
63      gp.firmwareV2 = 0;
64  #endif
65      NvWriteReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
66      NvWriteReserved(NV_FIRMWARE_V2, &gp.firmwareV2);
67
68      // initialize the total reset counter to 0
69      NvWriteReserved(NV_TOTAL_RESET_COUNT, &totalResetCount);
70
71      // initialize the clock stuff
72      go.clock = 0;
73      go.clockSafe = YES;
74
75  #ifdef _DRBG_STATE_SAVE
76      // initialize the current DRBG state in NV
77
78      _cpri__DrbgGetPutState(GET_STATE, sizeof(go.drbgState), (BYTE *)&go.drbgState);
79  #endif
80
81      NvWriteReserved(NV_ORDERLY_DATA, &go);
82
83      // Commit NV writes.  Manufacture process is an artificial process existing
84      // only in simulator environment and it is not defined in ISO/IEC 11889
85      // what should be the expected behavior if the NV write fails at this
86      // point.  Therefore, it is assumed the NV write here is always success and
87      // no return code of this function is checked.
88      NvCommit();
89
90      g_manufactured = TRUE;
91
92      return 0;
93  }
```

10.10.3.2 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.

Table 100

Return Value	Meaning
0	success
1	TPM not previously manufactured

```
94  LIB_EXPORT int
95  TPM_TearDown(
96      void
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
    )
{
    // stop crypt units
    CryptStopUnits();

    g_manufactured = FALSE;
    return 0;
}
```

10.11 Marshal.c

10.11.1 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).

10.11.2 Unmarshal and Marshal a Value

In ISO/IEC 11889-2, Table 39, the TPMI_DH_OBJECT type is defined as shown in Table 101:

Table 101— Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type from ISO/IEC 11889-2

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

This generates the following unmarshaling code:

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

and the following marshaling code:

NOTE 1 The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

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

10.11.3 Unmarshal and Marshal a Union

In ISO/IEC 11889-2, Table 185, the TPMU_PUBLIC_PARMS union is defined as shown in Table 102:

Table 102 — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S> from ISO/IEC 11889-2

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

From this table, the following unmarshaling code is generated:

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

NOTE 2 The **#ifdef/#endif** directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

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

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.11.4 Unmarshal and Marshal a Structure

In ISO/IEC 11889-2, Table 187, the TPMT_PUBLIC structure is defined as shown in Table 103:

Table 103 — Definition of TPMT_PUBLIC Structure from ISO/IEC 11889-2

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	"algorithm" associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object.
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

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

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

```
1   TPM_RC
2   TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, bool flag)
3   {
4       TPM_RC    result;
5       result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                          buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                       buffer, size, flag);
11      if(result != TPM_RC_SUCCESS)
12          return result;
13      result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                     buffer, size);
15      if(result != TPM_RC_SUCCESS)
16          return result;
17      result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                      buffer, size);
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                           buffer, size, (UINT32)target->type);
24      if(result != TPM_RC_SUCCESS)
25          return result;
26
27      result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                        buffer, size, (UINT32)target->type);
29      if(result != TPM_RC_SUCCESS)
30          return result;
31
32      return TPM_RC_SUCCESS;
33  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

The marshaling code for the TPMT_PUBLIC structure is:

```
1   UINT16
2   TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
3   {
4       UINT16    result = 0;
5       result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
6                                      (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
7       result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
8                                    (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
9   ;
10      result = (UINT16)(result + TPMA_OBJECT_Marshal(
11                               (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
12
13      result = (UINT16)(result + TPM2B_DIGEST_Marshal(
14                                  (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));
15
16      result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
17                               (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
18                                                                  (UINT32)source->type));
19
20      result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
21                                    (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
22                                                                  (UINT32)source->type));
23
24      return result;
25  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.11.5 Unmarshal and Marshal an Array

In ISO/IEC 11889-2, Table 98, the TPML_DIGEST Structure is defined as shown in Table 104:

Table 104 — Definition of TPML_DIGEST Structure from ISO/IEC 11889-2

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

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

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

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

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

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

    return TPM_RC_SUCCESS;
}
```

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

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

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

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

    return result;
}
```

The marshaling code for the array is:

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

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.11.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 10.11.5 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.

In ISO/IEC 11889-2, Table 76, the TPM2B_EVENT is defined as shown in Table 105:

Table 105 — Definition of TPM2B_EVENT Structure from ISO/IEC 11889-2

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

```
TPM_RC
TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    // if size equal to 0, the rest of the structure is a zero buffer.  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;
}
```

Which use these structure definitions:

```
typedef struct {
    UINT16          size;
    BYTE            buffer[1];
} TPM2B;

typedef struct {
    UINT16          size;
    BYTE            buffer[1024];
} EVENT_2B;

typedef union {
    EVENT_2B        t;    // The type-specific union member
    TPM2B           b;    // The generic union member
} TPM2B_EVENT;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

10.12 MemoryLib.c

10.12.1 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 in the TPM in order to avoid namespace contamination.

10.12.2 Includes and Data Definitions

```
#define MEMORY_LIB_C
#include "InternalRoutines.h"
```

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. However, the *s_actionInputBuffer* and *s_responseBuffer* are not needed at the same time and they could be the same buffer.

10.12.3 Functions on BYTE Arrays

10.12.3.1 MemoryMove()

This function moves data from one place in memory to another. No safety checks of any type are performed. If source and data buffer overlap, then the move is done as if an intermediate buffer were used.

NOTE This function is used by MemoryCopy(), MemoryCopy2B(), and MemoryConcat2b() and needs the caller to know the maximum size of the destination buffer so that there is no possibility of buffer overrun.

```
LIB_EXPORT void
MemoryMove(
    void            *destination,   // OUT: move destination
    const void      *source,        // IN: move source
    UINT32           size,          // IN: number of octets to moved
    UINT32           dSize          // IN: size of the receive buffer
    )
{
    const BYTE *p = (BYTE *)source;
    BYTE *q = (BYTE *)destination;

    if(destination == NULL || source == NULL)
        return;

    pAssert(size <= dSize);
    // if the destination buffer has a lower address than the
    // source, then moving bytes in ascending order is safe.
    dSize -= size;

    if (p>q || (p+size <= q))
    {
        while(size--)
            *q++ = *p++;
    }
    // If the destination buffer has a higher address than the
    // source, then move bytes from the end to the beginning.
    else if (p < q)
    {
        p += size;
        q += size;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
33          while (size--)
34              *--q = *--p;
35      }
36
37      // If the source and destination address are the same, nothing to move.
38      return;
39  }
```

10.12.3.2 MemoryCopy()

This function moves data from one place in memory to another. No safety checks of any type are performed. If the destination and source overlap, then the results are unpredictable.

```
40
41  #ifndef MemoryMove //%
42  void
43  MemoryCopy(
44      void            *destination,   // OUT: copy destination
45      void            *source,        // IN: copy source
46      UINT32           size,          // IN: number of octets being copied
47      UINT32           dSize          // IN: size of the receive buffer
48      )
49  {
50      MemoryMove(destination, source, size, dSize);
51  }
52  #else //%
53  //%#define MemoryCopy(destination, source, size, destSize)       \
54  //%    MemoryMove((destination), (source), (size), (destSize))
55  #endif //%
```

10.12.3.3 MemoryEqual()

This function indicates if two buffers have the same values in the indicated number of bytes.

Table 106

Return Value	Meaning
TRUE	all octets are the same
FALSE	all octets are not the same

```
56  LIB_EXPORT BOOL
57  MemoryEqual(
58      const void      *buffer1,       // IN: compare buffer1
59      const void      *buffer2,       // IN: compare buffer2
60      UINT32           size           // IN: size of bytes being compared
61      )
62  {
63      BOOL         equal = TRUE;
64      const BYTE  *b1, *b2;
65
66      b1 = (BYTE *)buffer1;
67      b2 = (BYTE *)buffer2;
68
69      // Compare all bytes so that there is no leakage of information
70      // due to timing differences.
71      for(; size > 0; size--)
72          equal = (*b1++ == *b2++) && equal;
73
74      return equal;
75  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.12.3.4 MemoryCopy2B()

This function copies a TPM2B. This can be used when the TPM2B types are the same or different. No size checking is done on the destination so the caller should make sure that the destination is large enough.

This function returns the number of octets in the data buffer of the TPM2B.

```
76  LIB_EXPORT INT16
77  MemoryCopy2B(
78      TPM2B           *dest,          // OUT: receiving TPM2B
79      const TPM2B     *source,        // IN: source TPM2B
80      UINT16           dSize          // IN: size of the receiving buffer
81      )
82  {
83
84      if(dest == NULL)
85          return 0;
86      if(source == NULL)
87          dest->size = 0;
88      else
89      {
90          dest->size = source->size;
91          MemoryMove(dest->buffer, source->buffer, dest->size, dSize);
92      }
93      return dest->size;
94  }
```

10.12.3.5 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))*.

```
95  LIB_EXPORT void
96  MemoryConcat2B(
97      TPM2B           *aInOut,        // IN/OUT: destination 2B
98      TPM2B           *bIn,           // IN: second 2B
99      UINT16           aSize          // IN: The size of aInOut.buffer (max values for
100                                     //     aInOut.size)
101     )
102 {
103     MemoryMove(&aInOut->buffer[aInOut->size],
104                bIn->buffer,
105                bIn->size,
106                aSize - aInOut->size);
107     aInOut->size = aInOut->size + bIn->size;
108     return;
109 }
```

10.12.3.6 Memory2BEqual()

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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 107

Return Value	Meaning
TRUE	size and buffer contents are the same
FALSE	size or buffer contents are not the same

```
LIB_EXPORT BOOL
Memory2BEqual(
    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);
}
```

10.12.3.7 MemorySet()

This function will set all the octets in the specified memory range to the specified octet value.

NOTE The **dSize** parameter forces the caller to know how big the receiving buffer is to make sure that there is no possibility that the caller will inadvertently run over the end of the buffer.

```
LIB_EXPORT void
MemorySet(
    void            *destination,   // OUT: memory destination
    char             value,         // IN: fill value
    UINT32           size           // IN: number of octets to fill
    )
{
    char *p = (char *)destination;
    while (size--)
        *p++ = value;
    return;
}
```

10.12.3.8 MemoryGetActionInputBuffer()

This function returns the address of the buffer into which the command parameters will be unmarshaled in preparation for calling the command actions.

```
BYTE *
MemoryGetActionInputBuffer(
    UINT32           size           // Size, in bytes, required for the input
                                    // unmarshaling
    )
{
    BYTE        *buf = NULL;

    if(size > 0)
    {
        // In this implementation, a static buffer is set aside for action output.
        // Other implementations may apply additional optimization based on command
        // code or other factors.
        UINT32      *p = s_actionInputBuffer;
        buf = (BYTE *)p;
        pAssert(size < <K>sizeof(s_actionInputBuffer));

        // size of an element in the buffer
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
151 #define SZ      sizeof(s_actionInputBuffer[0])
152
153         for(size = (size + SZ - 1) / SZ; size > 0; size--)
154             *p++ = 0;
155 #undef SZ
156     }
157     return buf;
158 }
```

10.12.3.9 MemoryGetActionOutputBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```
159 void *
160 MemoryGetActionOutputBuffer(
161     TPM_CC          command         // Command that requires the buffer
162     )
163 {
164     // In this implementation, a static buffer is set aside for action output.
165     // Other implementations may apply additional optimization based on the command
166     // code or other factors.
167     command = 0;        // Unreferenced parameter
168     return s_actionOutputBuffer;
169 }
```

10.12.3.10 MemoryGetResponseBuffer()

This function returns the address into which the command response is marshaled from values in the action output buffer.

```
170 BYTE *
171 MemoryGetResponseBuffer(
172     TPM_CC          command         // Command that requires the buffer
173     )
174 {
175     // In this implementation, a static buffer is set aside for responses.
176     // Other implementation may apply additional optimization based on the command
177     // code or other factors.
178     command = 0;        // Unreferenced parameter
179     return s_responseBuffer;
180 }
```

10.12.3.11 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.

```
181 UINT16
182 MemoryRemoveTrailingZeros (
183     TPM2B_AUTH      *auth           // IN/OUT: value to adjust
184     )
185 {
186     BYTE        *a = &auth->t.buffer[auth->t.size-1];
187     for(; auth->t.size > 0; auth->t.size--)
188     {
189         if(*a--)
190             break;
191     }
192     return auth->t.size;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
193 }
```

10.13 Power.c

10.13.1 Description

This file contains functions that receive the simulated power state transitions of the TPM.

10.13.2 Includes and Data Definitions

```
1  #define POWER_C
2  #include "InternalRoutines.h"
```

10.13.3 Functions

10.13.3.1 TPMInit()

This function is used to process a power on event.

```
3  void
4  TPMInit(
5      void
6      )
7  {
8      // Set state as not initialized. This means that Startup is required
9      s_initialized = FALSE;
10
11     return;
12 }
```

10.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2_Startup() has completed successfully).

```
13 void
14 TPMRegisterStartup(
15     void
16     )
17 {
18     s_initialized = TRUE;
19
20     return;
21 }
```

10.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2_Startup() has completed successfully after a _TPM_Init()).

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 108

Return Value	Meaning
TRUE	TPM has been initialized
FALSE	TPM has not been initialized

```
22  BOOL
23  TPMIsStarted(
24      void
25      )
26  {
27      return s_initialized;
28  }
```

10.14 PropertyCap.c

10.14.1 Description

This file contains the functions that are used for accessing the TPM_CAP_TPM_PROPERTY values.

10.14.2 Includes

```
1   #include "InternalRoutines.h"
```

10.14.3 Functions

10.14.3.1 PCRGetProperty()

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 109

Return Value	Meaning
TRUE	referenced property exists and *value* set
FALSE	referenced property does not exist

```
2   static BOOL
3   TPMPropertyIsDefined(
4       TPM_PT           property,      // IN: property
5       UINT32          *value          // OUT: property value
6       )
7   {
8       switch(property)
9       {
10          case TPM_PT_FAMILY_INDICATOR:
11              // from the title page of ISO/IEC 11889
12              // For ISO/IEC 11889, the value is "2.0".
13              *value = TPM_SPEC_FAMILY;
14              break;
15          case TPM_PT_LEVEL:
16              // from the title page of ISO/IEC 11889
17              *value = TPM_SPEC_LEVEL;
18              break;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
19          case TPM_PT_REVISION:
20              // from the title page of ISO/IEC 11889
21              *value = TPM_SPEC_VERSION;
22              break;
23          case TPM_PT_DAY_OF_YEAR:
24              // computed from the date value on the title page of ISO/IEC 11889
25              *value = TPM_SPEC_DAY_OF_YEAR;
26              break;
27          case TPM_PT_YEAR:
28              // from the title page of ISO/IEC 11889
29              *value = TPM_SPEC_YEAR;
30              break;
31          case TPM_PT_MANUFACTURER:
32              // vendor ID unique to each TPM manufacturer
33              *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
34              break;
35          case TPM_PT_VENDOR_STRING_1:
36              // first four characters of the vendor ID string
37              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
38              break;
39          case TPM_PT_VENDOR_STRING_2:
40              // second four characters of the vendor ID string
41  #ifdef VENDOR_STRING_2
42              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
43  #else
44              *value = 0;
45  #endif
46              break;
47          case TPM_PT_VENDOR_STRING_3:
48              // third four characters of the vendor ID string
49  #ifdef VENDOR_STRING_3
50              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
51  #else
52              *value = 0;
53  #endif
54              break;
55          case TPM_PT_VENDOR_STRING_4:
56              // fourth four characters of the vendor ID string
57  #ifdef VENDOR_STRING_4
58              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
59  #else
60              *value = 0;
61  #endif
62              break;
63          case TPM_PT_VENDOR_TPM_TYPE:
64              // vendor-defined value indicating the TPM model
65              *value = 1;
66              break;
67          case TPM_PT_FIRMWARE_VERSION_1:
68              // more significant 32-bits of a vendor-specific value
69              *value = gp.firmwareV1;
70              break;
71          case TPM_PT_FIRMWARE_VERSION_2:
72              // less significant 32-bits of a vendor-specific value
73              *value = gp.firmwareV2;
74              break;
75          case TPM_PT_INPUT_BUFFER:
76              // maximum size of TPM2B_MAX_BUFFER
77              *value = MAX_DIGEST_BUFFER;
78              break;
79          case TPM_PT_HR_TRANSIENT_MIN:
80              // minimum number of transient objects that can be held in TPM
81              // RAM
82              *value = MAX_LOADED_OBJECTS;
83              break;
84          case TPM_PT_HR_PERSISTENT_MIN:
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
85              // minimum number of persistent objects that can be held in
86              // TPM NV memory
87              // In this implementation, there is no minimum number of
88              // persistent objects.
89              *value = MIN_EVICT_OBJECTS;
90              break;
91          case TPM_PT_HR_LOADED_MIN:
92              // minimum number of authorization sessions that can be held in
93              // TPM RAM
94              *value = MAX_LOADED_SESSIONS;
95              break;
96          case TPM_PT_ACTIVE_SESSIONS_MAX:
97              // number of authorization sessions that may be active at a time
98              *value = MAX_ACTIVE_SESSIONS;
99              break;
100         case TPM_PT_PCR_COUNT:
101             // number of PCR implemented
102             *value = IMPLEMENTATION_PCR;
103             break;
104         case TPM_PT_PCR_SELECT_MIN:
105             // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
106             *value = PCR_SELECT_MIN;
107             break;
108         case TPM_PT_CONTEXT_GAP_MAX:
109             // maximum allowed difference (unsigned) between the contextID
110             // values of two saved session contexts
111             *value = (1 << (<K>sizeof(CONTEXT_SLOT) * 8)) - 1;
112             break;
113         case TPM_PT_NV_COUNTERS_MAX:
114             // maximum number of NV indexes that are allowed to have the
115             // TPMA_NV_COUNTER attribute SET
116             // In this implementation, there is no limitation on the number
117             // of counters, except for the size of the NV Index memory.
118             *value = 0;
119             break;
120         case TPM_PT_NV_INDEX_MAX:
121             // maximum size of an NV index data area
122             *value = MAX_NV_INDEX_SIZE;
123             break;
124         case TPM_PT_MEMORY:
125             // a TPMA_MEMORY indicating the memory management method for the TPM
126             {
127                 TPMA_MEMORY         attributes = {0};
128                 attributes.sharedNV = SET;
129                 attributes.objectCopiedToRam = SET;
130
131                 // Note: Different compilers may require a different method to cast
132                 // a bit field structure to a UINT32.
133                 *value = * (UINT32 *) &attributes;
134                 break;
135             }
136         case TPM_PT_CLOCK_UPDATE:
137             // interval, in seconds, between updates to the copy of
138             // TPMS_TIME_INFO .clock in NV
139             *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
140             break;
141         case TPM_PT_CONTEXT_HASH:
142             // algorithm used for the integrity hash on saved contexts and
143             // for digesting the fuData of TPM2_FirmwareRead()
144             *value = CONTEXT_INTEGRITY_HASH_ALG;
145             break;
146         case TPM_PT_CONTEXT_SYM:
147             // algorithm used for encryption of saved contexts
148             *value = CONTEXT_ENCRYPT_ALG;
149             break;
150         case TPM_PT_CONTEXT_SYM_SIZE:
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
151             // size of the key used for encryption of saved contexts
152             *value = CONTEXT_ENCRYPT_KEY_BITS;
153             break;
154         case TPM_PT_ORDERLY_COUNT:
155             // maximum difference between the volatile and non-volatile
156             // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
157             *value = MAX_ORDERLY_COUNT;
158             break;
159         case TPM_PT_MAX_COMMAND_SIZE:
160             // maximum value for 'commandSize'
161             *value = MAX_COMMAND_SIZE;
162             break;
163         case TPM_PT_MAX_RESPONSE_SIZE:
164             // maximum value for 'responseSize'
165             *value = MAX_RESPONSE_SIZE;
166             break;
167         case TPM_PT_MAX_DIGEST:
168             // maximum size of a digest that can be produced by the TPM
169             *value = sizeof(TPMU_HA);
170             break;
171         case TPM_PT_MAX_OBJECT_CONTEXT:
172             // maximum size of a TPMS_CONTEXT that will be returned by
173             // TPM2_ContextSave for object context
174             *value = 0;
175 
176             // adding sequence, saved handle and hierarchy
177             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
178                       sizeof(TPMI_RH_HIERARCHY);
179             // add size field in TPM2B_CONTEXT
180             *value += sizeof(UINT16);
181 
182             // add integrity hash size
183             *value += sizeof(UINT16) +
184                       CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
185 
186             // Add fingerprint size, which is the same as sequence size
187             *value += sizeof(UINT64);
188 
189             // Add OBJECT structure size
190             *value += sizeof(OBJECT);
191             break;
192         case TPM_PT_MAX_SESSION_CONTEXT:
193             // the maximum size of a TPMS_CONTEXT that will be returned by
194             // TPM2_ContextSave for object context
195             *value = 0;
196 
197             // adding sequence, saved handle and hierarchy
198             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
199                       sizeof(TPMI_RH_HIERARCHY);
200             // Add size field in TPM2B_CONTEXT
201             *value += sizeof(UINT16);
202 
203             // Add integrity hash size
204             *value += sizeof(UINT16) +
205                       CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
206             // Add fingerprint size, which is the same as sequence size
207             *value += sizeof(UINT64);
208 
209             // Add SESSION structure size
210             *value += sizeof(SESSION);
211             break;
212         case TPM_PT_PS_FAMILY_INDICATOR:
213             // platform specific values for the TPM_PT_PS parameters from
214             // the relevant platform-specific specification
215             // In the reference implementation, all of these values are 0.
216             *value = 0;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
217             break;
218         case TPM_PT_PS_LEVEL:
219             // level of the platform-specific specification
220             *value = 0;
221             break;
222         case TPM_PT_PS_REVISION:
223             // specification Revision times 100 for the platform-specific
224             // specification
225             *value = 0;
226             break;
227         case TPM_PT_PS_DAY_OF_YEAR:
228             // platform-specific specification day of year using TCG calendar
229             *value = 0;
230             break;
231         case TPM_PT_PS_YEAR:
232             // platform-specific specification year using the CE
233             *value = 0;
234             break;
235         case TPM_PT_SPLIT_MAX:
236             // number of split signing operations supported by the TPM
237             *value = 0;
238     #ifdef TPM_ALG_ECC
239             *value = sizeof(gr.commitArray) * 8;
240     #endif
241             break;
242         case TPM_PT_TOTAL_COMMANDS:
243             // total number of commands implemented in the TPM
244             // Since the reference implementation does not have any
245             // vendor-defined commands, this will be the same as the
246             // number of library commands.
247         {
248             UINT32 i;
249             *value = 0;
250
251             // calculate implemented command numbers
252             for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
253             {
254                 if(CommandIsImplemented(i)) (*value)++;
255             }
256             break;
257         }
258         case TPM_PT_LIBRARY_COMMANDS:
259             // number of commands from the TPM library that are implemented
260         {
261             UINT32 i;
262             *value = 0;
263
264             // calculate implemented command numbers
265             for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
266             {
267                 if(CommandIsImplemented(i)) (*value)++;
268             }
269             break;
270         }
271         case TPM_PT_VENDOR_COMMANDS:
272             // number of vendor commands that are implemented
273             *value = 0;
274             break;
275         case TPM_PT_PERMANENT:
276             // TPMA_PERMANENT
277         {
278             TPMA_PERMANENT          flags = {0};
279             if(gp.ownerAuth.t.size != 0)
280                 flags.ownerAuthSet = SET;
281             if(gp.endorsementAuth.t.size != 0)
282                 flags.endorsementAuthSet = SET;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
283             if(gp.lockoutAuth.t.size != 0)
284                 flags.lockoutAuthSet = SET;
285             if(gp.disableClear)
286                 flags.disableClear = SET;
287             if(gp.failedTries >= gp.maxTries)
288                 flags.inLockout = SET;
289             // In this implementation, EPS is always generated by TPM
290             flags.tpmGeneratedEPS = SET;
291 
292             // Note: Different compilers may require a different method to cast
293             // a bit field structure to a UINT32.
294             *value = * (UINT32 *) &flags;
295             break;
296         }
297         case TPM_PT_STARTUP_CLEAR:
298             // TPMA_STARTUP_CLEAR
299         {
300             TPMA_STARTUP_CLEAR      flags = {0};
301             if(g_phEnable)
302                 flags.phEnable = SET;
303             if(gc.shEnable)
304                 flags.shEnable = SET;
305             if(gc.ehEnable)
306                 flags.ehEnable = SET;
307             if(gc.phEnableNV)
308                 flags.phEnableNV = SET;
309             if(g_prevOrderlyState != SHUTDOWN_NONE)
310                 flags.orderly = SET;
311 
312             // Note: Different compilers may require a different method to cast
313             // a bit field structure to a UINT32.
314             *value = * (UINT32 *) &flags;
315             break;
316         }
317         case TPM_PT_HR_NV_INDEX:
318             // number of NV indexes currently defined
319             *value = NvCapGetIndexNumber();
320             break;
321         case TPM_PT_HR_LOADED:
322             // number of authorization sessions currently loaded into TPM
323             // RAM
324             *value = SessionCapGetLoadedNumber();
325             break;
326         case TPM_PT_HR_LOADED_AVAIL:
327             // number of additional authorization sessions, of any type,
328             // that could be loaded into TPM RAM
329             *value = SessionCapGetLoadedAvail();
330             break;
331         case TPM_PT_HR_ACTIVE:
332             // number of active authorization sessions currently being
333             // tracked by the TPM
334             *value = SessionCapGetActiveNumber();
335             break;
336         case TPM_PT_HR_ACTIVE_AVAIL:
337             // number of additional authorization sessions, of any type,
338             // that could be created
339             *value = SessionCapGetActiveAvail();
340             break;
341         case TPM_PT_HR_TRANSIENT_AVAIL:
342             // estimate of the number of additional transient objects that
343             // could be loaded into TPM RAM
344             *value = ObjectCapGetTransientAvail();
345             break;
346         case TPM_PT_HR_PERSISTENT:
347             // number of persistent objects currently loaded into TPM
348             // NV memory
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
349                 *value = NvCapGetPersistentNumber();
350                 break;
351             case TPM_PT_HR_PERSISTENT_AVAIL:
352                 // number of additional persistent objects that could be loaded
353                 // into NV memory
354                 *value = NvCapGetPersistentAvail();
355                 break;
356             case TPM_PT_NV_COUNTERS:
357                 // number of defined NV indexes that have NV TPMA_NV_COUNTER
358                 // attribute SET
359                 *value = NvCapGetCounterNumber();
360                 break;
361             case TPM_PT_NV_COUNTERS_AVAIL:
362                 // number of additional NV indexes that can be defined with their
363                 // TPMA_NV_COUNTER attribute SET
364                 *value = NvCapGetCounterAvail();
365                 break;
366             case TPM_PT_ALGORITHM_SET:
367                 // region code for the TPM
368                 *value = gp.algorithmSet;
369                 break;
370 
371             case TPM_PT_LOADED_CURVES:
372         #ifdef TPM_ALG_ECC
373                 // number of loaded ECC curves
374                 *value = CryptCapGetEccCurveNumber();
375         #else // TPM_ALG_ECC
376                 *value = 0;
377         #endif // TPM_ALG_ECC
378                 break;
379 
380             case TPM_PT_LOCKOUT_COUNTER:
381                 // current value of the lockout counter
382                 *value = gp.failedTries;
383                 break;
384             case TPM_PT_MAX_AUTH_FAIL:
385                 // number of authorization failures before DA lockout is invoked
386                 *value = gp.maxTries;
387                 break;
388             case TPM_PT_LOCKOUT_INTERVAL:
389                 // number of seconds before the value reported by
390                 // TPM_PT_LOCKOUT_COUNTER is decremented
391                 *value = gp.recoveryTime;
392                 break;
393             case TPM_PT_LOCKOUT_RECOVERY:
394                 // number of seconds after a lockoutAuth failure before use of
395                 // lockoutAuth may be attempted again
396                 *value = gp.lockoutRecovery;
397                 break;
398             case TPM_PT_AUDIT_COUNTER_0:
399                 // high-order 32 bits of the command audit counter
400                 *value = (UINT32) (gp.auditCounter >> 32);
401                 break;
402             case TPM_PT_AUDIT_COUNTER_1:
403                 // low-order 32 bits of the command audit counter
404                 *value = (UINT32) (gp.auditCounter);
405                 break;
406             default:
407                 // property is not defined
408                 return FALSE;
409                 break;
410         }
411 
412         return TRUE;
413     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.14.3.2 TPMCapGetProperties()

This function is used to get the TPM_PT values. The search of properties will start at *property* and continue until *propertyList* has as many values as will fit, or the last property has been reported, or the list has as many values as requested in *count*.

Table 110

Return Value	Meaning
YES	more properties are available
NO	no more properties to be reported

```
414  TPMI_YES_NO
415  TPMCapGetProperties(
416      TPM_PT                      property,      // IN: the starting TPM property
417      UINT32                      count,         // IN: maximum number of returned
418                                                 //     properties
419      TPML_TAGGED_TPM_PROPERTY    *propertyList  // OUT: property list
420      )
421  {
422      TPMI_YES_NO     more = NO;
423      UINT32          i;
424
425      // initialize output property list
426      propertyList->count = 0;
427
428      // maximum count of properties we may return is MAX_PCR_PROPERTIES
429      if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;
430
431      // If property is less than PT_FIXED, start from PT_FIXED.
432      if(property < PT_FIXED) property = PT_FIXED;
433
434      // Scan through the TPM properties of the requested group.
435      // The size of TPM property group is PT_GROUP * 2 for fix and
436      // variable groups.
437      for(i = property; i <= PT_FIXED + PT_GROUP * 2; i++)
438      {
439          UINT32          value;
440          if(TPMPropertyIsDefined((TPM_PT) i, &value))
441          {
442              if(propertyList->count < count)
443              {
444
445                  // If the list is not full, add this property
446                  propertyList->tpmProperty[propertyList->count].property =
447                      (TPM_PT) i;
448                  propertyList->tpmProperty[propertyList->count].value = value;
449                  propertyList->count++;
450              }
451              else
452              {
453                  // If the return list is full but there are more properties
454                  // available, set the indication and exit the loop.
455                  more = YES;
456                  break;
457              }
458          }
459      }
460      return more;
461  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

10.15 TpmFail.c

10.15.1 Includes, Defines, and Types

```
#define     TPM_FAIL_C
#include    "InternalRoutines.h"
#include    <assert.h>
```

On MS C compiler, can save the alignment state and set the alignment to 1 for the duration of the TPM_Types.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 TPM_Types.h and only include it for the #defines of the capabilities, properties, and command code values.

```
#pragma pack(push, 1)
#include "TPM_Types.h"
#pragma pack (pop)
#include "swap.h"
```

10.15.2 Typedefs

These defines are used primarily for sizing of the local response buffer.

```
#pragma pack(push,1)
typedef struct {
    TPM_ST          tag;
    UINT32          size;
    TPM_RC          code;
} HEADER;
typedef struct {
    UINT16      size;
    struct {
        UINT32      function;
        UINT32      line;
        UINT32      code;
    } values;
    TPM_RC      returnCode;
} GET_TEST_RESULT_PARAMETERS;
typedef struct {
    TPMI_YES_NO                 moreData;
    TPM_CAP                     capability; // Always TPM_CAP_TPM_PROPERTIES
    TPML_TAGGED_TPM_PROPERTY    tpmProperty; // a single tagged property
} GET_CAPABILITY_PARAMETERS;
typedef struct {
    HEADER header;
    GET_TEST_RESULT_PARAMETERS  getTestResult;
} TEST_RESPONSE;
typedef struct {
    HEADER header;
    GET_CAPABILITY_PARAMETERS   getCap;
} CAPABILITY_RESPONSE;
typedef union {
    TEST_RESPONSE           test;
    CAPABILITY_RESPONSE     cap;
} RESPONSES;
#pragma pack(pop)
```

Buffer to hold the responses. This may be a little larger than required due to padding that a compiler might add.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

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.

```
#ifndef __IGNORE_STATE__ // Don't define this value
static BYTE response[sizeof(RESPONSES)];
#endif
```

10.15.3 Local Functions

10.15.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```
static INT32
MarshalUint16(
    UINT16          integer,
    BYTE            **buffer
    )
{
    return UINT16_Marshal(&integer, buffer, NULL);
}
```

10.15.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```
static INT32
MarshalUint32(
    UINT32           integer,
    BYTE            **buffer
    )
{
    return UINT32_Marshal(&integer, buffer, NULL);
}
```

10.15.3.3 UnmarshalHeader()

Funtion to unmarshal the 10-byte command header.

```
static BOOL
UnmarshalHeader(
    HEADER          *header,
    BYTE            **buffer,
    INT32           *size
    )
{
    UINT32 usize;
    TPM_RC ucode;
    if(     UINT16_Unmarshal(&header->tag, buffer, size) != TPM_RC_SUCCESS
        ||  UINT32_Unmarshal(&usize, buffer, size) != TPM_RC_SUCCESS
        ||  UINT32_Unmarshal(&ucode, buffer, size) != TPM_RC_SUCCESS
        )
        return FALSE;
    header->size = usize;
    header->code = ucode;
    return TRUE;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

10.15.4 Public Functions

10.15.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```
78  LIB_EXPORT void
79  SetForceFailureMode(
80      void
81      )
82  {
83      g_forceFailureMode = TRUE;
84      return;
85  }
```

10.15.4.2 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().

```
86  void
87  TpmFail(
88      const char                  *function,
89      int line,             int     code
90      )
91  {
92      // Save the values that indicate where the error occurred.
93      // On a 64-bit machine, this may truncate the address of the string
94      // of the function name where the error occurred.
95      s_failFunction = *(UINT32*)&function;
96      s_failLine = line;
97      s_failCode = code;
98
99      // if asserts are enabled, then do an assert unless the failure mode code
100     // is being tested
101     assert(g_forceFailureMode);
102
103     // Clear this flag
104     g_forceFailureMode = FALSE;
105
106     // Jump to the failure mode code.
107     // Note: only get here if asserts are off or if we are testing failure mode
108     longjmp(&g_jumpBuffer[0], 1);
109 }
```

10.15.5 TpmFailureMode

This function is called by the interface code when the platform is in failure mode.

```
110 void
111 TpmFailureMode (
112     unsigned int     inRequestSize,     // IN: command buffer size
113     unsigned char   *inRequest,         // IN: command buffer
114     unsigned int    *outResponseSize,   // OUT: response buffer size
115     unsigned char   **outResponse       // OUT: response buffer
116     )
117 {
118     BYTE            *buffer;
119     UINT32           marshalSize;
120     UINT32           capability;
121     HEADER           header;    // unmarshaled command header
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
122         UINT32          pt;    // unmarshaled property type
123         UINT32          count; // unmarshaled property count
124
125         // If there is no command buffer, then just return TPM_RC_FAILURE
126         if(inRequestSize == 0 || inRequest == NULL)
127             goto FailureModeReturn;
128
129         // If the header is not correct for TPM2_GetCapability() or
130         // TPM2_GetTestResult() then just return the in failure mode response;
131         buffer = inRequest;
132         if(!UnmarshalHeader(&header, &inRequest, (INT32 *)&inRequestSize))
133             goto FailureModeReturn;
134         if(   header.tag  != TPM_ST_NO_SESSIONS
135            || header.size < 10)
136            goto FailureModeReturn;
137
138         switch (header.code) {
139         case TPM_CC_GetTestResult:
140
141             // make sure that the command size is correct
142             if(header.size != 10)
143                 goto FailureModeReturn;
144             buffer = &response[10];
145             marshalSize =  MarshalUint16(3 * sizeof(UINT32), &buffer);
146             marshalSize += MarshalUint32(s_failFunction, &buffer);
147             marshalSize += MarshalUint32(s_failLine, &buffer);
148             marshalSize += MarshalUint32(s_failCode, &buffer);
149             if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
150                 marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
151             else
152                 marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
153             break;
154
155         case TPM_CC_GetCapability:
156             // make sure that the size of the command is exactly the size
157             // returned for the capability, property, and count
158             if(     header.size!= (10 + (3 * sizeof(UINT32)))
159                 // also verify that this is requesting TPM properties
160                 ||      (UINT32_Unmarshal(&capability, &inRequest,
161                                           (INT32 *)&inRequestSize)
162                     !=  TPM_RC_SUCCESS)
163                 || (capability != TPM_CAP_TPM_PROPERTIES)
164                 ||      (UINT32_Unmarshal(&pt, &inRequest, (INT32 *)&inRequestSize)
165                     != TPM_RC_SUCCESS)
166                 ||      (UINT32_Unmarshal(&count, &inRequest, (INT32 *)&inRequestSize)
167                     != TPM_RC_SUCCESS)
168                 )
169
170                 goto FailureModeReturn;
171
172
173             // If in failure mode because of an unrecoverable read error, and the
174             // property is 0 and the count is 0, then this is an indication to
175             // re-manufacture the TPM. Do the re-manufacture but stay in failure
176             // mode until the TPM is reset.
177             // Note: this behavior is not required by ISO/IEC 11889 and it is
178             // OK to leave the TPM permanently bricked due to an unrecoverable NV
179             // error.
180             if( count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
181             {
182                 g_manufactured = FALSE;
183                 TPM_Manufacture(0);
184             }
185
186             if(count > 0)
187                 count = 1;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
188             else if(pt > TPM_PT_FIRMWARE_VERSION_2)
189                 count = 0;
190             if(pt < TPM_PT_MANUFACTURER)
191                 pt = TPM_PT_MANUFACTURER;
192
193             // set up for return
194             buffer = &response[10];
195             // if the request was for a PT less than the last one
196             // then we indicate more, otherwise, not.
197             if(pt < TPM_PT_FIRMWARE_VERSION_2)
198                 *buffer++ = YES;
199             else
200                 *buffer++ = NO;
201
202             marshalSize = 1;
203
204             // indicate the capability type
205             marshalSize += MarshalUint32(capability, &buffer);
206             // indicate the number of values that are being returned (0 or 1)
207             marshalSize += MarshalUint32(count, &buffer);
208             // indicate the property
209             marshalSize += MarshalUint32(pt, &buffer);
210
211             if(count > 0)
212                 switch (pt) {
213                 case TPM_PT_MANUFACTURER:
214                 // the vendor ID unique to each TPM manufacturer
215     #ifdef  MANUFACTURER
216                 pt = *(UINT32*)MANUFACTURER;
217     #else
218                 pt = 0;
219     #endif
220                 break;
221             case TPM_PT_VENDOR_STRING_1:
222                 // the first four characters of the vendor ID string
223     #ifdef  VENDOR_STRING_1
224                 pt = *(UINT32*)VENDOR_STRING_1;
225     #else
226                 pt = 0;
227     #endif
228                 break;
229             case TPM_PT_VENDOR_STRING_2:
230                 // the second four characters of the vendor ID string
231     #ifdef  VENDOR_STRING_2
232                 pt = *(UINT32*)VENDOR_STRING_2;
233     #else
234                 pt = 0;
235     #endif
236                 break;
237             case TPM_PT_VENDOR_STRING_3:
238                 // the third four characters of the vendor ID string
239     #ifdef  VENDOR_STRING_3
240                 pt = *(UINT32*)VENDOR_STRING_3;
241     #else
242                 pt = 0;
243     #endif
244                 break;
245             case TPM_PT_VENDOR_STRING_4:
246                 // the fourth four characters of the vendor ID string
247     #ifdef  VENDOR_STRING_4
248                 pt = *(UINT32*)VENDOR_STRING_4;
249     #else
250                 pt = 0;
251     #endif
252
253                 break;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
254             case TPM_PT_VENDOR_TPM_TYPE:
255                 // vendor-defined value indicating the TPM model
256                 // We just make up a number here
257                 pt = 1;
258                 break;
259             case TPM_PT_FIRMWARE_VERSION_1:
260                 // the more significant 32-bits of a vendor-specific value
261                 // indicating the version of the firmware
262     #ifdef  FIRMWARE_V1
263                 pt = FIRMWARE_V1;
264     #else
265                 pt = 0;
266     #endif
267                 break;
268             default: // TPM_PT_FIRMWARE_VERSION_2:
269                 // the less significant 32-bits of a vendor-specific value
270                 // indicating the version of the firmware
271     #ifdef  FIRMWARE_V2
272                 pt = FIRMWARE_V2;
273     #else
274                 pt = 0;
275     #endif
276                 break;
277             }
278             marshalSize += MarshalUint32(pt, &buffer);
279             break;
280         default: // default for switch (cc)
281             goto FailureModeReturn;
282         }
283         // Now do the header
284         buffer = response;
285         marshalSize = marshalSize + 10; // Add the header size to the
286                                         // stuff already marshaled
287         MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
288         MarshalUint32(marshalSize, &buffer); // responseSize
289         MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
290
291         *outResponseSize = marshalSize;
292         *outResponse = (unsigned char *)&response;
293         return;
294
295     FailureModeReturn:
296
297         buffer = response;
298
299         marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
300         marshalSize += MarshalUint32(10, &buffer);
301         marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
302
303         *outResponseSize = marshalSize;
304         *outResponse = (unsigned char *)response;
305         return;
306     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

11 Cryptographic Functions

11.1 Introduction

The files in clause 11 provide cryptographic support for the other functions in the TPM and the interface to the Crypto Engine.

Per the ISO/IEC 11889-1, clause 11.5, ”Authorization Subsystem” support for HMAC is mandatory and HMAC is defined in ISO/IEC 9797-2, making ISO/IEC 9797-2 indispensable for implementation of the required cryptographic functions for this International Standard.

Per the ISO/IEC 11889-1, clause 11.4.6.1, “Introduction” support for Cipher Feedback mode (CFB) is mandatory. CFB is defined ISO/IEC 10116:2006, making ISO/IEC 10116:2006 indispensable for implementation of the required cryptographic functions for this International Standard.

11.2 CryptUtil.c

11.2.1 Introduction

This module contains the interfaces to the CryptoEngine() and provides miscellaneous cryptographic functions in support of the TPM.

11.2.2 Includes

```
#include    "TPM_Types.h"
#include    "CryptoEngine.h"    // types shared by CryptUtil and CryptoEngine.
                                // Includes the function prototypes for the
                                // CryptoEngine functions.
#include    "Global.h"
#include    "InternalRoutines.h"
#include    "MemoryLib_fp.h"
//#include    "CryptSelfTest_fp.h"
```

11.2.3 TranslateCryptErrors()

This function converts errors from the cryptographic library into TPM_RC_VALUES.

Table 111

Error Returns	Meaning
TPM_RC_VALUE	CRYPT_FAIL
TPM_RC_NO_RESULT	CRYPT_NO_RESULT
TPM_RC_SCHEME	CRYPT_SCHEME
TPM_RC_VALUE	CRYPT_PARAMETER
TPM_RC_SIZE	CRYPT_UNDERFLOW
TPM_RC_ECC_POINT	CRYPT_POINT
TPM_RC_CANCELLED	CRYPT_CANCEL

```
static TPM_RC
TranslateCryptErrors (
    CRYPT_RESULT        retVal              // IN: crypt error to evaluate
)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
{
    switch (retVal)
    {
    case CRYPT_SUCCESS:
        return TPM_RC_SUCCESS;
    case CRYPT_FAIL:
        return TPM_RC_VALUE;
    case CRYPT_NO_RESULT:
        return TPM_RC_NO_RESULT;
    case CRYPT_SCHEME:
        return TPM_RC_SCHEME;
    case CRYPT_PARAMETER:
        return TPM_RC_VALUE;
    case CRYPT_UNDERFLOW:
        return TPM_RC_SIZE;
    case CRYPT_POINT:
        return TPM_RC_ECC_POINT;
    case CRYPT_CANCEL:
        return TPM_RC_CANCELED;
    default: // Other unknown warnings
        return TPM_RC_FAILURE;
    }
}
```

11.2.4 Random Number Generation Functions

11.2.4.1 Preamble

```
#ifdef TPM_ALG_NULL //%
#ifdef _DRBG_STATE_SAVE //%
```

11.2.4.2 CryptDrbgGetPutState()

Read or write the current state from the DRBG in the *cryptoEngine*.

```
void
CryptDrbgGetPutState(
    GET_PUT          direction      // IN: Get from or put to DRBG
    )
{
    _cpri__DrbgGetPutState(direction,
                           sizeof(go.drbgState),
                           (BYTE *)&go.drbgState);
}
#else   //% 00
//%#define CryptDrbgGetPutState(ignored)  // If not doing state save, turn this
//%                                        // into a null macro
#endif  //%
```

11.2.4.3 CryptStirRandom()

Stir random entropy

```
void
CryptStirRandom(
    UINT32           entropySize,   // IN: size of entropy buffer
    BYTE            *buffer         // IN: entropy buffer
    )
{
    // RNG self testing code may be inserted here

```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
59      // Call crypto engine random number stirring function
60      _cpri__StirRandom(entropySize, buffer);
61
62      return;
63  }
```

11.2.4.4 CryptGenerateRandom()

This is the interface to _cpri__GenerateRandom().

```
64  UINT16
65  CryptGenerateRandom(
66      UINT16           randomSize,    // IN: size of random number
67      BYTE            *buffer         // OUT: buffer of random number
68      )
69  {
70      UINT16              result;
71      pAssert(randomSize <= MAX_RSA_KEY_BYTES || randomSize <= PRIMARY_SEED_SIZE);
72      if(randomSize == 0)
73          return 0;
74
75      // Call crypto engine random number generation
76      result = _cpri__GenerateRandom(randomSize, buffer);
77      if(result != randomSize)
78          FAIL(FATAL_ERROR_INTERNAL);
79
80      return result;
81  }
82  #endif //TPM_ALG_NULL //%
```

11.2.5 Hash/HMAC Functions

11.2.5.1 CryptGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```
83  #ifdef TPM_ALG_KEYEDHASH        //% 1
84  TPM_ALG_ID
85  CryptGetContextAlg(
86      void            *state          // IN: the context to check
87      )
88  {
89      HASH_STATE  *context = (HASH_STATE *)state;
90      return _cpri__GetContextAlg(&context->state);
91  }
```

11.2.5.2 CryptStartHash()

This function starts a hash and return the size, in bytes, of the digest.

Table 112

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
92  UINT16
93  CryptStartHash(
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
94      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
95      HASH_STATE      *hashState       // OUT: the state of hash stack. It will be used
96                                       //      in hash update and completion
97      )
98  {
99      CRYPT_RESULT       retVal = 0;
100
101     pAssert(hashState != NULL);
102
103     TEST_HASH(hashAlg);
104
105     hashState->type = HASH_STATE_EMPTY;
106
107     // Call crypto engine start hash function
108     if((retVal = _cpri__StartHash(hashAlg, FALSE, &hashState->state)) > 0)
109         hashState->type = HASH_STATE_HASH;
110
111     return retVal;
112 }
```

11.2.5.3 CryptStartHashSequence()

Start a hash stack for a sequence object and return the size, in bytes, of the digest. This call uses the form of the hash state that requires context save and restored.

Table 113

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
113 UINT16
114 CryptStartHashSequence(
115     TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
116     HASH_STATE      *hashState       // OUT: the state of hash stack. It will be used
117                                      //      in hash update and completion
118     )
119 {
120     CRYPT_RESULT   retVal = 0;
121
122     pAssert(hashState != NULL);
123
124     TEST_HASH(hashAlg);
125
126     hashState->type = HASH_STATE_EMPTY;
127
128     // Call crypto engine start hash function
129     if((retVal = _cpri__StartHash(hashAlg, TRUE, &hashState->state)) > 0)
130         hashState->type = HASH_STATE_HASH;
131
132     return retVal;
133
134 }
```

11.2.5.4 CryptStartHMAC()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

Table 114

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
135  UINT16
136  CryptStartHMAC(
137      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
138      UINT16           keySize,        // IN: the size of HMAC key in bytes
139      BYTE            *key,            // IN: HMAC key
140      HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
141                                       //      in HMAC update and completion
142      )
143  {
144      HASH_STATE      *hashState = (HASH_STATE *)hmacState;
145      CRYPT_RESULT    retVal;
146
147      // This has to come before the pAssert in case we all calling this function
148      // during testing. If so, the first instance will have no arguments but the
149      // hash algorithm. The call from the test routine will have arguments. When
150      // the second call is done, then we return to the test dispatcher.
151      TEST_HASH(hashAlg);
152
153      pAssert(hashState != NULL);
154
155      hashState->type = HASH_STATE_EMPTY;
156
157      if((retVal =  _cpri__StartHMAC(hashAlg, FALSE, &hashState->state, keySize, key,
158                                     &hmacState->hmacKey.b)) > 0)
159          hashState->type = HASH_STATE_HMAC;
160
161      return retVal;
162  }
```

11.2.5.5 CryptStartHMACSequence()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

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.

This call is used to start a sequence HMAC that spans multiple TPM commands.

Table 115

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
163  UINT16
164  CryptStartHMACSequence(
165      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
166      UINT16           keySize,        // IN: the size of HMAC key in bytes
167      BYTE            *key,            // IN: HMAC key
168      HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
169                                       //      in HMAC update and completion
170      )
171  {
172      HASH_STATE      *hashState = (HASH_STATE *)hmacState;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
173         CRYPT_RESULT    retVal;
174
175         TEST_HASH(hashAlg);
176
177         hashState->type = HASH_STATE_EMPTY;
178
179         if((retVal =  _cpri__StartHMAC(hashAlg, TRUE, &hashState->state,
180                                        keySize, key, &hmacState->hmacKey.b)) > 0)
181             hashState->type = HASH_STATE_HMAC;
182
183         return retVal;
184     }
```

11.2.5.6 CryptStartHMAC2B()

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.

Table 116

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
185     LIB_EXPORT UINT16
186     CryptStartHMAC2B(
187         TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
188         TPM2B           *key,            // IN: HMAC key
189         HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
190                                          //      in HMAC update and completion
191         )
192     {
193         return CryptStartHMAC(hashAlg, key->size, key->buffer, hmacState);
194     }
```

11.2.5.7 CryptStartHMACSequence2B()

This function starts an HMAC sequence 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.

Table 117

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
195     UINT16
196     CryptStartHMACSequence2B(
197         TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
198         TPM2B           *key,            // IN: HMAC key
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
199         HMAC_STATE      *hmacState      // OUT: the state of HMAC stack. It will be used
200                                         //      in HMAC update and completion
201         )
202     {
203         return CryptStartHMACSequence(hashAlg, key->size, key->buffer, hmacState);
204     }
```

11.2.5.8 CryptUpdateDigest()

This function updates a digest (hash or HMAC) with an array of octets.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```
205     LIB_EXPORT void
206     CryptUpdateDigest(
207         void            *digestState,   // IN: the state of hash stack
208         UINT32           dataSize,      // IN: the size of data
209         BYTE            *data           // IN: data to be hashed
210         )
211     {
212         HASH_STATE      *hashState = (HASH_STATE *)digestState;
213
214         pAssert(digestState != NULL);
215
216         if(hashState->type != HASH_STATE_EMPTY && data != NULL && dataSize != 0)
217         {
218             // Call crypto engine update hash function
219             _cpri__UpdateHash(&hashState->state, dataSize, data);
220         }
221         return;
222     }
```

11.2.5.9 CryptUpdateDigest2B()

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.

```
223     LIB_EXPORT void
224     CryptUpdateDigest2B(
225         void            *digestState,   // IN: the digest state
226         TPM2B           *bIn            // IN: 2B containing the data
227         )
228     {
229         // Only compute the digest if a pointer to the 2B is provided.
230         // In CryptUpdateDigest(), if size is zero or buffer is NULL, then no change
231         // to the digest occurs. This function should not provide a buffer if bIn is
232         // not provided.
233         if(bIn != NULL)
234             CryptUpdateDigest(digestState, bIn->size, bIn->buffer);
235         return;
236     }
```

11.2.5.10 CryptUpdateDigestInt()

This function is used to include an integer value to a hash stack. The function marshals the integer into its canonical form before calling CryptUpdateHash().

```
237     LIB_EXPORT void
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
238  CryptUpdateDigestInt(
239      void            *state,         // IN: the state of hash stack
240      UINT32           intSize,       // IN: the size of 'intValue' in bytes
241      void            *intValue       // IN: integer value to be hashed
242      )
243  {
244
245  #if BIG_ENDIAN_TPM == YES
246      pAssert(    intValue != NULL && (intSize == 1 || intSize == 2
247              ||  intSize == 4 || intSize == 8));
248      CryptUpdateHash(state, inSize, (BYTE *)intValue);
249  #else
250
251      BYTE        marshalBuffer[8];
252      // Point to the big end of an little-endian value
253      BYTE        *p = &((BYTE *)intValue)[intSize - 1];
254      // Point to the big end of an big-endian value
255      BYTE        *q = marshalBuffer;
256
257      pAssert(intValue != NULL);
258      switch (intSize)
259      {
260      case 8:
261          *q++ = *p--;
262          *q++ = *p--;
263          *q++ = *p--;
264          *q++ = *p--;
265      case 4:
266          *q++ = *p--;
267          *q++ = *p--;
268      case 2:
269          *q++ = *p--;
270      case 1:
271          *q = *p;
272          // Call update the hash
273          CryptUpdateDigest(state, intSize, marshalBuffer);
274          break;
275      default:
276          FAIL(0);
277      }
278
279  #endif
280      return;
281  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

11.2.5.11 CryptCompleteHash()

This function completes a hash sequence and returns the digest.

This function can be called to complete either an HMAC or hash sequence. The state type determines if the context type is a hash or HMAC. If an HMAC, then the call is forwarded to CryptCompleteHash().

If **digestSize** is smaller than the digest size of hash/HMAC algorithm, the most significant bytes of required size will be returned.

Table 118

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
282  LIB_EXPORT UINT16
283  CryptCompleteHash(
284      void            *state,         // IN: the state of hash stack
```


© ISO/IEC 2015 – All rights reserved

```
285         UINT16           digestSize,    // IN: size of digest buffer
286         BYTE            *digest         // OUT: hash digest
287         )
288     {
289         HASH_STATE      *hashState = (HASH_STATE *)state;     // local value
290
291         // If the session type is HMAC, then could forward this to
292         // the HMAC processing and not cause an error. However, if no
293         // function calls this routine to forward it, then we can't get
294         // test coverage. The decision is to assert if this is called with
295         // the type == HMAC and fix anything that makes the wrong call.
296         pAssert(hashState->type == HASH_STATE_HASH);
297
298         // Set the state to empty so that it doesn't get used again
299         hashState->type = HASH_STATE_EMPTY;
300
301         // Call crypto engine complete hash function
302         return     _cpri__CompleteHash(&hashState->state, digestSize, digest);
303     }
```

11.2.5.12 CryptCompleteHash2B()

This function is the same as CypteCompleteHash() but the digest is placed in a TPM2B. This is the most common use and this is provided for clarity for this part of ISO/IEC 11889. 'digest.size' should be set to indicate the number of bytes to place in the buffer.

Table 119

Return Value	Meaning
>=0	the number of bytes placed in 'digest.buffer'

```
304     LIB_EXPORT UINT16
305     CryptCompleteHash2B(
306         void            *state,         // IN: the state of hash stack
307         TPM2B           *digest         // IN: the size of the buffer Out: requested
308                                         //     number of bytes
309         )
310     {
311         UINT16              retVal = 0;
312
313         if(digest != NULL)
314             retVal = CryptCompleteHash(state, digest->size, digest->buffer);
315
316         return retVal;
317     }
```

11.2.5.13 CryptHashBlock()

Hash a block of data and return the results. If the digest is larger than *retSize*, it is truncated and with the least significant octets dropped.

Table 120

Return Value	Meaning
>=0	the number of bytes placed in *ret*

```
318     LIB_EXPORT UINT16
319     CryptHashBlock(
320         TPM_ALG_ID       algId,         // IN: the hash algorithm to use
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
321         UINT16           blockSize,     // IN: size of the data block
322         BYTE            *block,         // IN: address of the block to hash
323         UINT16           retSize,       // IN: size of the return buffer
324         BYTE            *ret            // OUT: address of the buffer
325         )
326     {
327         TEST_HASH(algId);
328
329         return _cpri__HashBlock(algId, blockSize, block, retSize, ret);
330     }
```

11.2.5.14 CryptCompleteHMAC()

This function completes a HMAC sequence and returns the digest. If *digestSize* is smaller than the digest size of the HMAC algorithm, the most significant bytes of required size will be returned.

Table 121

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
331     LIB_EXPORT UINT16
332     CryptCompleteHMAC(
333         HMAC_STATE      *hmacState,     // IN: the state of HMAC stack
334         UINT32           digestSize,    // IN: size of digest buffer
335         BYTE            *digest         // OUT: HMAC digest
336         )
337     {
338         HASH_STATE      *hashState;
339
340         pAssert(hmacState != NULL);
341         hashState = &hmacState->hashState;
342
343         pAssert(hashState->type == HASH_STATE_HMAC);
344
345         hashState->type = HASH_STATE_EMPTY;
346
347         return _cpri__CompleteHMAC(&hashState->state, &hmacState->hmacKey.b,
348                                    digestSize, digest);
349
350     }
```

11.2.5.15 CryptCompleteHMAC2B()

This function is the same as CryptCompleteHMAC() but the HMAC result is returned in a TPM2B which is the most common use.

Table 122

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
351     LIB_EXPORT UINT16
352     CryptCompleteHMAC2B(
353         HMAC_STATE      *hmacState,     // IN: the state of HMAC stack
354         TPM2B           *digest         // OUT: HMAC
355         )
356     {
357         UINT16              retVal = 0;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
358         if(digest != NULL)
359             retVal = CryptCompleteHMAC(hmacState, digest->size, digest->buffer);
360         return retVal;
361     }
```

11.2.5.16 CryptHashStateImportExport()

This function is used to prepare a hash state context for LIB_EXPORT or to import it into the internal format. It is used by TPM2_ContextSave() and TPM2_ContextLoad() via SequenceDataImportExport(). This is just a pass-through function to the crypto library.

```
362     void
363     CryptHashStateImportExport(
364         HASH_STATE      *internalFmt,   // IN: state to LIB_EXPORT
365         HASH_STATE      *externalFmt,   // OUT: exported state
366         IMPORT_EXPORT    direction
367         )
368     {
369         _cpri__ImportExportHashState(&internalFmt->state,
370                                      (EXPORT_HASH_STATE *)&externalFmt->state,
371                                      direction);
372     }
```

11.2.5.17 CryptGetHashDigestSize()

This function returns the digest size in bytes for a hash algorithm.

Table 123

Return Value	Meaning
0	digest size for TPM_ALG_NULL
> 0	digest size

```
373     LIB_EXPORT UINT16
374     CryptGetHashDigestSize(
375         TPM_ALG_ID       hashAlg        // IN: hash algorithm
376         )
377     {
378         return _cpri__GetDigestSize(hashAlg);
379     }
```

11.2.5.18 CryptGetHashBlockSize()

Get the digest size in byte of a hash algorithm.

Table 124

Return Value	Meaning
0	block size for TPM_ALG_NULL
> 0	block size

```
380     LIB_EXPORT UINT16
381     CryptGetHashBlockSize(
382         TPM_ALG_ID       hash           // IN: hash algorithm to look up
383         )
384     {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
385         return _cpri__GetHashBlockSize(hash);
386     }
```

11.2.5.19 CryptGetHashAlgByIndex()

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* value of 2 will return the last implemented hash. All other index values will return TPM_ALG_NULL.

Table 125

Return Value	Meaning
TPM_ALG_xxx()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
387     LIB_EXPORT TPM_ALG_ID
388     CryptGetHashAlgByIndex(
389         UINT32          index           // IN: the index
390         )
391     {
392         return _cpri__GetHashAlgByIndex(index);
393     }
```

11.2.5.20 CryptSignHMAC()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

Table 126

Error Returns	Meaning

```
394     static TPM_RC
395     CryptSignHMAC(
396         OBJECT              *signKey,       // IN: HMAC key sign the hash
397         TPMT_SIG_SCHEME     *scheme,        // IN: signing scheme
398         TPM2B_DIGEST        *hashData,      // IN: hash to be signed
399         TPMT_SIGNATURE      *signature      // OUT: signature
400         )
401     {
402         HMAC_STATE          hmacState;
403         UINT32              digestSize;
404
405         // HMAC algorithm self testing code may be inserted here
406
407         digestSize = CryptStartHMAC2B(scheme->details.hmac.hashAlg,
408                                       &signKey->sensitive.sensitive.bits.b,
409                                       &hmacState);
410
411         // The hash algorithm must be a valid one.
412         pAssert(digestSize > 0);
413
414         CryptUpdateDigest2B(&hmacState, &hashData->b);
415
416         CryptCompleteHMAC(&hmacState, digestSize,
417                           (BYTE *) &signature->signature.hmac.digest);
418
419         // Set HMAC algorithm
420         signature->signature.hmac.hashAlg = scheme->details.hmac.hashAlg;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
421
422         return TPM_RC_SUCCESS;
423     }
```

11.2.5.21 CryptHMACVerifySignature()

This function will verify a signature signed by a HMAC key.

Table 127

Error Returns	Meaning
TPM_RC_SIGNATURE	if invalid input or signature is not genuine

```
424     static TPM_RC
425     CryptHMACVerifySignature(
426         OBJECT          *signKey,       // IN: HMAC key signed the hash
427         TPM2B_DIGEST    *hashData,      // IN: digest being verified
428         TPMT_SIGNATURE  *signature      // IN: signature to be verified
429         )
430     {
431         HMAC_STATE          hmacState;
432         TPM2B_DIGEST        digestToCompare;
433
434         digestToCompare.t.size = CryptStartHMAC2B(signature->signature.hmac.hashAlg,
435                                  &signKey->sensitive.sensitive.bits.b, &hmacState);
436
437         CryptUpdateDigest2B(&hmacState, &hashData->b);
438
439         CryptCompleteHMAC2B(&hmacState, &digestToCompare.b);
440
441         // Compare digest
442         if(MemoryEqual(digestToCompare.t.buffer,
443                        (BYTE *) &signature->signature.hmac.digest,
444                        digestToCompare.t.size))
445             return TPM_RC_SUCCESS;
446         else
447             return TPM_RC_SIGNATURE;
448
449     }
```

11.2.5.22 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Table 128

Error Returns	Meaning
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```
450     static TPM_RC
451     CryptGenerateKeyedHash(
452         TPMT_PUBLIC             *publicArea,        // IN/OUT: the public area template
453                                                     //     for the new key.
454         TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
455         TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
456         TPM_ALG_ID               kdfHashAlg,        // IN: algorithm for the KDF
457         TPM2B_SEED              *seed,              // IN: the seed
458         TPM2B_NAME              *name               // IN: name of the object
459         )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
460 {
461     TPMT_KEYEDHASH_SCHEME   *scheme;
462     TPM_ALG_ID               hashAlg;
463     UINT16                   hashBlockSize;
464 
465     scheme = &publicArea->parameters.keyedHashDetail.scheme;
466 
467     pAssert(publicArea->type == TPM_ALG_KEYEDHASH);
468 
469     // Pick the limiting hash algorithm
470     if(scheme->scheme == TPM_ALG_NULL)
471         hashAlg = publicArea->nameAlg;
472     else if(scheme->scheme == TPM_ALG_XOR)
473         hashAlg = scheme->details.xor.hashAlg;
474     else
475         hashAlg = scheme->details.hmac.hashAlg;
476     hashBlockSize =  CryptGetHashBlockSize(hashAlg);
477 
478     // if this is a signing or a decryption key, then then the limit
479     // for the data size is the block size of the hash. This limit
480     // is set because larger values have lower entropy because of the
481     // HMAC function.
482     if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR)
483     {
484         if(    (   publicArea->objectAttributes.decrypt
485                 ||  publicArea->objectAttributes.sign)
486            && sensitiveCreate->data.t.size > hashBlockSize)
487 
488             return TPM_RC_SIZE;
489     }
490     else
491     {
492         // If the TPM is going to generate the data, then set the size to be the
493         // size of the digest of the algorithm
494         sensitive->sensitive.sym.t.size = CryptGetHashDigestSize(hashAlg);
495         sensitiveCreate->data.t.size = 0;
496     }
497 
498     // Fill in the sensitive area
499     CryptGenerateNewSymmetric(sensitiveCreate, sensitive, kdfHashAlg,
500                               seed, name);
501 
502     // Create unique area in public
503     CryptComputeSymmetricUnique(publicArea->nameAlg,
504                                 sensitive, &publicArea->unique.sym);
505 
506     return TPM_RC_SUCCESS;
507 }
```

11.2.5.23 CryptKDFa()

This function generates a key using the KDFa() formulation in ISO/IEC 11889-1. In this implementation, this is a macro invocation of _cpri__KDFa() in the hash module of the CryptoEngine(). This macro sets *once* to FALSE so that KDFa() will iterate as many times as necessary to generate *sizeInBits* number of bits.

```
508 //%#define CryptKDFa(hashAlg, key, label, contextU, contextV,   \
509 //%                  sizeInBits, keyStream, counterInOut)      \
510 //%         TEST_HASH(hashAlg);                                \
511 //%        _cpri__KDFa(                                        \
512 //%                    ((TPM_ALG_ID)hashAlg),                  \
513 //%                    ((TPM2B *)key),                         \
514 //%                    ((const char *)label),                  \
515 //%                    ((TPM2B *)contextU),                    \
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
516 //%                    ((TPM2B *)contextV),                    \
517 //%                    ((UINT32)sizeInBits),                   \
518 //%                    ((BYTE *)keyStream),                    \
519 //%                    ((UINT32 *)counterInOut),               \
520 //%                    ((BOOL) FALSE)                          \
521 //%                   )
522 //%
```

11.2.5.24 CryptKDFaOnce()

This function generates a key using the KDFa() formulation in ISO/IEC 11889-1. In this implementation, this is a macro invocation of _cpri__KDFa() in the hash module of the CryptoEngine(). This macro will call _cpri__KDFa() with **once** TRUE so that only one iteration is performed, regardless of *sizeInBits*.

```
523 //%#define CryptKDFaOnce(hashAlg, key, label, contextU, contextV,    \
524 //%                      sizeInBits, keyStream, counterInOut)       \
525 //%         TEST_HASH(hashAlg);                                     \
526 //%        _cpri__KDFa(                                             \
527 //%                    ((TPM_ALG_ID)hashAlg),                       \
528 //%                    ((TPM2B *)key),                              \
529 //%                    ((const char *)label),                       \
530 //%                    ((TPM2B *)contextU),                         \
531 //%                    ((TPM2B *)contextV),                         \
532 //%                    ((UINT32)sizeInBits),                        \
533 //%                    ((BYTE *)keyStream),                         \
534 //%                    ((UINT32 *)counterInOut),                    \
535 //%                    ((BOOL) TRUE)                                \
536 //%                   )
537 //%
```

11.2.5.25 KDFa()

This function is used by functions outside of CryptUtil() to access _cpri_KDFa().

```
538 void
539 KDFa(
540     TPM_ALG_ID       hash,          // IN: hash algorithm used in HMAC
541     TPM2B           *key,           // IN: HMAC key
542     const char      *label,         // IN: a null-terminated label for KDF
543     TPM2B           *contextU,      // IN: context U
544     TPM2B           *contextV,      // IN: context V
545     UINT32           sizeInBits,    // IN: size of generated key in bits
546     BYTE            *keyStream,     // OUT: key buffer
547     UINT32          *counterInOut   // IN/OUT: caller may provide the iteration
548                                     //     counter for incremental operations to
549                                     //     avoid large intermediate buffers.
550     )
551 {
552     CryptKDFa(hash, key, label, contextU, contextV, sizeInBits,
553               keyStream, counterInOut);
554 }
```

11.2.5.26 CryptKDFe()

This function generates a key using the KDFa() formulation in ISO/IEC 11889. In this implementation, this is a macro invocation of _cpri__KDFe() in the hash module of the CryptoEngine().

```
555 //%#define CryptKDFe(hashAlg, Z, label, partyUInfo, partyVInfo,          \
556 //%                  sizeInBits, keyStream)                             \
557 //% TEST_HASH(hashAlg);                                                 \
558 //% _cpri__KDFe(                                                        \
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
559     //%             ((TPM_ALG_ID)hashAlg),                                \
560     //%             ((TPM2B *)Z),                                         \
561     //%             ((const char *)label),                                \
562     //%             ((TPM2B *)partyUInfo),                                \
563     //%             ((TPM2B *)partyVInfo),                                \
564     //%             ((UINT32)sizeInBits),                                 \
565     //%             ((BYTE *)keyStream)                                   \
566     //%             )
567     //%
568     #endif //TPM_ALG_KEYEDHASH    //% 1
```

11.2.6 RSA Functions

11.2.6.1 BuildRSA()

Function to set the cryptographic elements of an RSA key into a structure to simplify the interface to _cpri__ RSA function. This can/should be eliminated by building this structure into the object structure.

```
569     #ifdef TPM_ALG_RSA          //% 2
570     static void
571     BuildRSA(
572         OBJECT          *rsaKey,
573         RSA_KEY         *key
574         )
575     {
576         key->exponent = rsaKey->publicArea.parameters.rsaDetail.exponent;
577         if(key->exponent == 0)
578             key->exponent = RSA_DEFAULT_PUBLIC_EXPONENT;
579         key->publicKey = &rsaKey->publicArea.unique.rsa.b;
580
581         if(rsaKey->attributes.publicOnly || rsaKey->privateExponent.t.size == 0)
582             key->privateKey = NULL;
583         else
584             key->privateKey = &(rsaKey->privateExponent.b);
585     }
```

11.2.6.2 CryptTestKeyRSA()

This function provides the interface to _cpri__TestKeyRSA(). If both *p* and *q* are provided, *n* will be set to *p***q*.

If only *p* is provided, *q* is computed by *q* = *n*/*p*. If *n* mod *p* != 0, TPM_RC_BINDING is returned.

The key is validated by checking that a *d* can be found such that *e d* mod ((*p*-1)*(*q*-1)) = 1. If *d* is found that satisfies this requirement, it will be placed in *d*.

Table 129

Error Returns	Meaning
TPM_RC_BINDING	the public and private portions of the key are not matched

```
586     TPM_RC
587     CryptTestKeyRSA(
588         TPM2B           *d,             // OUT: receives the private exponent
589         UINT32           e,             // IN: public exponent
590         TPM2B           *n,             // IN/OUT: public modulus
591         TPM2B           *p,             // IN: a first prime
592         TPM2B           *q              // IN: an optional second prime
593         )
594     {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
595         CRYPT_RESULT    retVal;
596
597         TEST(ALG_NULL_VALUE);
598
599         pAssert(d != NULL && n != NULL && p != NULL);
600         // Set the exponent
601         if(e == 0)
602             e = RSA_DEFAULT_PUBLIC_EXPONENT;
603         // CRYPT_PARAMETER
604         retVal =_cpri__TestKeyRSA(d, e, n, p, q);
605         if(retVal == CRYPT_SUCCESS)
606             return TPM_RC_SUCCESS;
607         else
608             return TPM_RC_BINDING;  // convert CRYPT_PARAMETER
609     }
```

11.2.6.3 CryptGenerateKeyRSA()

This function is called to generate an RSA key from a provided seed. It calls _cpri__GenerateKeyRSA() to perform the computations. The implementation is vendor specific.

Table 130

Error Returns	Meaning
TPM_RC_RANGE	the exponent value is not supported
TPM_RC_CANCELLED	key generation has been canceled
TPM_RC_VALUE	exponent is not prime or is less than 3; or could not find a prime using the provided parameters

```
610     static TPM_RC
611     CryptGenerateKeyRSA(
612         TPMT_PUBLIC         *publicArea,        // IN/OUT: The public area template for
613                                                 //      the new key. The public key
614                                                 //      area will be replaced by the
615                                                 //      product of two primes found by
616                                                 //      this function
617         TPMT_SENSITIVE      *sensitive,         // OUT: the sensitive area will be
618                                                 //      updated to contain the first
619                                                 //      prime and the symmetric
620                                                 //      encryption key
621         TPM_ALG_ID           hashAlg,           // IN: the hash algorithm for the KDF
622         TPM2B_SEED          *seed,              // IN: Seed for the creation
623         TPM2B_NAME          *name,              // IN: Object name
624         UINT32              *counter            // OUT: last iteration of the counter
625     )
626     {
627         CRYPT_RESULT    retVal;
628         UINT32          exponent = publicArea->parameters.rsaDetail.exponent;
629
630         TEST_HASH(hashAlg);
631         TEST(ALG_NULL_VALUE);
632
633         // In this implementation, only the default exponent is allowed
634         if(exponent != 0 && exponent != RSA_DEFAULT_PUBLIC_EXPONENT)
635             return TPM_RC_RANGE;
636         exponent = RSA_DEFAULT_PUBLIC_EXPONENT;
637
638         *counter = 0;
639
640         // _cpri_GenerateKeyRSA can return CRYPT_CANCEL or CRYPT_FAIL
641         retVal = _cpri__GenerateKeyRSA(&publicArea->unique.rsa.b,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
642                                     &sensitive->sensitive.rsa.b,
643                                     publicArea->parameters.rsaDetail.keyBits,
644                                     exponent,
645                                     hashAlg,
646                                     &seed->b,
647                                     "RSA key by vendor",
648                                     &name->b,
649                                     counter);
650
651     // CRYPT_CANCEL -> TPM_RC_CANCELLED; CRYPT_FAIL -> TPM_RC_VALUE
652     return TranslateCryptErrors(retVal);
653
654 }
```

11.2.6.4 CryptLoadPrivateRSA()

This function is called to generate the private exponent of an RSA key. It uses CryptTestKeyRSA().

Table 131

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of *rsaKey* are not matched

```
655 TPM_RC
656 CryptLoadPrivateRSA(
657     OBJECT          *rsaKey         // IN: the RSA key object
658     )
659 {
660     TPM_RC           result;
661     TPMT_PUBLIC     *publicArea = &rsaKey->publicArea;
662     TPMT_SENSITIVE  *sensitive = &rsaKey->sensitive;
663
664     // Load key by computing the private exponent
665     // TPM_RC_BINDING
666     result = CryptTestKeyRSA(&(rsaKey->privateExponent.b),
667                              publicArea->parameters.rsaDetail.exponent,
668                              &(publicArea->unique.rsa.b),
669                              &(sensitive->sensitive.rsa.b),
670                              NULL);
671     if(result == TPM_RC_SUCCESS)
672         rsaKey->attributes.privateExp = SET;
673
674     return result;
675 }
```

11.2.6.5 CryptSelectRSAScheme()

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.

```
676 TPMT_RSA_DECRYPT*
677 CryptSelectRSAScheme(
678     TPMI_DH_OBJECT       rsaHandle,     // IN: handle of sign key
679     TPMT_RSA_DECRYPT    *scheme         // IN: a sign or decrypt scheme
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
680         )
681     {
682         OBJECT              *rsaObject;
683         TPMT_ASYM_SCHEME    *keyScheme;
684         TPMT_RSA_DECRYPT    *retVal = NULL;
685
686         // Get sign object pointer
687         rsaObject = ObjectGet(rsaHandle);
688         keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
689
690         // if the default scheme of the object is TPM_ALG_NULL, then select the
691         // input scheme
692         if(keyScheme->scheme == TPM_ALG_NULL)
693         {
694             retVal = scheme;
695         }
696         // if the object scheme is not TPM_ALG_NULL and the input scheme is
697         // TPM_ALG_NULL, then select the default scheme of the object.
698         else if(scheme->scheme == TPM_ALG_NULL)
699         {
700             // if input scheme is NULL
701             retVal = (TPMT_RSA_DECRYPT *)keyScheme;
702         }
703         // get here if both the object scheme and the input scheme are
704         // not TPM_ALG_NULL. Need to insure that they are the same.
705         // IMPLEMENTATION NOTE: This could cause problems if future versions have
706         // schemes that have more values than just a hash algorithm. A new function
707         // (IsSchemeSame()) might be needed then.
708         else if(   keyScheme->scheme == scheme->scheme
709                 && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
710         {
711             retVal = scheme;
712         }
713         // two different, incompatible schemes specified will return NULL
714         return retVal;
715     }
```

11.2.6.6 CryptDecryptRSA()

This function is the interface to _cpri__DecryptRSA(). It handles the return codes from that function and converts them from CRYPT_RESULT to TPM_RC values. The *rsaKey* parameter must reference an RSA decryption key.

Table 132

Error Returns	Meaning
TPM_RC_BINDING	Public and private parts of the key are not cryptographically bound.
TPM_RC_SIZE	Size of data to decrypt is not the same as the key size.
TPM_RC_VALUE	Numeric value of the encrypted data is greater than the public exponent, or output buffer is too small for the decrypted message.

```
716     TPM_RC
717     CryptDecryptRSA(
718         UINT16              *dataOutSize,   // OUT: size of plain text in bytes
719         BYTE                *dataOut,       // OUT: plain text
720         OBJECT              *rsaKey,        // IN: internal RSA key
721         TPMT_RSA_DECRYPT    *scheme,        // IN: selects the padding scheme
722         UINT16               cipherInSize,  // IN: size of cipher text  in byte
723         BYTE                *cipherIn,      // IN: cipher text
724         const char          *label          // IN: a label, when needed
725         )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
{
    RSA_KEY         key;
    CRYPT_RESULT    retVal = CRYPT_SUCCESS;
    UINT32          dSize;                  // Place to put temporary value for the
                                            // returned data size
    TPMI_ALG_HASH   hashAlg = TPM_ALG_NULL; // hash algorithm in the selected
                                            // padding scheme
    TPM_RC          result = TPM_RC_SUCCESS;

    // pointer checks
    pAssert(    (dataOutSize != NULL) && (dataOut != NULL)
            &&  (rsaKey != NULL) && (cipherIn != NULL));

    // The public type is a RSA decrypt key
    pAssert(    (rsaKey->publicArea.type == TPM_ALG_RSA
            &&  rsaKey->publicArea.objectAttributes.decrypt == SET));

    // Must have the private portion loaded.  This check is made before this
    // function is called.
    pAssert(rsaKey->attributes.publicOnly == CLEAR);

    // decryption requires that the private modulus be present
    if(rsaKey->attributes.privateExp == CLEAR)
    {

        // Load key by computing the private exponent
        // CryptLoadPrivateRSA may return TPM_RC_BINDING
        result = CryptLoadPrivateRSA(rsaKey);
    }

    // the input buffer must be the size of the key
    if(result == TPM_RC_SUCCESS)
    {
        if(cipherInSize != rsaKey->publicArea.unique.rsa.t.size)
            result = TPM_RC_SIZE;
        else
        {
            BuildRSA(rsaKey, &key);

            // Initialize the dOutSize parameter
            dSize = *dataOutSize;

            // For OAEP scheme, initialize the hash algorithm for padding
            if(scheme->scheme == TPM_ALG_OAEP)
            {
                hashAlg = scheme->details.oaep.hashAlg;
                TEST_HASH(hashAlg);
            }
            // See if the padding mode needs to be tested
            TEST(scheme->scheme);

            // _cpri__DecryptRSA may return CRYPT_PARAMETER CRYPT_FAIL CRYPT_SCHEME
            retVal = _cpri__DecryptRSA(&dSize, dataOut, &key, scheme->scheme,
                                       cipherInSize, cipherIn, hashAlg, label);

            // Scheme must have been validated when the key was loaded/imported
            pAssert(retVal != CRYPT_SCHEME);

            // Set the return size
            pAssert(dSize <= UINT16_MAX);
            *dataOutSize = (UINT16)dSize;

            // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_FAIL -> TPM_RC_VALUE
            result = TranslateCryptErrors(retVal);
        }
    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
792         return result;
793     }
```

11.2.6.7 CryptEncryptRSA()

This function provides the interface to _cpri__EncryptRSA(). The object referenced by **rsaKey** is required to be an RSA decryption key.

Table 133

Error Returns	Meaning
TPM_RC_SCHEME	*scheme* is not supported
TPM_RC_VALUE	numeric value of *dataIn* is greater than the key modulus

```
794     TPM_RC
795     CryptEncryptRSA(
796         UINT16              *cipherOutSize, // OUT: size of cipher text in byte
797         BYTE                *cipherOut,     // OUT: cipher text
798         OBJECT              *rsaKey,        // IN: internal RSA key
799         TPMT_RSA_DECRYPT    *scheme,        // IN: selects the padding scheme
800         UINT16               dataInSize,    // IN: size of plain text in byte
801         BYTE                *dataIn,        // IN: plain text
802         const char          *label          // IN: an optional label
803         )
804     {
805         RSA_KEY              key;
806         CRYPT_RESULT         retVal;
807         UINT32               cOutSize;              // Conversion variable
808         TPMI_ALG_HASH        hashAlg = TPM_ALG_NULL;    // hash algorithm in selected
809                                                         // padding scheme
810
811         // must have a pointer to a key and some data to encrypt
812         pAssert(rsaKey != NULL && dataIn != NULL);
813
814         // The public type is a RSA decryption key
815         pAssert(   rsaKey->publicArea.type == TPM_ALG_RSA
816                 && rsaKey->publicArea.objectAttributes.decrypt == SET);
817
818         // If the cipher buffer must be provided and it must be large enough
819         // for the result
820         pAssert(   cipherOut != NULL
821                 && cipherOutSize != NULL
822                 && *cipherOutSize >= rsaKey->publicArea.unique.rsa.t.size);
823
824         // Only need the public key and exponent for encryption
825         BuildRSA(rsaKey, &key);
826
827         // Copy the size to the conversion buffer
828         cOutSize = *cipherOutSize;
829
830         // For OAEP scheme, initialize the hash algorithm for padding
831         if(scheme->scheme == TPM_ALG_OAEP)
832         {
833             hashAlg = scheme->details.oaep.hashAlg;
834             TEST_HASH(hashAlg);
835         }
836
837         // This is a public key operation and does not require that the private key
838         // be loaded. To verify this, need to do the full algorithm
839         TEST(scheme->scheme);
840
841         // Encrypt the data with the public exponent
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
842         // _cpri__EncryptRSA may return CRYPT_PARAMETER or CRYPT_SCHEME
843         retVal = _cpri__EncryptRSA(&cOutSize,cipherOut, &key, scheme->scheme,
844                                    dataInSize, dataIn, hashAlg, label);
845
846         pAssert (cOutSize <= UINT16_MAX);
847         *cipherOutSize = (UINT16)cOutSize;
848         // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_SCHEME -> TPM_RC_SCHEME
849         return TranslateCryptErrors(retVal);
850     }
```

11.2.6.8 CryptSignRSA()

This function is used to sign a digest with an RSA signing key.

Table 134

Error Returns	Meaning
TPM_RC_BINDING	public and private part of *signKey* are not properly bound
TPM_RC_SCHEME	*scheme* is not supported
TPM_RC_VALUE	*hashData* is larger than the modulus of *signKey*, or the size of *hashData* does not match hash algorithm in *scheme*

```
851     static TPM_RC
852     CryptSignRSA(
853         OBJECT              *signKey,       // IN: RSA key signs the hash
854         TPMT_SIG_SCHEME     *scheme,        // IN: sign scheme
855         TPM2B_DIGEST        *hashData,      // IN: hash to be signed
856         TPMT_SIGNATURE      *sig            // OUT: signature
857         )
858     {
859         UINT32               signSize;
860         RSA_KEY              key;
861         CRYPT_RESULT         retVal;
862         TPM_RC               result = TPM_RC_SUCCESS;
863
864         pAssert(    (signKey != NULL) && (scheme != NULL)
865                     && (hashData != NULL) && (sig != NULL));
866
867         // assume that the key has private part loaded and that it is a signing key.
868         pAssert(   (signKey->attributes.publicOnly == CLEAR)
869                 && (signKey->publicArea.objectAttributes.sign == SET));
870
871         // check if the private exponent has been computed
872         if(signKey->attributes.privateExp == CLEAR)
873             // May return TPM_RC_BINDING
874             result = CryptLoadPrivateRSA(signKey);
875
876         if(result == TPM_RC_SUCCESS)
877         {
878             BuildRSA(signKey, &key);
879
880             // Make sure that the hash is tested
881             TEST_HASH(sig->signature.any.hashAlg);
882
883             // Run a test of the RSA sign
884             TEST(scheme->scheme);
885
886             // _crypi__SignRSA can return CRYPT_SCHEME and CRYPT_PARAMETER
887             retVal = _cpri__SignRSA(&signSize,
888                                     sig->signature.rsassa.sig.t.buffer,
889                                     &key,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
890                                     sig->sigAlg,
891                                     sig->signature.any.hashAlg,
892                                     hashData->t.size, hashData->t.buffer);
893         pAssert(signSize <= UINT16_MAX);
894         sig->signature.rsassa.sig.t.size = (UINT16)signSize;
895
896         // CRYPT_SCHEME -> TPM_RC_SCHEME; CRYPT_PARAMTER -> TPM_RC_VALUE
897         result = TranslateCryptErrors(retVal);
898     }
899     return result;
900 }
```

11.2.6.9 CryptRSAVerifySignature()

This function is used to verify signature signed by a RSA key.

Table 135

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not genuine
TPM_RC_SCHEME	signature scheme not supported

```
901 static TPM_RC
902 CryptRSAVerifySignature(
903     OBJECT          *signKey,       // IN: RSA key signed the hash
904     TPM2B_DIGEST    *digestData,    // IN: digest being signed
905     TPMT_SIGNATURE  *sig            // IN: signature to be verified
906     )
907 {
908     RSA_KEY             key;
909     CRYPT_RESULT        retVal;
910     TPM_RC              result;
911
912     // Validate parameter assumptions
913     pAssert((signKey != NULL) && (digestData != NULL) && (sig != NULL));
914
915     TEST_HASH(sig->signature.any.hashAlg);
916     TEST(sig->sigAlg);
917
918     // This is a public-key-only operation
919     BuildRSA(signKey, &key);
920
921     // Call crypto engine to verify signature
922     // _cpri_ValidateSignaturRSA may return CRYPT_FAIL or CRYPT_SCHEME
923     retVal = _cpri__ValidateSignatureRSA(&key,
924                                          sig->sigAlg,
925                                          sig->signature.any.hashAlg,
926                                          digestData->t.size,
927                                          digestData->t.buffer,
928                                          sig->signature.rsassa.sig.t.size,
929                                          sig->signature.rsassa.sig.t.buffer,
930                                          0);
931     // _cpri__ValidateSignatureRSA can return CRYPT_SUCCESS, CRYPT_FAIL, or
932     // CRYPT_SCHEME. Translate CRYPT_FAIL to TPM_RC_SIGNATURE
933     if(retVal == CRYPT_FAIL)
934         result = TPM_RC_SIGNATURE;
935     else
936         // CRYPT_SCHEME -> TPM_RC_SCHEME
937         result = TranslateCryptErrors(retVal);
938
939     return result;
940 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
941 #endif //TPM_ALG_RSA      //% 2
```

11.2.7 ECC Functions

11.2.7.1 CryptEccGetCurveDataPointer()

This function returns a pointer to an ECC_CURVE_VALUES structure that contains the parameters for the key size and schemes for a given curve.

```
942 #ifdef TPM_ALG_ECC //% 3
943 static const ECC_CURVE    *
944 CryptEccGetCurveDataPointer(
945     TPM_ECC_CURVE    curveID        // IN: id of the curve
946     )
947 {
948     return _cpri__EccGetParametersByCurveId(curveID);
949 }
```

11.2.7.2 CryptEccGetKeySizeInBits()

This function returns the size in bits of the key associated with a curve.

```
950 UINT16
951 CryptEccGetKeySizeInBits(
952     TPM_ECC_CURVE    curveID        // IN: id of the curve
953     )
954 {
955     const ECC_CURVE         *curve = CryptEccGetCurveDataPointer(curveID);
956     UINT16                   keySizeInBits = 0;
957
958     if(curve != NULL)
959         keySizeInBits = curve->keySizeBits;
960
961     return keySizeInBits;
962 }
```

11.2.7.3 CryptEccGetKeySizeBytes()

This macro returns the size of the ECC key in bytes. It uses CryptEccGetKeySizeInBits().

```
963 // The next lines will be placed in CyrptUtil_fp.h with the //% removed
964 //% #define CryptEccGetKeySizeInBytes(curve)          \
965 //%     ((CryptEccGetKeySizeInBits(curve)+7)/8)
```

11.2.7.4 CryptEccGetParameter()

This function returns a pointer to an ECC curve parameter. The parameter is selected by a single character designator from the set of {pnabxyh}.

```
966 LIB_EXPORT const TPM2B *
967 CryptEccGetParameter(
968     char             p,             // IN: the parameter selector
969     TPM_ECC_CURVE    curveId        // IN: the curve id
970     )
971 {
972     const ECC_CURVE     *curve = _cpri__EccGetParametersByCurveId(curveId);
973     const TPM2B         *parameter = NULL;
974
975     if(curve != NULL)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
976         {
977             switch (p)
978             {
979             case 'p':
980                 parameter = curve->curveData->p;
981                 break;
982             case 'n':
983                 parameter =  curve->curveData->n;
984                 break;
985             case 'a':
986                 parameter =  curve->curveData->a;
987                 break;
988             case 'b':
989                 parameter =  curve->curveData->b;
990                 break;
991             case 'x':
992                 parameter =  curve->curveData->x;
993                 break;
994             case 'y':
995                 parameter =  curve->curveData->y;
996                 break;
997             case 'h':
998                 parameter =  curve->curveData->h;
999                 break;
1000            default:
1001                break;
1002            }
1003        }
1004        return parameter;
1005    }
```

11.2.7.5 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```
1006    const TPMT_ECC_SCHEME *
1007    CryptGetCurveSignScheme(
1008        TPM_ECC_CURVE    curveId     // IN: The curve selector
1009        )
1010    {
1011        const ECC_CURVE        *curve = _cpri__EccGetParametersByCurveId(curveId);
1012        const TPMT_ECC_SCHEME  *scheme = NULL;
1013
1014        if(curve != NULL)
1015            scheme =  &(curve->sign);
1016        return scheme;
1017    }
```

11.2.7.6 CryptEccIsPointOnCurve()

This function will validate that an ECC point is on the curve of given *curveID*.

Table 136

Return Value	Meaning
TRUE	if the point is on curve
FALSE	if the point is not on curve

```
1018    BOOL
1019    CryptEccIsPointOnCurve(
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
1020        TPM_ECC_CURVE    curveID,       // IN: ECC curve ID
1021        TPMS_ECC_POINT  *Q              // IN: ECC point
1022        )
1023    {
1024        // Make sure that point multiply is working
1025        TEST(TPM_ALG_ECC);
1026        // Check point on curve logic by seeing if the test key is on the curve
1027
1028        // Call crypto engine function to check if a ECC public point is on the
1029        // given curve
1030        if(_cpri__EccIsPointOnCurve(curveID, Q))
1031            return TRUE;
1032        else
1033            return FALSE;
1034    }
```

11.2.7.7 CryptNewEccKey()

This function creates a random ECC key that is not derived from other parameters as is a Primary Key.

```
1035    TPM_RC
1036    CryptNewEccKey(
1037        TPM_ECC_CURVE            curveID,       // IN: ECC curve
1038        TPMS_ECC_POINT          *publicPoint,   // OUT: public point
1039        TPM2B_ECC_PARAMETER     *sensitive      // OUT: private area
1040        )
1041    {
1042        TPM_RC              result = TPM_RC_SUCCESS;
1043        // _cpri__GetEphemeralECC may return CRYPT_PARAMETER
1044        if(_cpri__GetEphemeralEcc(publicPoint, sensitive, curveID) != CRYPT_SUCCESS)
1045            // Something is wrong with the key.
1046            result = TPM_RC_KEY;
1047
1048        return result;
1049    }
```

11.2.7.8 CryptEccPointMultiply()

This function is used to perform a point multiply *R* = [*d*]*Q*. If *Q* is not provided, the multiplication is performed using the generator point of the curve.

Table 137

xError Returns	Meaning
TPM_RC_ECC_POINT	invalid optional ECC point *pIn*
TPM_RC_NO_RESULT	multiplication resulted in a point at infinity
TPM_RC_CANCELED	if a self-test was done, it might have been aborted

```
1050    TPM_RC
1051    CryptEccPointMultiply(
1052        TPMS_ECC_POINT          *pOut,          // OUT: output point
1053        TPM_ECC_CURVE            curveId,       // IN: curve selector
1054        TPM2B_ECC_PARAMETER     *dIn,           // IN: public scalar
1055        TPMS_ECC_POINT          *pIn            // IN: optional point
1056        )
1057    {
1058        TPM2B_ECC_PARAMETER     *n = NULL;
1059        CRYPT_RESULT            retVal;
1060
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1061        pAssert(pOut != NULL && dIn != NULL);
1062
1063        if(pIn != NULL)
1064        {
1065            n = dIn;
1066            dIn = NULL;
1067        }
1068        // Do a test of point multiply
1069        TEST(TPM_ALG_ECC);
1070
1071        // _cpri__EccPointMultiply may return CRYPT_POINT or CRYPT_NO_RESULT
1072        retVal = _cpri__EccPointMultiply(pOut, curveId, dIn, pIn, n);
1073
1074        // CRYPT_POINT->TPM_RC_ECC_POINT and CRYPT_NO_RESULT->TPM_RC_NO_RESULT
1075        return TranslateCryptErrors(retVal);
1076    }
```

11.2.7.9 CryptGenerateKeyECC()

This function generates an ECC key from a seed value.

The method here may not work for objects that have an order (*G*) that with a different size than a private key.

Table 138

Error Returns	Meaning
TPM_RC_VALUE	hash algorithm is not supported

```
1077    static TPM_RC
1078    CryptGenerateKeyECC(
1079        TPMT_PUBLIC     *publicArea,    // IN/OUT: The public area template for the new
1080                                        //     key.
1081        TPMT_SENSITIVE  *sensitive,     // IN/OUT: the sensitive area
1082        TPM_ALG_ID       hashAlg,       // IN: algorithm for the KDF
1083        TPM2B_SEED      *seed,          // IN: the seed value
1084        TPM2B_NAME      *name,          // IN: the name of the object
1085        UINT32          *counter        // OUT: the iteration counter
1086        )
1087    {
1088        CRYPT_RESULT        retVal;
1089
1090        TEST_HASH(hashAlg);
1091        TEST(ALG_ECDSA_VALUE); // ECDSA is used to verify each key
1092
1093        // The iteration counter has no meaning for ECC key generation. The parameter
1094        // will be overloaded for those implementations that have a requirement for
1095        // doing pair-wise consistency checks on signing keys. If the counter parameter
1096        // is 0 or NULL, then no consistency check is done. If it is other than 0, then
1097        // a consistency check is run. This modification allow this code to work with
1098        // the existing versions of the CrytpoEngine and with FIPS-compliant versions
1099        // as well.
1100        *counter = (UINT32)(publicArea->objectAttributes.sign == SET);
1101
1102        // _cpri__GenerateKeyEcc only has one error return (CRYPT_PARAMETER) which means
1103        // that the hash algorithm is not supported. This should not be possible
1104        retVal = _cpri__GenerateKeyEcc(&publicArea->unique.ecc,
1105                                       &sensitive->sensitive.ecc,
1106                                       publicArea->parameters.eccDetail.curveID,
1107                                       hashAlg, &seed->b, "ECC key by vendor",
1108                                       &name->b, counter);
1109        // This will only be useful if _cpri__GenerateKeyEcc return CRYPT_CANCEL
1110        return TranslateCryptErrors(retVal);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
}
```

11.2.7.10 CryptSignECC()

This function is used for ECC signing operations. If the signing scheme is a split scheme, and the signing operation is successful, the commit value is retired.

Table 139

Error Returns	Meaning
TPM_RC_SCHEME	unsupported *scheme*
TPM_RC_VALUE	invalid commit status (in case of a split scheme) or failed to generate r value.

```
static TPM_RC
CryptSignECC(
    OBJECT              *signKey,       // IN: ECC key to sign the hash
    TPMT_SIG_SCHEME     *scheme,        // IN: sign scheme
    TPM2B_DIGEST        *hashData,      // IN: hash to be signed
    TPMT_SIGNATURE      *signature      // OUT: signature
    )
{
    TPM2B_ECC_PARAMETER      r;
    TPM2B_ECC_PARAMETER     *pr = NULL;
    CRYPT_RESULT             retVal;

    // Run a test of the ECC sign and verify if it has not already been run
    TEST_HASH(scheme->details.any.hashAlg);
    TEST(scheme->scheme);

    if(CryptIsSplitSign(scheme->scheme))
    {
        // When this code was written, the only split scheme was ECDAA
        // (which can also be used for U-Prove).
        if(!CryptGenerateR(&r,
                           &scheme->details.ecdaa.count,
                           signKey->publicArea.parameters.eccDetail.curveID,
                           &signKey->name))
            return TPM_RC_VALUE;
        pr = &r;
    }
    // Call crypto engine function to sign
    // _cpri__SignEcc may return CRYPT_SCHEME
    retVal = _cpri__SignEcc(&signature->signature.ecdsa.signatureR,
                            &signature->signature.ecdsa.signatureS,
                            scheme->scheme,
                            scheme->details.any.hashAlg,
                            signKey->publicArea.parameters.eccDetail.curveID,
                            &signKey->sensitive.sensitive.ecc,
                            &hashData->b,
                            pr
                            );
    if(CryptIsSplitSign(scheme->scheme) && retVal == CRYPT_SUCCESS)
        CryptEndCommit(scheme->details.ecdaa.count);
    // CRYPT_SCHEME->TPM_RC_SCHEME
    return TranslateCryptErrors(retVal);
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

11.2.7.11 CryptECCVerifySignature()

This function is used to verify a signature created with an ECC key.

Table 140

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not valid
TPM_RC_SCHEME	the signing scheme or *hashAlg* is not supported

```
1155    static TPM_RC
1156    CryptECCVerifySignature(
1157        OBJECT          *signKey,       // IN: ECC key signed the hash
1158        TPM2B_DIGEST    *digestData,    // IN: digest being signed
1159        TPMT_SIGNATURE  *signature      // IN: signature to be verified
1160        )
1161    {
1162        CRYPT_RESULT        retVal;
1163
1164        TEST_HASH(signature->signature.any.hashAlg);
1165        TEST(signature->sigAlg);
1166
1167        // This implementation uses the fact that all the defined ECC signing
1168        // schemes have the hash as the first parameter.
1169        // _cpriValidateSignatureEcc may return CRYPT_FAIL or CRYP_SCHEME
1170        retVal = _cpri__ValidateSignatureEcc(&signature->signature.ecdsa.signatureR,
1171                                             &signature->signature.ecdsa.signatureS,
1172                                             signature->sigAlg,
1173                                             signature->signature.any.hashAlg,
1174                                             signKey->publicArea.parameters.eccDetail.curveID,
1175                                             &signKey->publicArea.unique.ecc,
1176                                             &digestData->b);
1177        if(retVal == CRYPT_FAIL)
1178            return TPM_RC_SIGNATURE;
1179        // CRYPT_SCHEME->TPM_RC_SCHEME
1180        return TranslateCryptErrors(retVal);
1181    }
```

11.2.7.12 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 141

Return Value	Meaning
TRUE	r value computed
FALSE	no r value computed

```
1182    BOOL
1183    CryptGenerateR(
1184        TPM2B_ECC_PARAMETER     *r,             // OUT: the generated random value
1185        UINT16                  *c,             // IN/OUT: count value.
1186        TPMI_ECC_CURVE           curveID,       // IN: the curve for the value
1187        TPM2B_NAME              *name           // IN: optional name of a key to
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1188                                            //     associate with 'r'
1189        )
1190    {
1191        // This holds the marshaled g_commitCounter.
1192        TPM2B_TYPE(8B, 8);
1193        TPM2B_8B                cntr = {8,{0}};
1194
1195        UINT32                   iterations;
1196        const TPM2B             *n;
1197        UINT64                   currentCount = gr.commitCounter;
1198        // This is just to suppress a compiler warning about a conditional expression
1199        // being a constant. This is because of the macro expansion of ryptKDFa
1200        TPMI_ALG_HASH            hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
1201
1202        n =  CryptEccGetParameter('n', curveID);
1203        pAssert(r != NULL && n != NULL);
1204
1205        // If this is the commit phase, use the current value of the commit counter
1206        if(c != NULL)
1207        {
1208
1209            UINT16      t1;
1210            // if the array bit is not set, can't use the value.
1211            if(!BitIsSet((*c & COMMIT_INDEX_MASK), gr.commitArray,
1212                         sizeof(gr.commitArray)))
1213                return FALSE;
1214
1215            // If it is the sign phase, figure out what the counter value was
1216            // when the commitment was made.
1217            //
1218            // When gr.commitArray has less than 64K bits, the extra
1219            // bits of 'c' are used as a check to make sure that the
1220            // signing operation is not using an out of range count value
1221            t1 = (UINT16)currentCount;
1222
1223            // If the lower bits of c are greater or equal to the lower bits of t1
1224            // then the upper bits of t1 must be one more than the upper bits
1225            // of c
1226            if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
1227                // Since the counter is behind, reduce the current count
1228                currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
1229
1230            t1 = (UINT16)currentCount;
1231            if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
1232                return FALSE;
1233            // set the counter to the value that was
1234            // present when the commitment was made
1235            currentCount = (currentCount & 0xffffffffffff0000) | *c;
1236
1237        }
1238        // Marshal the count value to a TPM2B buffer for the KDF
1239        cntr.t.size = sizeof(currentCount);
1240        UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
1241
1242        // Now can do the KDF to create the random value for the signing operation
1243        // During the creation process, we may generate an r that does not meet the
1244        // requirements of the random value.
1245        // want to generate a new r.
1246
1247        r->t.size = n->size;
1248
1249        // Arbitrary upper limit on the number of times that we can look for
1250        // a suitable random value.  The normally number of tries will be 1.
1251        for(iterations = 1; iterations < 1000000;)
1252        {
1253            BYTE    *pr = &r->b.buffer[0];
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1254            int     i;
1255            CryptKDFa(hashAlg, &gr.commitNonce.b, "ECDAA Commit",
1256                      name, &cntr.b, n->size * 8, r->t.buffer, &iterations);
1257
1258            // random value must be less than the prime
1259            if(CryptCompare(r->b.size, r->b.buffer, n->size, n->buffer) >= 0)
1260                continue;
1261
1262            // in this implementation it is required that at least bit
1263            // in the upper half of the number be set
1264            for(i = n->size/2; i > 0; i--)
1265                if(*pr++ != 0)
1266                    return TRUE;
1267        }
1268        return FALSE;
1269    }
```

11.2.7.13 CryptCommit()

This function is called when the count value is committed. The gr.c*ommitArray* 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.

```
1270    UINT16
1271    CryptCommit(
1272        void
1273        )
1274    {
1275        UINT16      oldCount = (UINT16)gr.commitCounter;
1276        gr.commitCounter++;
1277        BitSet(oldCount & COMMIT_INDEX_MASK, gr.commitArray, sizeof(gr.commitArray));
1278        return oldCount;
1279    }
```

11.2.7.14 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the gr.c*ommitArray* bit associated with the count value so that it can't be used again.

```
1280    void
1281    CryptEndCommit(
1282        UINT16                  c               // IN: the counter value of the commitment
1283        )
1284    {
1285        BitClear((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
1286    }
```

11.2.7.15 CryptCommitCompute()

This function performs the computations for the TPM2_Commit() command. This could be a macro.

Table 142

Error Returns	Meaning
TPM_RC_NO_RESULT	*K*, *L*, or *E* is the point at infinity
TPM_RC_CANCELLED	command was canceled

```
1287    TPM_RC
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
CryptCommitCompute(
    TPMS_ECC_POINT          *K,             // OUT: [d]B
    TPMS_ECC_POINT          *L,             // OUT: [r]B
    TPMS_ECC_POINT          *E,             // OUT: [r]M
    TPM_ECC_CURVE            curveID,       // IN: The curve for the computations
    TPMS_ECC_POINT          *M,             // IN: M (P1)
    TPMS_ECC_POINT          *B,             // IN: B (x2, y2)
    TPM2B_ECC_PARAMETER     *d,             // IN: the private scalar
    TPM2B_ECC_PARAMETER     *r              // IN: the computed r value
    )
{
    TEST(ALG_ECDH_VALUE);
    // CRYPT_NO_RESULT->TPM_RC_NO_RESULT CRYPT_CANCEL->TPM_RC_CANCELLED
    return TranslateCryptErrors(
                _cpri__EccCommitCompute(K, L , E, curveID, M, B, d, r));
}
```

11.2.7.16 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve.

Table 143

Return Value	Meaning
TRUE	Get parameters success
FALSE	Unsupported ECC curve ID

```
BOOL
CryptEccGetParameters(
    TPM_ECC_CURVE                curveId,       // IN: ECC curve ID
    TPMS_ALGORITHM_DETAIL_ECC   *parameters     // OUT: ECC parameters
    )
{
    const ECC_CURVE             *curve = _cpri__EccGetParametersByCurveId(curveId);
    const ECC_CURVE_DATA        *data;
    BOOL                         found = curve != NULL;

    if(found)
    {

        data = curve->curveData;

        parameters->curveID = curve->curveId;

        // Key size in bit
        parameters->keySize = curve->keySizeBits;

        // KDF
        parameters->kdf = curve->kdf;

        // Sign
        parameters->sign = curve->sign;

        // Copy p value
        MemoryCopy2B(&parameters->p.b, data->p, sizeof(parameters->p.t.buffer));

        // Copy a value
        MemoryCopy2B(&parameters->a.b, data->a, sizeof(parameters->a.t.buffer));

        // Copy b value
        MemoryCopy2B(&parameters->b.b, data->b, sizeof(parameters->b.t.buffer));

```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
1339            // Copy Gx value
1340            MemoryCopy2B(&parameters->gX.b, data->x, sizeof(parameters->gX.t.buffer));
1341
1342            // Copy Gy value
1343            MemoryCopy2B(&parameters->gY.b, data->y, sizeof(parameters->gY.t.buffer));
1344
1345            // Copy n value
1346            MemoryCopy2B(&parameters->n.b, data->n, sizeof(parameters->n.t.buffer));
1347
1348            // Copy h value
1349            MemoryCopy2B(&parameters->h.b, data->h, sizeof(parameters->h.t.buffer));
1350        }
1351        return found;
1352    }
1353    #if CC_ZGen_2Phase == YES
```

CryptEcc2PhaseKeyExchange() This is the interface to the key exchange function.

```
1354    TPM_RC
1355    CryptEcc2PhaseKeyExchange(
1356        TPMS_ECC_POINT          *outZ1,         // OUT: the computed point
1357        TPMS_ECC_POINT          *outZ2,         // OUT: optional second point
1358        TPM_ALG_ID               scheme,        // IN: the key exchange scheme
1359        TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
1360        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1361        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1362        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1363        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1364        )
1365    {
1366        return (TranslateCryptErrors(_cpri__C_2_2_KeyExchange(outZ1,
1367                                                              outZ2,
1368                                                              scheme,
1369                                                              curveId,
1370                                                              dsA,
1371                                                              deA,
1372                                                              QsB,
1373                                                              QeB)));
1374    }
1375    #endif //  CC_ZGen_2Phase
1376    #endif //TPM_ALG_ECC  //% 3
```

11.2.7.17 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.

```
1377    BOOL
1378    CryptIsSchemeAnonymous(
1379        TPM_ALG_ID       scheme         // IN: the scheme algorithm to test
1380        )
1381    {
1382    #ifdef TPM_ALG_ECDAA
1383        return (scheme == TPM_ALG_ECDAA);
1384    #else
1385        UNREFERENCED(scheme);
1386        return  0;
1387    #endif
1388    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

11.2.8 Symmetric Functions

11.2.8.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```
1389  void
1390  ParmDecryptSym(
1391      TPM_ALG_ID       symAlg,         // IN: the symmetric algorithm
1392      TPM_ALG_ID       hash,           // IN: hash algorithm for KDFa
1393      UINT16           keySizeInBits, // IN: key key size in bits
1394      TPM2B           *key,            // IN: KDF HMAC key
1395      TPM2B           *nonceCaller,    // IN: nonce caller
1396      TPM2B           *nonceTpm,       // IN: nonce TPM
1397      UINT32           dataSize,       // IN: size of parameter buffer
1398      BYTE            *data            // OUT: buffer to be decrypted
1399      )
1400  {
1401      // KDF output buffer
1402      // It contains parameters for the CFB encryption
1403      // From MSB to LSB, they are the key and iv
1404      BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
1405      // Symmetric key size in byte
1406      UINT16           keySize = (keySizeInBits + 7) / 8;
1407      TPM2B_IV         iv;
1408
1409      iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
1410      // If there is decryption to do...
1411      if(iv.t.size > 0)
1412      {
1413          // Generate key and iv
1414          // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
1415          // label values.
1416          CryptKDFa(hash, key, "CFB", nonceCaller, nonceTpm,
1417                    keySizeInBits + (iv.t.size * 8), symParmString, NULL);
1418          MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size,
1419                     sizeof(iv.t.buffer));
1420
1421          CryptSymmetricDecrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB,
1422                                symParmString, &iv, dataSize, data);
1423      }
1424      return;
1425  }
```

11.2.8.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
1426  void
1427  ParmEncryptSym(
1428      TPM_ALG_ID       symAlg,         // IN: symmetric algorithm
1429      TPM_ALG_ID       hash,           // IN: hash algorithm for KDFa
1430      UINT16           keySizeInBits, // IN: AES key size in bits
1431      TPM2B           *key,            // IN: KDF HMAC key
1432      TPM2B           *nonceCaller,    // IN: nonce caller
1433      TPM2B           *nonceTpm,       // IN: nonce TPM
1434      UINT32           dataSize,       // IN: size of parameter buffer
1435      BYTE            *data            // OUT: buffer to be encrypted
1436      )
1437  {
1438      // KDF output buffer
1439      // It contains parameters for the CFB encryption
1440      BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
1441
1442        // Symmetric key size in bytes
1443        UINT16          keySize = (keySizeInBits + 7) / 8;
1444
1445        TPM2B_IV        iv;
1446
1447        iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
1448        // See if there is any encryption to do
1449        if(iv.t.size > 0)
1450        {
1451            // Generate key and iv
1452            // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
1453            // label values.
1454            CryptKDFa(hash, key, "CFB", nonceTpm, nonceCaller,
1455                      keySizeInBits + (iv.t.size * 8), symParmString, NULL);
1456
1457            MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size,
1458                       sizeof(iv.t.buffer));
1459
1460            CryptSymmetricEncrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB,
1461                                  symParmString, &iv, dataSize, data);
1462        }
1463        return;
1464    }
```

11.2.8.3 CryptGenerateNewSymmetric()

This function creates the sensitive symmetric values for an HMAC or symmetric key. If the sensitive area is zero, then the sensitive creation key data is copied. If it is not zero, then the TPM will generate a random value of the selected size.

```
1465    void
1466    CryptGenerateNewSymmetric(
1467        TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
1468        TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
1469        TPM_ALG_ID               hashAlg,           // IN: hash algorithm for the KDF
1470        TPM2B_SEED              *seed,              // IN: seed used in creation
1471        TPM2B_NAME              *name               // IN: name of the object
1472        )
1473    {
1474        // This function is called to create a key and obfuscation value for a
1475        // symmetric key that can either be a block cipher or an XOR key. The buffer
1476        // in sensitive->sensitive will hold either. When we call the function
1477        // to copy the input value or generated value to the sensitive->sensitive
1478        // buffer we will need to have a size for the output buffer. This define
1479        // computes the maximum that it might need to be and uses that. It will always
1480        // be smaller than the largest value that will fit.
1481        #define MAX_SENSITIVE_SIZE                                                  \
1482            (MAX(sizeof(sensitive->sensitive.bits.t.buffer),                        \
1483                 sizeof(sensitive->sensitive.sym.t.buffer)))
1484
1485        // set the size of the obfuscation value
1486        sensitive->seedValue.t.size = CryptGetHashDigestSize(hashAlg);
1487
1488        // If the input sensitive size is zero, then create both the sensitive data
1489        // and the obfuscation value
1490        if(sensitiveCreate->data.t.size == 0)
1491        {
1492            BYTE                    symValues[MAX_SYM_KEY_BYTES +  MAX_DIGEST_SIZE];
1493            INT16                   requestSize;
1494
1495            // Set the size of the request to be the size of the key and the
1496            // obfuscation value
1497            requestSize =   sensitive->sensitive.sym.t.size
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
1498                        + sensitive->seedValue.t.size;
1499
1500            _cpri__GenerateSeededRandom(requestSize, symValues, hashAlg, &seed->b,
1501                                        "symmetric sensitive", &name->b, NULL);
1502
1503            // Copy the new key
1504            MemoryCopy(sensitive->sensitive.sym.t.buffer,
1505                       symValues, sensitive->sensitive.sym.t.size,
1506                       MAX_SENSITIVE_SIZE);
1507
1508            // copy the obfuscation value
1509            MemoryCopy(sensitive->seedValue.t.buffer,
1510                       &symValues[sensitive->sensitive.sym.t.size],
1511                       sensitive->seedValue.t.size,
1512                       sizeof(sensitive->seedValue.t.buffer));
1513        }
1514        else
1515        {
1516            // Copy input symmetric key to sensitive area as long as it will fit
1517            MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
1518                         MAX_SENSITIVE_SIZE);
1519
1520            // Create the obfuscation value
1521            _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
1522                                        sensitive->seedValue.t.buffer,
1523                                        hashAlg, &seed->b,
1524                                        "symmetric obfuscation", &name->b, NULL);
1525        }
1526        return;
1527    }
```

11.2.8.4 CryptGenerateKeySymmetric()

This function derives a symmetric cipher key from the provided seed.

Error Returns	Meaning
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area

```
1528    static TPM_RC
1529    CryptGenerateKeySymmetric(
1530        TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
1531                                                    //     for the new key.
1532        TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
1533        TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
1534        TPM_ALG_ID               hashAlg,           // IN: hash algorithm for the KDF
1535        TPM2B_SEED              *seed,              // IN: seed used in creation
1536        TPM2B_NAME              *name               // IN: name of the object
1537        )
1538    {
1539        // If this is not a new key, then the provided key data must be the right size
1540        if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR)
1541        {
1542            if(     (sensitiveCreate->data.t.size * 8)
1543                !=  publicArea->parameters.symDetail.sym.keyBits.sym)
1544                return TPM_RC_KEY_SIZE;
1545            // Make sure that the key size is OK.
1546            // This implementation only supports symmetric key sizes that are
1547            // multiples of 8
1548            if(publicArea->parameters.symDetail.sym.keyBits.sym % 8 != 0)
1549                return TPM_RC_KEY_SIZE;
1550        }
1551        else
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
1552        {
1553            // TPM is going to generate the key so set the size
1554            sensitive->sensitive.sym.t.size
1555                = publicArea->parameters.symDetail.sym.keyBits.sym / 8;
1556            sensitiveCreate->data.t.size = 0;
1557        }
1558        // Fill in the sensitive area
1559        CryptGenerateNewSymmetric(sensitiveCreate, sensitive, hashAlg,
1560                                  seed, name);
1561
1562        // Create unique area in public
1563        CryptComputeSymmetricUnique(publicArea->nameAlg,
1564                                    sensitive, &publicArea->unique.sym);
1565
1566        return TPM_RC_SUCCESS;
1567    }
```

11.2.8.5 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.

```
1568    #ifdef TPM_ALG_KEYEDHASH //% 5
1569    void
1570    CryptXORObfuscation(
1571        TPM_ALG_ID       hash,          // IN: hash algorithm for KDF
1572        TPM2B           *key,           // IN: KDF key
1573        TPM2B           *contextU,      // IN: contextU
1574        TPM2B           *contextV,      // IN: contextV
1575        UINT32           dataSize,      // IN: size of data buffer
1576        BYTE            *data           // IN/OUT: data to be XORed in place
1577        )
1578    {
1579        BYTE             mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
1580        BYTE            *pm;
1581        UINT32           i;
1582        UINT32           counter = 0;
1583        UINT16           hLen = CryptGetHashDigestSize(hash);
1584        UINT32           requestSize = dataSize * 8;
1585        INT32            remainBytes = (INT32) dataSize;
1586
1587        pAssert((key != NULL) && (data != NULL) && (hLen != 0));
1588
1589        // Call KDFa to generate XOR mask
1590        for(; remainBytes > 0; remainBytes -= hLen)
1591        {
1592            // Make a call to KDFa to get next iteration
1593            // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
1594            // label values.
1595            CryptKDFaOnce(hash, key, "XOR", contextU, contextV,
1596                          requestSize, mask, &counter);
1597
1598            // XOR next piece of the data
1599            pm = mask;
1600            for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
1601                *data++ ^= *pm++;
1602        }
1603        return;
1604    }
1605    #endif //TPM_ALG_KEYED_HASH //%5
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

11.2.9 Initialization and shut down

11.2.9.1 CryptInitUnits()

This function is called when the TPM receives a _TPM_Init() indication. After function returns, the hash algorithms should be available.

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.

```
1606  void
1607  CryptInitUnits(
1608      void
1609      )
1610  {
1611      // Initialize the vector of implemented algorithms
1612      AlgorithmGetImplementedVector(&g_implementedAlgorithms);
1613
1614      // Indicate that all test are necessary
1615      CryptInitializeToTest();
1616
1617      // Call crypto engine unit initialization
1618      // It is assumed that crypt engine initialization should always succeed.
1619      // Otherwise, TPM should go to failure mode.
1620      if(_cpri__InitCryptoUnits(&TpmFail) != CRYPT_SUCCESS)
1621          FAIL(FATAL_ERROR_INTERNAL);
1622      return;
1623  }
```

11.2.9.2 CryptStopUnits()

This function is only used in a simulated environment. There should be no reason to shut down the cryptography on an actual TPM other than loss of power. After receiving TPM2_Startup(), the TPM should be able to accept commands until it loses power and, unless the TPM is in Failure Mode, the cryptographic algorithms should be available.

```
1624  void
1625  CryptStopUnits(
1626      void
1627      )
1628  {
1629      // Call crypto engine unit stopping
1630      _cpri__StopCryptoUnits();
1631
1632      return;
1633  }
```

11.2.9.3 CryptUtilStartup()

This function is called by TPM2_Startup() to initialize the functions in this crypto library and in the provided CryptoEngine(). In this implementation, the only initialization required in this library is initialization of the Commit nonce on TPM Reset.

This function returns false if some problem prevents the functions from starting correctly. The TPM should go into failure mode.

```
1634  BOOL
1635  CryptUtilStartup(
1636      STARTUP_TYPE     type           // IN: the startup type
1637      )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
{
    // Make sure that the crypto library functions are ready.
    // NOTE: need to initialize the crypto before loading
    // the RND state may trigger a self-test which
    // uses the
    if( !_cpri__Startup())
        return FALSE;

    // Initialize the state of the RNG.
    CryptDrbgGetPutState(PUT_STATE);

    if(type == SU_RESET)
    {
#ifdef TPM_ALG_ECC
        // Get a new  random commit nonce
        gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
        _cpri__GenerateRandom(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
        // Reset the counter and commit array
        gr.commitCounter = 0;
        MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
#endif // TPM_ALG_ECC
    }

    // If the shutdown was orderly, then the values recovered from NV will
    // be OK to use. If the shutdown was not orderly, then a TPM Reset was required
    // and we would have initialized in the code above.

    return TRUE;
}
```

11.2.10 Algorithm-Independent Functions

11.2.10.1 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.

11.2.10.2 CryptIsAsymAlgorithm()

This function indicates if an algorithm is an asymmetric algorithm.

Table 144

Return Value	Meaning
TRUE	if it is an asymmetric algorithm
FALSE	if it is not an asymmetric algorithm

```
BOOL
CryptIsAsymAlgorithm(
    TPM_ALG_ID       algID          // IN: algorithm ID
    )
{
    return (
#ifdef TPM_ALG_RSA
              algID ==  TPM_ALG_RSA
#endif
#if defined TPM_ALG_RSA && defined TPM_ALG_ECC
              ||
#endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
#ifdef TPM_ALG_ECC
             algID == TPM_ALG_ECC
#endif
          );
}
```

11.2.10.3 CryptGetSymmetricBlockSize()

This function returns the size in octets of the symmetric encryption block used by an algorithm and key size combination.

```
INT16
CryptGetSymmetricBlockSize(
    TPMI_ALG_SYM     algorithm,     // IN: symmetric algorithm
    UINT16           keySize        // IN: key size in bit
    )
{
    return _cpri__GetSymmetricBlockSize(algorithm, keySize);
}
```

11.2.10.4 CryptSymmetricEncrypt()

This function does in-place encryption of a buffer using the indicated symmetric algorithm, key, IV, and mode. If the symmetric algorithm and mode are not defined, the TPM will fail.

```
void
CryptSymmetricEncrypt(
    BYTE                *encrypted,     // OUT: the encrypted data
    TPM_ALG_ID           algorithm,     // IN: algorithm for encryption
    UINT16               keySizeInBits, // IN: key size in bits
    TPMI_ALG_SYM_MODE    mode,          // IN: symmetric encryption mode
    BYTE                *key,           // IN: encryption key
    TPM2B_IV            *ivIn,          // IN/OUT: Input IV and output chaining
                                        //     value for the next block
    UINT32               dataSize,      // IN: data size in byte
    BYTE                *data           // IN/OUT: data buffer
    )
{
    TPM2B_IV             defaultIv = {0};

    TEST(algorithm);

    pAssert(encrypted != NULL && key != NULL);

    defaultIv.t.size = _cpri__GetSymmetricBlockSize(algorithm, keySizeInBits);

    // If the IV is not provided, or if doing ECB, then use the default IV
    if(ivIn == NULL || mode == TPM_ALG_ECB)
       ivIn = &defaultIv;

    // Make sure that there is an IV and that the provided size is the
    // requried size.
    pAssert(ivIn->t.size == defaultIv.t.size);

   if( _cpri__SymmetricEncrypt(
        encrypted,
        algorithm,
        keySizeInBits,
        key,
        ivIn,
        mode,
        dataSize,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1729            data) != CRYPT_SUCCESS)
1730            FAIL(FATAL_ERROR_CRYPTO);
1731
1732        return;
1733
1734    }
```

11.2.10.5 CryptSymmetricDecrypt()

This function does in-place decryption of a buffer using the indicated symmetric algorithm, key, IV, and mode. If the symmetric algorithm and mode are not defined, the TPM will fail.

```
1735    void
1736    CryptSymmetricDecrypt(
1737        BYTE                *decrypted,     // OUT: the decrypted data
1738        TPM_ALG_ID           algorithm,     // IN: algorithm for encryption
1739        UINT16               keySizeInBits, // IN: key size in bits
1740        TPMI_ALG_SYM_MODE    mode,          // IN: symmetric encryption mode
1741        BYTE                *key,           // IN: encryption key
1742        TPM2B_IV            *ivIn,          // IN/OUT: Input IV and output chaining
1743                                            //     value for the next block
1744        UINT32               dataSize,      // IN: data size in byte
1745        BYTE                *data           // IN/OUT: data buffer
1746        )
1747    {
1748        TPM2B_IV             defaultIv = {0};
1749
1750        TEST(algorithm);
1751
1752        pAssert(decrypted != NULL && key != NULL);
1753
1754        defaultIv.t.size = _cpri__GetSymmetricBlockSize(algorithm, keySizeInBits);
1755
1756        // If the IV is not provided, or if doing ECB, then use the default IV
1757        if(ivIn == NULL || mode == TPM_ALG_ECB)
1758            ivIn = &defaultIv;
1759
1760        // Make sure that there is an IV and that the provided size is the
1761        // requried size.
1762        pAssert(ivIn->t.size == defaultIv.t.size);
1763
1764        if(_cpri__SymmetricDecrypt(
1765            decrypted,
1766            algorithm,
1767            keySizeInBits,
1768            key,
1769            ivIn,
1770            mode,
1771            dataSize,
1772            data) != CRYPT_SUCCESS)
1773            FAIL(FATAL_ERROR_CRYPTO);
1774
1775        return;
1776
1777    }
```

11.2.10.6 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().

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 145

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a valid decryption key
TPM_RC_KEY	invalid ECC key (public point is not on the curve)
TPM_RC_SCHEME	RSA key with an unsupported padding scheme
TPM_RC_VALUE	numeric value of the data to be decrypted is greater than the RSA key modulus

```
1778  TPM_RC
1779  CryptSecretEncrypt(
1780      TPMI_DH_OBJECT           keyHandle,     // IN: encryption key handle
1781      const char              *label,         // IN: a null-terminated string as L
1782      TPM2B_DATA              *data,          // OUT: secret value
1783      TPM2B_ENCRYPTED_SECRET  *secret         // OUT: secret structure
1784      )
1785  {
1786      TPM_RC       result = TPM_RC_SUCCESS;
1787      OBJECT      *encryptKey = ObjectGet(keyHandle);  // TPM key used for encrypt
1788
1789      pAssert(data != NULL && secret != NULL);
1790
1791      // The output secret value has the size of the digest produced by the nameAlg.
1792      data->t.size = CryptGetHashDigestSize(encryptKey->publicArea.nameAlg);
1793
1794      pAssert(encryptKey->publicArea.objectAttributes.decrypt == SET);
1795
1796      switch(encryptKey->publicArea.type)
1797      {
1798  #ifdef TPM_ALG_RSA
1799          case TPM_ALG_RSA:
1800          {
1801              TPMT_RSA_DECRYPT            scheme;
1802
1803              // Use OAEP scheme
1804              scheme.scheme = TPM_ALG_OAEP;
1805              scheme.details.oaep.hashAlg = encryptKey->publicArea.nameAlg;
1806
1807              // Create secret data from RNG
1808              CryptGenerateRandom(data->t.size, data->t.buffer);
1809
1810              // Encrypt the data by RSA OAEP into encrypted secret
1811              result = CryptEncryptRSA(&secret->t.size, secret->t.secret,
1812                                       encryptKey, &scheme,
1813                                       data->t.size, data->t.buffer, label);
1814          }
1815          break;
1816  #endif //TPM_ALG_RSA
1817
1818  #ifdef TPM_ALG_ECC
1819          case TPM_ALG_ECC:
1820          {
1821              TPMS_ECC_POINT      eccPublic;
1822              TPM2B_ECC_PARAMETER eccPrivate;
1823              TPMS_ECC_POINT      eccSecret;
1824              BYTE               *buffer = secret->t.secret;
1825
1826              // Need to make sure that the public point of the key is on the
1827              // curve defined by the key.
1828              if(!_cpri__EccIsPointOnCurve(
1829                          encryptKey->publicArea.parameters.eccDetail.curveID,
1830                          &encryptKey->publicArea.unique.ecc))
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1831                result = TPM_RC_KEY;
1832            else
1833            {
1834
1835                // Call crypto engine to create an auxiliary ECC key
1836                // We assume crypt engine initialization should always success.
1837                // Otherwise, TPM should go to failure mode.
1838                CryptNewEccKey(encryptKey->publicArea.parameters.eccDetail.curveID,
1839                               &eccPublic, &eccPrivate);
1840
1841                // Marshal ECC public to secret structure. This will be used by the
1842                // recipient to decrypt the secret with their private key.
1843                secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
1844
1845                // Compute ECDH shared secret which is R = [d]Q where d is the
1846                // private part of the ephemeral key and Q is the public part of a
1847                // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
1848                // because the auxiliary ECC key is just created according to the
1849                // parameters of input ECC encrypt key.
1850                if(    CryptEccPointMultiply(&eccSecret,
1851                               encryptKey->publicArea.parameters.eccDetail.curveID,
1852                               &eccPrivate,
1853                               &encryptKey->publicArea.unique.ecc)
1854                   != CRYPT_SUCCESS)
1855                    result = TPM_RC_KEY;
1856                else
1857
1858                    // The secret value is computed from Z using KDFe as:
1859                    // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
1860                    // Where:
1861                    //  HashID  the nameAlg of the decrypt key
1862                    //  Z   the x coordinate (Px) of the product (P) of the point
1863                    //      (Q) of the secret and the private x coordinate (de,V)
1864                    //      of the decryption key
1865                    //  Use a null-terminated string containing "SECRET"
1866                    //    See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for
1867                    //    normative KDF label values.
1868                    //  PartyUInfo  the x coordinate of the point in the secret
1869                    //              (Qe,U )
1870                    //  PartyVInfo  the x coordinate of the public key (Qs,V )
1871                    //  bits  the number of bits in the digest of HashID
1872                    // Retrieve seed from KDFe
1873
1874                    CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
1875                              label, &eccPublic.x.b,
1876                              &encryptKey->publicArea.unique.ecc.x.b,
1877                              data->t.size * 8, data->t.buffer);
1878            }
1879        }
1880        break;
1881 #endif //TPM_ALG_ECC
1882
1883     default:
1884        FAIL(FATAL_ERROR_INTERNAL);
1885        break;
1886     }
1887
1888     return result;
1889 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

11.2.10.7 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.

Table 146

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

```
1890    TPM_RC
1891    CryptSecretDecrypt(
1892        TPM_HANDLE                tpmKey,        // IN: decrypt key
1893        TPM2B_NONCE              *nonceCaller,   // IN: nonceCaller.  It is needed for
1894                                                 //     symmetric decryption.  For
1895                                                 //     asymmetric decryption, this
1896                                                 //     parameter is NULL
1897        const char               *label,         // IN: a null-terminated string as L
1898        TPM2B_ENCRYPTED_SECRET   *secret,        // IN: input secret
1899        TPM2B_DATA               *data           // OUT: decrypted secret value
1900        )
1901    {
1902        TPM_RC      result = TPM_RC_SUCCESS;
1903        OBJECT      *decryptKey = ObjectGet(tpmKey);   //TPM key used for decrypting
1904
1905        // Decryption for secret
1906        switch(decryptKey->publicArea.type)
1907        {
1908
1909    #ifdef TPM_ALG_RSA
1910            case TPM_ALG_RSA:
1911            {
1912                TPMT_RSA_DECRYPT        scheme;
1913
1914                // Use OAEP scheme
1915                scheme.scheme = TPM_ALG_OAEP;
1916                scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
1917
1918                // Set the output buffer capacity
1919                data->t.size = sizeof(data->t.buffer);
1920
1921                // Decrypt seed by RSA OAEP
1922                result = CryptDecryptRSA(&data->t.size, data->t.buffer, decryptKey,
1923                                         &scheme,
1924                                         secret->t.size, secret->t.secret,label);
1925                if(   (result == TPM_RC_SUCCESS)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1926                && (data->t.size
1927                     > CryptGetHashDigestSize(decryptKey->publicArea.nameAlg)))
1928                 result = TPM_RC_VALUE;
1929         }
1930         break;
1931 #endif //TPM_ALG_RSA
1932
1933 #ifdef TPM_ALG_ECC
1934         case TPM_ALG_ECC:
1935         {
1936             TPMS_ECC_POINT       eccPublic;
1937             TPMS_ECC_POINT       eccSecret;
1938             BYTE                *buffer = secret->t.secret;
1939             INT32                size = secret->t.size;
1940
1941             // Retrieve ECC point from secret buffer
1942             result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
1943             if(result == TPM_RC_SUCCESS)
1944             {
1945                 result = CryptEccPointMultiply(&eccSecret,
1946                                 decryptKey->publicArea.parameters.eccDetail.curveID,
1947                                 &decryptKey->sensitive.sensitive.ecc,
1948                                 &eccPublic);
1949
1950                 if(result == TPM_RC_SUCCESS)
1951                 {
1952
1953                     // Set the size of the "recovered" secret value to be the size
1954                     // of the digest produced by the nameAlg.
1955                     data->t.size =
1956                             CryptGetHashDigestSize(decryptKey->publicArea.nameAlg);
1957
1958                     // The secret value is computed from Z using KDFe as:
1959                     // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
1960                     // Where:
1961                     //  HashID -- the nameAlg of the decrypt key
1962                     //  Z --  the x coordinate (Px) of the product (P) of the point
1963                     //        (Q) of the secret and the private x coordinate (de,V)
1964                     //        of the decryption key
1965                     //  Use -- a null-terminated string containing "SECRET"
1966                     //    See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for
1967                     //    normative KDF label values.
1968                     //  PartyUInfo -- the x coordinate of the point in the secret
1969                     //                (Qe,U )
1970                     //  PartyVInfo -- the x coordinate of the public key (Qs,V )
1971                     //  bits -- the number of bits in the digest of HashID
1972                     // Retrieve seed from KDFe
1973                     CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
1974                               &eccPublic.x.b,
1975                               &decryptKey->publicArea.unique.ecc.x.b,
1976                               data->t.size * 8, data->t.buffer);
1977                 }
1978             }
1979         }
1980         break;
1981 #endif //TPM_ALG_ECC
1982
1983         case TPM_ALG_KEYEDHASH:
1984             // The seed size can not be bigger than the digest size of nameAlg
1985             if(secret->t.size >
1986                     CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
1987                 result = TPM_RC_VALUE;
1988             else
1989             {
1990                 // Retrieve seed by XOR Obfuscation:
1991                 //    seed = XOR(secret, hash, key, nonceCaller, nullNonce)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1992                //    where:
1993                //    secret  the secret parameter from the TPM2_StartAuthHMAC
1994                //            command
1995                //            which contains the seed value
1996                //    hash    nameAlg  of tpmKey
1997                //    key     the key or data value in the object referenced by
1998                //            entityHandle in the TPM2_StartAuthHMAC command
1999                //    nonceCaller the parameter from the TPM2_StartAuthHMAC command
2000                //    nullNonce   a zero-length nonce
2001                // XOR Obfuscation in place
2002                CryptXORObfuscation(decryptKey->publicArea.nameAlg,
2003                                    &decryptKey->sensitive.sensitive.bits.b,
2004                                    &nonceCaller->b, NULL,
2005                                    secret->t.size, secret->t.secret);
2006                // Copy decrypted seed
2007                MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
2008            }
2009            break;
2010        case TPM_ALG_SYMCIPHER:
2011            {
2012                TPM2B_IV                iv = {0};
2013                TPMT_SYM_DEF_OBJECT     *symDef;
2014                // The seed size can not be bigger than the digest size of nameAlg
2015                if(secret->t.size >
2016                        CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
2017                    result = TPM_RC_VALUE;
2018                else
2019                {
2020                    symDef = &decryptKey->publicArea.parameters.symDetail.sym;
2021                    iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
2022                                                           symDef->keyBits.sym);
2023                    pAssert(iv.t.size != 0);
2024                    if(nonceCaller->t.size >= iv.t.size)
2025                        MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size,
2026                                   sizeof(iv.t.buffer));
2027                    else
2028                        MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
2029                                   nonceCaller->t.size, sizeof(iv.t.buffer));
2030                    // CFB decrypt in place, using nonceCaller as iv
2031                    CryptSymmetricDecrypt(secret->t.secret, symDef->algorithm,
2032                                          symDef->keyBits.sym, TPM_ALG_CFB,
2033                                          decryptKey->sensitive.sensitive.sym.t.buffer,
2034                                          &iv, secret->t.size, secret->t.secret);
2035
2036                    // Copy decrypted seed
2037                    MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
2038                }
2039            }
2040            break;
2041        default:
2042            pAssert(0);
2043            break;
2044    }
2045    return result;
2046 }
```

11.2.10.8 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```
2047 void
2048 CryptParameterEncryption(
2049     TPM_HANDLE       handle,            // IN: encrypt session handle
2050     TPM2B           *nonceCaller,       // IN: nonce caller
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
2051        UINT16           leadingSizeInByte, // IN: the size of the leading size field in
2052                                            //     bytes
2053        TPM2B_AUTH      *extraKey,          // IN: additional key material other than
2054                                            //     session auth
2055        BYTE            *buffer             // IN/OUT: parameter buffer to be encrypted
2056        )
2057    {
2058        SESSION     *session = SessionGet(handle);  // encrypt session
2059        TPM2B_TYPE(SYM_KEY, (  sizeof(extraKey->t.buffer)
2060                             + sizeof(session->sessionKey.t.buffer)));
2061        TPM2B_SYM_KEY        key;               // encryption key
2062        UINT32               cipherSize = 0;    // size of cipher text
2063
2064        pAssert(session->sessionKey.t.size + extraKey->t.size <= sizeof(key.t.buffer));
2065
2066        // Retrieve encrypted data size.
2067        if(leadingSizeInByte == 2)
2068        {
2069            // Extract the first two bytes as the size field as the data size
2070            // encrypt
2071            cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
2072            // advance the buffer
2073            buffer = &buffer[2];
2074        }
2075    #ifdef      TPM4B
2076        else if(leadingSizeInByte == 4)
2077        {
2078            // use the first four bytes to indicate the number of bytes to encrypt
2079            cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
2080            //advance pointer
2081            buffer = &buffer[4];
2082        }
2083    #endif
2084        else
2085        {
2086            pAssert(FALSE);
2087        }
2088
2089        // Compute encryption key by concatenating sessionAuth with extra key
2090        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
2091        MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
2092
2093        if (session->symmetric.algorithm == TPM_ALG_XOR)
2094
2095            // XOR parameter encryption formulation:
2096            //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
2097            CryptXORObfuscation(session->authHashAlg, &(key.b),
2098                                &(session->nonceTPM.b),
2099                                nonceCaller, cipherSize, buffer);
2100        else
2101            ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
2102                           session->symmetric.keyBits.aes, &(key.b),
2103                           nonceCaller, &(session->nonceTPM.b),
2104                           cipherSize, buffer);
2105        return;
2106    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

11.2.10.9 CryptParameterDecryption()

This function does in-place decryption of a command parameter.

© ISO/IEC 2015 – All rights reserved

Table 147

Error Returns	Meaning
TPM_RC_SIZE	The number of bytes in the input buffer is less than the number of bytes to be decrypted.

```
2107    TPM_RC
2108    CryptParameterDecryption(
2109        TPM_HANDLE       handle,            // IN: encrypted session handle
2110        TPM2B           *nonceCaller,       // IN: nonce caller
2111        UINT32           bufferSize,        // IN: size of parameter buffer
2112        UINT16           leadingSizeInByte, // IN: the size of the leading size field in
2113                                            //     byte
2114        TPM2B_AUTH      *extraKey,          // IN: the authValue
2115        BYTE            *buffer             // IN/OUT: parameter buffer to be decrypted
2116        )
2117    {
2118        SESSION         *session = SessionGet(handle);  // encrypt session
2119        // The HMAC key is going to be the concatenation of the session key and any
2120        // additional key material (like the authValue). The size of both of these
2121        // is the size of the buffer which can contain a TPMT_HA.
2122        TPM2B_TYPE(HMAC_KEY, (  sizeof(extraKey->t.buffer)
2123                              + sizeof(session->sessionKey.t.buffer)));
2124        TPM2B_HMAC_KEY          key;            // decryption key
2125        UINT32                  cipherSize = 0; // size of cipher text
2126
2127        pAssert(session->sessionKey.t.size + extraKey->t.size <= sizeof(key.t.buffer));
2128
2129        // Retrieve encrypted data size.
2130        if(leadingSizeInByte == 2)
2131        {
2132            // The first two bytes of the buffer are the size of the
2133            // data to be decrypted
2134            cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
2135            buffer = &buffer[2];   // advance the buffer
2136        }
2137    #ifdef  TPM4B
2138        else if(leadingSizeInByte == 4)
2139        {
2140            // the leading size is four bytes so get the four byte size field
2141            cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
2142            buffer = &buffer[4];   //advance pointer
2143        }
2144    #endif
2145        else
2146        {
2147            pAssert(FALSE);
2148        }
2149        if(cipherSize > bufferSize)
2150            return TPM_RC_SIZE;
2151
2152        // Compute decryption key by concatenating sessionAuth with extra input key
2153        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
2154        MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
2155
2156        if(session->symmetric.algorithm == TPM_ALG_XOR)
2157            // XOR parameter decryption formulation:
2158            //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
2159            // Call XOR obfuscation function
2160            CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
2161                                &(session->nonceTPM.b), cipherSize, buffer);
2162        else
2163            // Assume that it is one of the symmetric block ciphers.
2164            ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
2165                           session->symmetric.keyBits.sym,
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2166                                &key.b, nonceCaller, &session->nonceTPM.b,
2167                                cipherSize, buffer);
2168
2169        return TPM_RC_SUCCESS;
2170
2171    }
```

11.2.10.10 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```
2172    void
2173    CryptComputeSymmetricUnique(
2174        TPMI_ALG_HASH    nameAlg,       // IN: object name algorithm
2175        TPMT_SENSITIVE  *sensitive,     // IN: sensitive area
2176        TPM2B_DIGEST    *unique         // OUT: unique buffer
2177        )
2178    {
2179        HASH_STATE  hashState;
2180
2181        pAssert(sensitive != NULL && unique != NULL);
2182
2183        // Compute the public value as the hash of sensitive.symkey || unique.buffer
2184        unique->t.size = CryptGetHashDigestSize(nameAlg);
2185        CryptStartHash(nameAlg, &hashState);
2186
2187        // Add obfuscation value
2188        CryptUpdateDigest2B(&hashState, &sensitive->seedValue.b);
2189
2190        // Add sensitive value
2191        CryptUpdateDigest2B(&hashState, &sensitive->sensitive.any.b);
2192
2193        CryptCompleteHash2B(&hashState, &unique->b);
2194
2195        return;
2196    }
2197    #if 0 //%
```

11.2.10.11 CryptComputeSymValue()

This function computes the *seedValue* field in asymmetric sensitive areas.

```
2198    void
2199    CryptComputeSymValue(
2200        TPM_HANDLE       parentHandle,  // IN: parent handle of the object to be created
2201        TPMT_PUBLIC     *publicArea,    // IN/OUT: the public area template
2202        TPMT_SENSITIVE  *sensitive,     // IN: sensitive area
2203        TPM2B_SEED      *seed,          // IN: the seed
2204        TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm for KDFa
2205        TPM2B_NAME      *name           // IN: object name
2206        )
2207    {
2208        TPM2B_AUTH   *proof = NULL;
2209
2210        if(CryptIsAsymAlgorithm(publicArea->type))
2211        {
2212            // Generate seedValue only when an asymmetric key is a storage key
2213            if(publicArea->objectAttributes.decrypt == SET
2214                    && publicArea->objectAttributes.restricted == SET)
2215            {
2216                // If this is a primary object in the endorsement hierarchy, use
2217                // ehProof in the creation of the symmetric seed so that child
2218                // objects in the endorsement hierarchy are voided on TPM2_Clear()
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
            // or TPM2_ChangeEPS()
            if(   parentHandle == TPM_RH_ENDORSEMENT
               && publicArea->objectAttributes.fixedTPM == SET)
                proof = &gp.ehProof;
        }
        else
        {
            sensitive->seedValue.t.size = 0;
            return;
        }
    }

    // For all object types, the size of seedValue is the digest size of nameAlg
    sensitive->seedValue.t.size = CryptGetHashDigestSize(publicArea->nameAlg);

    // Compute seedValue using implementation-dependent method
    _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
                                sensitive->seedValue.t.buffer,
                                hashAlg,
                                &seed->b,
                                "seedValue",
                                &name->b,
                                (TPM2B *)proof);
    return;
}
#endif //%
```

11.2.10.12 CryptCreateObject()

This function creates an object. It:

a) fills in the created key in public and sensitive area;

b) creates a random number in sensitive area for symmetric keys; and

c) compute the unique id in public area for symmetric keys.

Table 148

Error Returns	Meaning
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area for a symmetric key
TPM_RC_RANGE	for an RSA key, the exponent is not supported
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme for a keyed hash object
TPM_RC_VALUE	exponent is not prime or could not find a prime using the provided parameters for an RSA key; unsupported name algorithm for an ECC key

```
TPM_RC
CryptCreateObject(
    TPM_HANDLE                parentHandle,      // IN/OUT: indication of the seed
                                                 //     source
    TPMT_PUBLIC              *publicArea,        // IN/OUT: public area
    TPMS_SENSITIVE_CREATE    *sensitiveCreate,   // IN: sensitive creation
    TPMT_SENSITIVE           *sensitive          // OUT: sensitive area
    )
{
    // Next value is a placeholder for a random seed that is used in
    // key creation when the parent is not a primary seed. It has the same
    // size as the primary seed.
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
2257
2258        TPM2B_SEED       localSeed;     // data to seed key creation if this
2259                                        // is not a primary seed
2260
2261        TPM2B_SEED      *seed = NULL;
2262        TPM_RC           result = TPM_RC_SUCCESS;
2263
2264        TPM2B_NAME       name;
2265        TPM_ALG_ID       hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
2266        OBJECT          *parent;
2267        UINT32           counter;
2268
2269        // Set the sensitive type for the object
2270        sensitive->sensitiveType = publicArea->type;
2271        ObjectComputeName(publicArea, &name);
2272
2273        // For all objects, copy the initial auth data
2274        sensitive->authValue = sensitiveCreate->userAuth;
2275
2276        // If this is a permanent handle assume that it is a hierarchy
2277        if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
2278        {
2279            seed = HierarchyGetPrimarySeed(parentHandle);
2280        }
2281        else
2282        {
2283            // If not hierarchy handle, get parent
2284            parent = ObjectGet(parentHandle);
2285            hashAlg = parent->publicArea.nameAlg;
2286
2287            // Use random value as seed for non-primary objects
2288            localSeed.t.size = PRIMARY_SEED_SIZE;
2289            CryptGenerateRandom(PRIMARY_SEED_SIZE, localSeed.t.buffer);
2290            seed = &localSeed;
2291        }
2292
2293        switch(publicArea->type)
2294        {
2295    #ifdef TPM_ALG_RSA
2296            // Create RSA key
2297        case TPM_ALG_RSA:
2298            result = CryptGenerateKeyRSA(publicArea, sensitive,
2299                                         hashAlg, seed, &name, &counter);
2300            break;
2301    #endif // TPM_ALG_RSA
2302
2303    #ifdef TPM_ALG_ECC
2304            // Create ECC key
2305        case TPM_ALG_ECC:
2306            result = CryptGenerateKeyECC(publicArea, sensitive,
2307                                         hashAlg, seed, &name, &counter);
2308            break;
2309    #endif // TPM_ALG_ECC
2310
2311            // Collect symmetric key information
2312        case TPM_ALG_SYMCIPHER:
2313            return CryptGenerateKeySymmetric(publicArea, sensitiveCreate,
2314                                             sensitive, hashAlg, seed, &name);
2315            break;
2316        case TPM_ALG_KEYEDHASH:
2317            return CryptGenerateKeyedHash(publicArea, sensitiveCreate,
2318                                          sensitive, hashAlg, seed, &name);
2319            break;
2320        default:
2321            pAssert(0);
2322            break;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2323        }
2324        if(result == TPM_RC_SUCCESS)
2325        {
2326            TPM2B_AUTH          *proof = NULL;
2327
2328            if(publicArea->objectAttributes.decrypt == SET
2329                    && publicArea->objectAttributes.restricted == SET)
2330            {
2331                // If this is a primary object in the endorsement hierarchy, use
2332                // ehProof in the creation of the symmetric seed so that child
2333                // objects in the endorsement hierarchy are voided on TPM2_Clear()
2334                // or TPM2_ChangeEPS()
2335                if(   parentHandle == TPM_RH_ENDORSEMENT
2336                   && publicArea->objectAttributes.fixedTPM == SET)
2337                    proof = &gp.ehProof;
2338
2339                // For all object types, the size of seedValue is the digest size
2340                // of its nameAlg
2341                sensitive->seedValue.t.size
2342                    = CryptGetHashDigestSize(publicArea->nameAlg);
2343
2344                // Compute seedValue using implementation-dependent method
2345                _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
2346                                            sensitive->seedValue.t.buffer,
2347                                            hashAlg,
2348                                            &seed->b,
2349                                            "seedValuea",
2350                                            &name.b,
2351                                            (TPM2B *)proof);
2352            }
2353            else
2354            {
2355                sensitive->seedValue.t.size = 0;
2356            }
2357        }
2358
2359        return result;
2360
2361    }
```

11.2.10.13 CryptObjectIsPublicConsistent()

This function checks that the key sizes in the public area are consistent. For an asymmetric key, the size of the public key must match the size indicated by the public->parameters.

Checks for the algorithm types matching the key type are handled by the unmarshaling operation.

Table 149

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```
2362    BOOL
2363    CryptObjectIsPublicConsistent(
2364        TPMT_PUBLIC     *publicArea     // IN: public area
2365        )
2366    {
2367        BOOL                OK = TRUE;
2368        switch (publicArea->type)
2369        {
2370    #ifdef TPM_ALG_RSA
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2371            case TPM_ALG_RSA:
2372                OK = CryptAreKeySizesConsistent(publicArea);
2373                break;
2374    #endif //TPM_ALG_RSA
2375
2376    #ifdef TPM_ALG_ECC
2377            case TPM_ALG_ECC:
2378                {
2379                    const ECC_CURVE                *curveValue;
2380
2381                    // Check that the public point is on the indicated curve.
2382                    OK = CryptEccIsPointOnCurve(
2383                                    publicArea->parameters.eccDetail.curveID,
2384                                    &publicArea->unique.ecc);
2385                    if(OK)
2386                    {
2387                        curveValue = CryptEccGetCurveDataPointer(
2388                                                publicArea->parameters.eccDetail.curveID);
2389                        pAssert(curveValue != NULL);
2390
2391                        // The input ECC curve must be a supported curve
2392                        // IF a scheme is defined for the curve, then that scheme must
2393                        // be used.
2394                        OK =    (curveValue->sign.scheme == TPM_ALG_NULL
2395                             || (   publicArea->parameters.eccDetail.scheme.scheme
2396                                 == curveValue->sign.scheme));
2397                        OK = OK && CryptAreKeySizesConsistent(publicArea);
2398                    }
2399                }
2400                break;
2401    #endif //TPM_ALG_ECC
2402
2403            default:
2404                // Symmetric object common checks
2405                // There is noting to check with a symmetric key that is public only.
2406                // Also not sure that there is anything useful to be done with it
2407                // either.
2408                break;
2409        }
2410        return OK;
2411    }
```

11.2.10.14 CryptObjectPublicPrivateMatch()

This function checks the cryptographic binding between the public and sensitive areas.

Table 150

Error Returns	Meaning
TPM_RC_TYPE	the type of the public and private areas are not the same
TPM_RC_FAILURE	crypto error
TPM_RC_BINDING	the public and private areas are not cryptographically matched.

```
2412    TPM_RC
2413    CryptObjectPublicPrivateMatch(
2414        OBJECT          *object         // IN: the object to check
2415        )
2416    {
2417        TPMT_PUBLIC         *publicArea;
2418        TPMT_SENSITIVE      *sensitive;
2419        TPM_RC               result = TPM_RC_SUCCESS;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2420        BOOL                isAsymmetric = FALSE;
2421
2422        pAssert(object != NULL);
2423        publicArea = &object->publicArea;
2424        sensitive = &object->sensitive;
2425        if(publicArea->type != sensitive->sensitiveType)
2426            return TPM_RC_TYPE;
2427
2428        switch(publicArea->type)
2429        {
2430    #ifdef TPM_ALG_RSA
2431        case TPM_ALG_RSA:
2432            isAsymmetric = TRUE;
2433            // The public and private key sizes need to be consistent
2434            if(sensitive->sensitive.rsa.t.size != publicArea->unique.rsa.t.size/2)
2435                result = TPM_RC_BINDING;
2436            else
2437            // Load key by computing the private exponent
2438                result = CryptLoadPrivateRSA(object);
2439            break;
2440    #endif
2441    #ifdef TPM_ALG_ECC
2442            // This function is called from ObjectLoad() which has already checked to
2443            // see that the public point is on the curve so no need to repeat that
2444            // check.
2445        case TPM_ALG_ECC:
2446            isAsymmetric = TRUE;
2447            if(   publicArea->unique.ecc.x.t.size
2448                    != sensitive->sensitive.ecc.t.size)
2449                result = TPM_RC_BINDING;
2450            else if(publicArea->nameAlg != TPM_ALG_NULL)
2451            {
2452                TPMS_ECC_POINT          publicToCompare;
2453                // Compute ECC public key
2454                CryptEccPointMultiply(&publicToCompare,
2455                                      publicArea->parameters.eccDetail.curveID,
2456                                      &sensitive->sensitive.ecc, NULL);
2457                // Compare ECC public key
2458                if(   (!Memory2BEqual(&publicArea->unique.ecc.x.b,
2459                                      &publicToCompare.x.b))
2460                   || (!Memory2BEqual(&publicArea->unique.ecc.y.b,
2461                                      &publicToCompare.y.b)))
2462                    result = TPM_RC_BINDING;
2463            }
2464            break;
2465    #endif
2466        case TPM_ALG_KEYEDHASH:
2467            break;
2468        case TPM_ALG_SYMCIPHER:
2469            if(   (publicArea->parameters.symDetail.sym.keyBits.sym + 7)/8
2470                != sensitive->sensitive.sym.t.size)
2471                result = TPM_RC_BINDING;
2472            break;
2473        default:
2474            // The choice here is an assert or a return of a bad type for the object
2475            pAssert(0);
2476            break;
2477        }
2478
2479        // For asymmetric keys, the algorithm for validating the linkage between
2480        // the public and private areas is algorithm dependent. For symmetric keys
2481        // the linkage is based on hashing the symKey and obfuscation values.
2482        if(   result == TPM_RC_SUCCESS && !isAsymmetric
2483           && publicArea->nameAlg != TPM_ALG_NULL)
2484        {
2485            TPM2B_DIGEST    uniqueToCompare;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2486
2487            // Compute unique for symmetric key
2488            CryptComputeSymmetricUnique(publicArea->nameAlg, sensitive,
2489                                        &uniqueToCompare);
2490            // Compare unique
2491            if(!Memory2BEqual(&publicArea->unique.sym.b,
2492                              &uniqueToCompare.b))
2493                result = TPM_RC_BINDING;
2494        }
2495        return result;
2496
2497    }
```

11.2.10.15 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

```
2498    TPMI_ALG_HASH
2499    CryptGetSignHashAlg(
2500        TPMT_SIGNATURE  *auth           // IN: signature
2501        )
2502    {
2503        pAssert(auth->sigAlg != TPM_ALG_NULL);
2504
2505        // Get authHash algorithm based on signing scheme
2506        switch(auth->sigAlg)
2507        {
2508
2509    #ifdef  TPM_ALG_RSA
2510            case TPM_ALG_RSASSA:
2511                return auth->signature.rsassa.hash;
2512
2513            case TPM_ALG_RSAPSS:
2514                return auth->signature.rsapss.hash;
2515
2516        #endif //TPM_ALG_RSA
2517
2518        #ifdef TPM_ALG_ECC
2519            case TPM_ALG_ECDSA:
2520                return auth->signature.ecdsa.hash;
2521
2522        #endif //TPM_ALG_ECC
2523
2524            case TPM_ALG_HMAC:
2525                return auth->signature.hmac.hashAlg;
2526
2527            default:
2528                return TPM_ALG_NULL;
2529        }
2530    }
```

11.2.10.16 CryptIsSplitSign()

This function us used to determine if the signing operation is a split signing operation that required a TPM2_Commit().

```
2531    BOOL
2532    CryptIsSplitSign(
2533        TPM_ALG_ID       scheme         // IN: the algorithm selector
2534        )
2535    {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2536        if(   scheme != scheme
2537    #   ifdef   TPM_ALG_ECDAA
2538            || scheme == TPM_ALG_ECDAA
2539    #   endif   // TPM_ALG_ECDAA
2540
2541          )
2542            return TRUE;
2543        return FALSE;
2544    }
```

11.2.10.17 CryptIsSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```
2545    BOOL
2546    CryptIsSignScheme(
2547        TPMI_ALG_ASYM_SCHEME     scheme
2548        )
2549    {
2550        BOOL            isSignScheme = FALSE;
2551
2552        switch(scheme)
2553        {
2554    #ifdef TPM_ALG_RSA
2555            // If RSA is implemented, then both signing schemes are required
2556        case TPM_ALG_RSASSA:
2557        case TPM_ALG_RSAPSS:
2558            isSignScheme = TRUE;
2559            break;
2560    #endif //TPM_ALG_RSA
2561
2562    #ifdef TPM_ALG_ECC
2563            // If ECC is implemented ECDSA is required
2564        case TPM_ALG_ECDSA:
2565    #ifdef  TPM_ALG_ECDAA
2566            // ECDAA is optional
2567        case TPM_ALG_ECDAA:
2568    #endif
2569    #ifdef   TPM_ALG_ECSCHNORR
2570            // Schnorr is also optional
2571        case TPM_ALG_ECSCHNORR:
2572    #endif
2573    #ifdef  TPM_ALG_SM2
2574        case TPM_ALG_SM2:
2575    #endif
2576            isSignScheme = TRUE;
2577            break;
2578    #endif //TPM_ALG_ECC
2579        default:
2580            break;
2581        }
2582        return isSignScheme;
2583    }
```

11.2.10.18 CryptIsDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
2584    BOOL
2585    CryptIsDecryptScheme(
2586        TPMI_ALG_ASYM_SCHEME     scheme
2587        )
2588    {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
2589        BOOL        isDecryptScheme = FALSE;
2590
2591        switch(scheme)
2592        {
2593    #ifdef TPM_ALG_RSA
2594            // If RSA is implemented, then both decrypt schemes are required
2595        case TPM_ALG_RSAES:
2596        case TPM_ALG_OAEP:
2597             isDecryptScheme = TRUE;
2598            break;
2599    #endif //TPM_ALG_RSA
2600
2601    #ifdef TPM_ALG_ECC
2602            // If ECC is implemented ECDH is required
2603        case TPM_ALG_ECDH:
2604    #ifdef TPM_ALG_SM2
2605        case TPM_ALG_SM2:
2606    #endif
2607    #ifdef  TPM_ALG_ECMQV
2608        case TPM_ALG_ECMQV:
2609    #endif
2610            isDecryptScheme = TRUE;
2611            break;
2612    #endif //TPM_ALG_ECC
2613        default:
2614            break;
2615        }
2616        return isDecryptScheme;
2617    }
```

11.2.10.19 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.

Table 151

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	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*

```
2618    TPM_RC
2619    CryptSelectSignScheme(
2620        TPMI_DH_OBJECT       signHandle,    // IN: handle of signing key
2621        TPMT_SIG_SCHEME     *scheme         // IN/OUT: signing scheme
2622        )
2623    {
2624        OBJECT              *signObject;
2625        TPMT_SIG_SCHEME     *objectScheme;
2626        TPMT_PUBLIC         *publicArea;
2627        TPM_RC               result = TPM_RC_SUCCESS;
2628
2629        // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
2630        // of the setting of scheme
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2631        if(signHandle == TPM_RH_NULL)
2632        {
2633            scheme->scheme = TPM_ALG_NULL;
2634            scheme->details.any.hashAlg = TPM_ALG_NULL;
2635        }
2636        else
2637        {
2638            // sign handle is not NULL so...
2639            // Get sign object pointer
2640            signObject = ObjectGet(signHandle);
2641            publicArea = &signObject->publicArea;
2642
2643            // is this a signing key?
2644            if(!publicArea->objectAttributes.sign)
2645                result = TPM_RC_KEY;
2646            else
2647            {
2648                // "parms" defined to avoid long code lines.
2649                TPMU_PUBLIC_PARMS   *parms = &publicArea->parameters;
2650                if(CryptIsAsymAlgorithm(publicArea->type))
2651                    objectScheme = (TPMT_SIG_SCHEME *)&parms->asymDetail.scheme;
2652                else
2653                    objectScheme = (TPMT_SIG_SCHEME *)&parms->keyedHashDetail.scheme;
2654
2655                // If the object doesn't have a default scheme, then use the
2656                // input scheme.
2657                if(objectScheme->scheme == TPM_ALG_NULL)
2658                {
2659                    // Input and default can't both be NULL
2660                    if(scheme->scheme == TPM_ALG_NULL)
2661                        result = TPM_RC_SCHEME;
2662
2663                    // Assume that the scheme is compatible with the key. If not,
2664                    // we will generate an error in the signing operation.
2665
2666                }
2667                else if(scheme->scheme == TPM_ALG_NULL)
2668                {
2669                    // input scheme is NULL so use default
2670
2671                    // First, check to see if the default requires that the caller
2672                    // provided scheme data
2673                    if(CryptIsSplitSign(objectScheme->scheme))
2674                        result = TPM_RC_SCHEME;
2675                    else
2676                    {
2677                        scheme->scheme = objectScheme->scheme;
2678                        scheme->details.any.hashAlg
2679                                    = objectScheme->details.any.hashAlg;
2680                    }
2681                }
2682                else
2683                {
2684                    // Both input and object have scheme selectors
2685                    // If the scheme and the hash are not the same then...
2686                    if(   objectScheme->scheme != scheme->scheme
2687                       || (   objectScheme->details.any.hashAlg
2688                           != scheme->details.any.hashAlg))
2689                        result = TPM_RC_SCHEME;
2690                }
2691            }
2692
2693        }
2694        return result;
2695    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

11.2.10.20 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 152

Error Returns	Meaning
TPM_RC_SCHEME	*signScheme* is not compatible with the signing key type
TPM_RC_VALUE	*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)

```
2696    TPM_RC
2697    CryptSign(
2698        TPMI_DH_OBJECT       signHandle,     // IN: The handle of sign key
2699        TPMT_SIG_SCHEME     *signScheme,     // IN: sign scheme.
2700        TPM2B_DIGEST        *digest,         // IN: The digest being signed
2701        TPMT_SIGNATURE      *signature       // OUT: signature
2702        )
2703    {
2704        OBJECT              *signKey = ObjectGet(signHandle);
2705        TPM_RC               result = TPM_RC_SCHEME;
2706
2707        // check if input handle is a sign key
2708        pAssert(signKey->publicArea.objectAttributes.sign == SET);
2709
2710        // Must have the private portion loaded.  This check is made during
2711        // authorization.
2712        pAssert(signKey->attributes.publicOnly == CLEAR);
2713
2714        // Initialize signature scheme
2715        signature->sigAlg = signScheme->scheme;
2716
2717        // If the signature algorithm is TPM_ALG_NULL, then we are done
2718        if(signature->sigAlg == TPM_ALG_NULL)
2719            return TPM_RC_SUCCESS;
2720
2721        // All the schemes other than TPM_ALG_NULL have a hash algorithm
2722        TEST_HASH(signScheme->details.any.hashAlg);
2723
2724        // Initialize signature hash
2725        // Note: need to do the check for alg null first because the null scheme
2726        // doesn't have a hashAlg member.
2727        signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
2728
2729        // perform sign operation based on different key type
2730        switch (signKey->publicArea.type)
2731        {
2732
2733    #ifdef TPM_ALG_RSA
2734            case TPM_ALG_RSA:
2735                result = CryptSignRSA(signKey, signScheme, digest, signature);
2736                break;
2737    #endif //TPM_ALG_RSA
2738
2739    #ifdef TPM_ALG_ECC
2740            case TPM_ALG_ECC:
2741                result = CryptSignECC(signKey, signScheme, digest, signature);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2742            break;
2743    #endif //TPM_ALG_ECC
2744            case TPM_ALG_KEYEDHASH:
2745                result = CryptSignHMAC(signKey, signScheme, digest, signature);
2746                break;
2747            default:
2748                break;
2749        }
2750
2751        return result;
2752    }
```

11.2.10.21 CryptVerifySignature()

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.

This function requires that *auth* is not a NULL pointer.

Table 153

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SCHEME	the scheme is not supported
TPM_RC_HANDLE	an HMAC key was selected but the private part of the key is not loaded

```
2753    TPM_RC
2754    CryptVerifySignature(
2755        TPMI_DH_OBJECT   keyHandle,     // IN: The handle of sign key
2756        TPM2B_DIGEST    *digest,        // IN: The digest being validated
2757        TPMT_SIGNATURE  *signature      // IN: signature
2758        )
2759    {
2760        // NOTE: ObjectGet will either return a pointer to a loaded object or
2761        // will assert. It will never return a non-valid value. This makes it save
2762        // to initialize 'publicArea' with the return value from ObjectGet() without
2763        // checking it first.
2764        OBJECT              *authObject = ObjectGet(keyHandle);
2765        TPMT_PUBLIC         *publicArea = &authObject->publicArea;
2766        TPM_RC               result = TPM_RC_SCHEME;
2767
2768        // The input unmarshaling should prevent any input signature from being
2769        // a NULL signature, but just in case
2770        if(signature->sigAlg == TPM_ALG_NULL)
2771            return TPM_RC_SIGNATURE;
2772
2773        switch (publicArea->type)
2774        {
2775
2776    #ifdef TPM_ALG_RSA
2777        case TPM_ALG_RSA:
2778            result = CryptRSAVerifySignature(authObject, digest, signature);
2779            break;
2780    #endif //TPM_ALG_RSA
2781
2782    #ifdef TPM_ALG_ECC
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2783        case TPM_ALG_ECC:
2784            result = CryptECCVerifySignature(authObject, digest, signature);
2785            break;
2786
2787    #endif // TMP_ALG_ECC
2788
2789        case TPM_ALG_KEYEDHASH:
2790            if(authObject->attributes.publicOnly)
2791                result = TPM_RCS_HANDLE;
2792            else
2793                result = CryptHMACVerifySignature(authObject, digest, signature);
2794            break;
2795
2796        default:
2797            break;
2798        }
2799        return result;
2800
2801    }
```

11.2.11 Math functions

11.2.11.1 CryptDivide()

This function interfaces to the math library for large number divide.

Table 154

Error Returns	Meaning
TPM_RC_SIZE	*quotient* or *remainder* is too small to receive the result

```
2802    TPM_RC
2803    CryptDivide(
2804        TPM2B           *numerator,     // IN: numerator
2805        TPM2B           *denominator,   // IN: denominator
2806        TPM2B           *quotient,      // OUT: quotient = numerator / denominator.
2807        TPM2B           *remainder      // OUT: numerator mod denominator.
2808        )
2809    {
2810        pAssert(   numerator != NULL && denominator!= NULL
2811                && (quotient != NULL || remainder != NULL)
2812               );
2813        // assume denominator is not 0
2814        pAssert(denominator->size != 0);
2815
2816        return TranslateCryptErrors(_math__Div(numerator,
2817                                               denominator,
2818                                               quotient,
2819                                               remainder)
2820                                    );
2821    }
```

11.2.11.2 CryptCompare()

This function interfaces to the math library for large number, unsigned compare.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table 155

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
2822    LIB_EXPORT int
2823    CryptCompare(
2824        const UINT32     aSize,          // IN: size of a
2825        const BYTE      *a,              // IN: a buffer
2826        const UINT32     bSize,          // IN: size of b
2827        const BYTE      *b               // IN: b buffer
2828        )
2829    {
2830        return _math__uComp(aSize, a, bSize, b);
2831    }
```

11.2.11.3 CryptCompareSigned()

This function interfaces to the math library for large number, signed compare.

Table 156

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
2832    int
2833    CryptCompareSigned(
2834        UINT32           aSize,          // IN: size of a
2835        BYTE            *a,              // IN: a buffer
2836        UINT32           bSize,          // IN: size of b
2837        BYTE            *b               // IN: b buffer
2838        )
2839    {
2840        return _math__Comp(aSize, a, bSize, b);
2841    }
```

11.2.11.4 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 ISO/IEC 11889-3, clause 11.4, “TPM2_GetTestResult”.

```
2842    TPM_RC
2843    CryptGetTestResult(
2844        TPM2B_MAX_BUFFER    *outData        // OUT: test result data
2845        )
2846    {
2847        outData->t.size = 0;
2848        return TPM_RC_SUCCESS;
2849    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

11.2.12 Capability Support

11.2.12.1 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

Table 157

Return Value	Meaning
YES	if no more ECC curve is available
NO	if there are more ECC curves not reported

```
2850    #ifdef TPM_ALG_ECC //% 5
2851    TPMI_YES_NO
2852    CryptCapGetECCCurve(
2853        TPM_ECC_CURVE    curveID,       // IN: the starting ECC curve
2854        UINT32           maxCount,      // IN: count of returned curves
2855        TPML_ECC_CURVE  *curveList      // OUT: ECC curve list
2856        )
2857    {
2858        TPMI_YES_NO       more = NO;
2859        UINT16            i;
2860        UINT32            count = _cpri__EccGetCurveCount();
2861        TPM_ECC_CURVE     curve;
2862
2863        // Initialize output property list
2864        curveList->count = 0;
2865
2866        // The maximum count of curves we may return is MAX_ECC_CURVES
2867        if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
2868
2869        // Scan the eccCurveValues array
2870        for(i = 0; i < count; i++)
2871        {
2872            curve = _cpri__GetCurveIdByIndex(i);
2873            // If curveID is less than the starting curveID, skip it
2874            if(curve < curveID)
2875                continue;
2876
2877            if(curveList->count < maxCount)
2878            {
2879                // If we have not filled up the return list, add more curves to
2880                // it
2881                curveList->eccCurves[curveList->count] = curve;
2882                curveList->count++;
2883            }
2884            else
2885            {
2886                // If the return list is full but we still have curves
2887                // available, report this and stop iterating
2888                more = YES;
2889                break;
2890            }
2891
2892        }
2893
2894        return more;
2895
2896    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

11.2.12.2 CryptCapGetEccCurveNumber()

This function returns the number of ECC curves supported by the TPM.

```
2897    UINT32
2898    CryptCapGetEccCurveNumber(
2899        void
2900        )
2901    {
2902        // There is an array that holds the curve data. Its size divided by the
2903        // size of an entry is the number of values in the table.
2904        return _cpri__EccGetCurveCount();
2905    }
2906    #endif //TPM_ALG_ECC //% 5
```

11.2.12.3 CryptAreKeySizesConsistent()

This function validates that the public key size values are consistent for an asymmetric key.

NOTE This is not a comprehensive test of the public key.

Table 158

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```
2907    BOOL
2908    CryptAreKeySizesConsistent(
2909        TPMT_PUBLIC     *publicArea     // IN: the public area to check
2910        )
2911    {
2912        BOOL            consistent = FALSE;
2913
2914        switch (publicArea->type)
2915        {
2916    #ifdef TPM_ALG_RSA
2917            case TPM_ALG_RSA:
2918                // The key size in bits is filtered by the unmarshaling
2919                consistent =  (    ((publicArea->parameters.rsaDetail.keyBits+7)/8)
2920                                == publicArea->unique.rsa.t.size);
2921                break;
2922    #endif //TPM_ALG_RSA
2923
2924    #ifdef TPM_ALG_ECC
2925            case TPM_ALG_ECC:
2926                {
2927                    UINT16          keySizeInBytes;
2928                    TPM_ECC_CURVE   curveId = publicArea->parameters.eccDetail.curveID;
2929
2930                    keySizeInBytes = CryptEccGetKeySizeInBytes(curveId);
2931
2932                    consistent =   keySizeInBytes > 0
2933                                && publicArea->unique.ecc.x.t.size <= keySizeInBytes
2934                                && publicArea->unique.ecc.y.t.size <= keySizeInBytes;
2935                }
2936                break;
2937    #endif //TPM_ALG_ECC
2938            default:
2939                break;
2940        }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
2941
2942        return consistent;
2943    }
```

11.2.12.4 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 ISO/IEC 11889-2 parser so that the *g_implementedAlgorithms* vector can be a const. That's not how it is now.

```
2944    void
2945    CryptAlgsSetImplemented(
2946        void
2947        )
2948    {
2949        AlgorithmGetImplementedVector(&g_implementedAlgorithms);
2950    }
```

11.3 Ticket.c

11.3.1 Introduction

Clause 11.3.3 contains the functions used for ticket computations.

11.3.2 Includes

```
1   #include "InternalRoutines.h"
```

11.3.3 Functions

11.3.3.1 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 159

Return Value	Meaning
TRUE	It is safe to produce ticket
FALSE	It is not safe to produce ticket

```
2   BOOL
3   TicketIsSafe(
4       TPM2B           *buffer
5       )
6   {
7       TPM_GENERATED   valueToCompare = TPM_GENERATED_VALUE;
8       BYTE            bufferToCompare[sizeof(valueToCompare)];
9       BYTE            *marshalBuffer;
10
11      // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
12      // it is not safe to generate a ticket
13      if(buffer->size < <K>sizeof(valueToCompare))
14          return FALSE;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
15
16      marshalBuffer = bufferToCompare;
17      TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);
18      if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
19          return FALSE;
20      else
21          return TRUE;
22  }
```

11.3.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```
23  void
24  TicketComputeVerified(
25      TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
26      TPM2B_DIGEST        *digest,         // IN: digest
27      TPM2B_NAME          *keyName,        // IN: name of key that signed the values
28      TPMT_TK_VERIFIED    *ticket          // OUT: verified ticket
29      )
30  {
31      TPM2B_AUTH          *proof;
32      HMAC_STATE           hmacState;
33
34      // Fill in ticket fields
35      ticket->tag = TPM_ST_VERIFIED;
36      ticket->hierarchy = hierarchy;
37
38      // Use the proof value of the hierarchy
39      proof = HierarchyGetProof(hierarchy);
40
41      // Start HMAC
42      ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
43                                               &proof->b, &hmacState);
44
45      // add TPM_ST_VERIFIED
46      CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
47
48      // add digest
49      CryptUpdateDigest2B(&hmacState, &digest->b);
50
51      // add key name
52      CryptUpdateDigest2B(&hmacState, &keyName->b);
53
54      // complete HMAC
55      CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
56
57      return;
58  }
```

11.3.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```
59  void
60  TicketComputeAuth(
61      TPM_ST               type,           // IN: the type of ticket.
62      TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
63      UINT64               timeout,        // IN: timeout
64      TPM2B_DIGEST        *cpHashA,        // IN: input cpHashA
65      TPM2B_NONCE         *policyRef,      // IN: input policyRef
66      TPM2B_NAME          *entityName,     // IN: name of entity
67      TPMT_TK_AUTH        *ticket          // OUT: Created ticket
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
68      )
69  {
70      TPM2B_AUTH          *proof;
71      HMAC_STATE           hmacState;
72
73      // Get proper proof
74      proof = HierarchyGetProof(hierarchy);
75
76      // Fill in ticket fields
77      ticket->tag = type;
78      ticket->hierarchy = hierarchy;
79
80      // Start HMAC
81      ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
82                                               &proof->b, &hmacState);
83
84      // Adding TPM_ST_AUTH
85      CryptUpdateDigestInt(&hmacState, sizeof(UINT16), &ticket->tag);
86
87      // Adding timeout
88      CryptUpdateDigestInt(&hmacState, sizeof(UINT64), &timeout);
89
90      // Adding cpHash
91      CryptUpdateDigest2B(&hmacState, &cpHashA->b);
92
93      // Adding policyRef
94      CryptUpdateDigest2B(&hmacState, &policyRef->b);
95
96      // Adding keyName
97      CryptUpdateDigest2B(&hmacState, &entityName->b);
98
99      // Compute HMAC
100     CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
101
102     return;
103 }
```

11.3.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```
104 void
105 TicketComputeHashCheck(
106     TPMI_RH_HIERARCHY    hierarchy,     // IN: hierarchy constant for ticket
107     TPM_ALG_ID           hashAlg,       // IN: the hash algorithm used to create
108                                         //     'digest'
109     TPM2B_DIGEST        *digest,        // IN: input digest
110     TPMT_TK_HASHCHECK   *ticket         // OUT: Created ticket
111     )
112 {
113     TPM2B_AUTH          *proof;
114     HMAC_STATE           hmacState;
115
116     // Get proper proof
117     proof = HierarchyGetProof(hierarchy);
118
119     // Fill in ticket fields
120     ticket->tag = TPM_ST_HASHCHECK;
121     ticket->hierarchy = hierarchy;
122
123     ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
124                                              &proof->b, &hmacState);
125
126     // Add TPM_ST_HASHCHECK
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
127         CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
128
129         // Add hash algorithm
130         CryptUpdateDigestInt(&hmacState, sizeof(hashAlg), &hashAlg);
131
132         // Add digest
133         CryptUpdateDigest2B(&hmacState, &digest->b);
134
135         // Compute HMAC
136         CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
137
138         return;
139     }
```

11.3.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```
140     void
141     TicketComputeCreation(
142         TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy for ticket
143         TPM2B_NAME          *name,           // IN: object name
144         TPM2B_DIGEST        *creation,       // IN: creation hash
145         TPMT_TK_CREATION    *ticket          // OUT: created ticket
146         )
147     {
148         TPM2B_AUTH          *proof;
149         HMAC_STATE           hmacState;
150
151         // Get proper proof
152         proof = HierarchyGetProof(hierarchy);
153
154         // Fill in ticket fields
155         ticket->tag = TPM_ST_CREATION;
156         ticket->hierarchy = hierarchy;
157
158         ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
159                                                  &proof->b, &hmacState);
160
161         // Add TPM_ST_CREATION
162         CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
163
164         // Add name
165         CryptUpdateDigest2B(&hmacState, &name->b);
166
167         // Add creation hash
168         CryptUpdateDigest2B(&hmacState, &creation->b);
169
170         // Compute HMAC
171         CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
172
173         return;
174     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

11.4 CryptSelfTest.c

11.4.1 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.

© ISO/IEC 2015 – All rights reserved

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.

```
#include     "Global.h"
#include     "CryptoEngine.h"
#include     "InternalRoutines.h"
#include     "AlgorithmCap_fp.h"
#include     "CryptSelfTest_fp.h"
#include     "AlgorithmTests_fp.h"
```

11.4.2 Functions

11.4.2.1 RunSelfTest()

Local function to run self-test

```
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;
}
```

11.4.2.2 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.

Table 160

Error Returns	Meaning
TPM_RC_CANCELED	if the command is canceled

```
LIB_EXPORT
TPM_RC
CryptSelfTest(
    TPMI_YES_NO      fullTest       // IN: if full test is required
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
    )
{
    if(g_forceFailureMode)
        FAIL(FATAL_ERROR_FORCED);

    // If the caller requested a full test, then reset the to test vector so that
    // all the tests will be run
    if(fullTest == YES)
    {
        MemoryCopy(g_toTest,
                   g_implementedAlgorithms,
                   sizeof(g_toTest), sizeof(g_toTest));
    }
    return CryptRunSelfTests(&g_toTest);
}
```

11.4.2.3 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 161

Error Returns	Meaning
TPM_RC_CANCELED	processing of this command was canceled
TPM_RC_TESTING	if *toTest* list is not empty
TPM_RC_VALUE	an algorithm in the *toTest* list is not implemented

```
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++)
        {
            TPM_ALG_ID      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))
                return TPM_RC_VALUE;
            SET_BIT(alg, toTestVector);
        }
        // Run the test
        if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
            return TPM_RC_CANCELED;
    }
    // Fill in the toDoList with the algorithms that are still untested
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
75      toDoList->count = 0;
76
77      for(alg = TPM_ALG_FIRST;
78          toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
79          alg++)
80      {
81          if(TEST_BIT(alg, g_toTest))
82              toDoList->algorithms[toDoList->count++] = alg;
83      }
84      return TPM_RC_SUCCESS;
85  }
```

11.4.2.4 CryptInitializeToTest()

This function will initialize the data structures for testing all the algorithms. This should not be called unless CryptAlgsSetImplemented() has been called

```
86  void
87  CryptInitializeToTest(
88      void
89      )
90  {
91      MemoryCopy(g_toTest,
92                 g_implementedAlgorithms,
93                 sizeof(g_toTest),
94                 sizeof(g_toTest));
95      // Setting the algorithm to null causes the test function to just clear
96      // out any algorithms for which there is no test.
97      CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);
98
99      return;
100 }
```

11.4.2.5 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 algorithms 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 TPM_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 162

Error Returns	Meaning
TPM_RC_SUCCESS	test complete
TPM_RC_CANCELED	test was canceled

```
101 LIB_EXPORT
102 TPM_RC
103 CryptTestAlgorithm(
104     TPM_ALG_ID           alg,
105     ALGORITHM_VECTOR    *toTest
106     )
107 {
108     TPM_RC                  result = TPM_RC_SUCCESS;
109 #ifdef SELF_TEST
110     TPM_RC TestAlgorithm(TPM_ALG_ID alg, ALGORITHM_VECTOR *toTest);
111     result = TestAlgorithm(alg, toTest);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
112     #else
113         // If this is an attempt to determine the algorithms for which there is a
114         // self test, pretend that all of them do. We do that by not clearing any
115         // of the algorithm bits. When/if this function is called to run tests, it
116         // will over report. This can be changed so that any call to check on which
117         // algorithms have tests, 'toTest' can be cleared.
118         if(alg != TPM_ALG_ERROR)
119         {
120             CLEAR_BIT(alg, g_toTest);
121             if(toTest != NULL)
122                 CLEAR_BIT(alg, *toTest);
123         }
124     #endif
125         return result;
126     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are derived from the values in the annexes of ISO/IEC 11889-2. This file would change based on the implementation.

The values shown in this version of the file reflect the example settings in ISO/IEC 11889-2.

A.2 Implementation.h

```
#ifndef _IMPLEMENTATION_H
#define _IMPLEMENTATION_H
    #include    "BaseTypes.h"
#undef TRUE
#undef FALSE
```

Change these definitions to turn all algorithms or commands on or off

```
#define      ALG_YES      YES
#define      ALG_NO       NO
#define      CC_YES       YES
#define      CC_NO        NO
```

From the ISO/IEC 11889-2, Table A.1, “Defines for SHA1 Hash Values”

```
#define  SHA1_DIGEST_SIZE    20
#define  SHA1_BLOCK_SIZE     64
#define  SHA1_DER_SIZE       15
#define  SHA1_DER            {\
    0x30,0x21,0x30,0x09,0x06,0x05,0x2B,0x0E,0x03,0x02,0x1A,0x05,0x00,0x04,0x14}
```

From the ISO/IEC 11889-2, Table A.2, “Defines for SHA256 Hash Values”

```
#define  SHA256_DIGEST_SIZE    32
#define  SHA256_BLOCK_SIZE     64
#define  SHA256_DER_SIZE       19
#define  SHA256_DER            {\
    0x30,0x31,0x30,0x0D,0X06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X01,    \
    0X05,0X00,0X04,0X20}
```

From the ISO/IEC 11889-2, Table A.3, “Defines for SHA384 Hash Values”

```
#define  SHA384_DIGEST_SIZE    48
#define  SHA384_BLOCK_SIZE     128
#define  SHA384_DER_SIZE       19
#define  SHA384_DER            {\
    0x30,0x41,0x30,0x0D,0X06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X02,    \
    0X05,0X00,0X04,0X30}
```

From the ISO/IEC 11889-2, Table A.4, “Defines for SHA512 Hash Values”

```
#define  SHA512_DIGEST_SIZE    64
#define  SHA512_BLOCK_SIZE     128
#define  SHA512_DER_SIZE       19
#define  SHA512_DER            {\
    0x30,0x51,0x30,0x0D,0X06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X03,    \
    0X05,0X00,0X04,0X40}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

From the ISO/IEC 11889-2, Table A.5, “SM3_256 Hash Values”

```
33  #define  SM3_256_DIGEST_SIZE    32
34  #define  SM3_256_BLOCK_SIZE     64
35  #define  SM3_256_DER_SIZE       18
36  #define  SM3_256_DER            {\
37      0x30,0x30,0x30,0x0C,0X06,0X08,0X2A,0X81,0X1C,0X81,0X45,0X01,0X83,0X11,0X05,    \
38      0X00,0X04,0X20}
```

From the ISO/IEC 11889-2, Table A.6, “Defines for Architectural Limits Values”

```
39  #define  MAX_SESSION_NUMBER    3
```

From the ISO/IEC 11889-2, Table B.1, “Defines for Logic Values”

```
40  #define  YES      1
41  #define  NO       0
42  #define  TRUE     1
43  #define  FALSE    0
44  #define  SET      1
45  #define  CLEAR    0
```

From the ISO/IEC 11889-2, Table B.2, “Defines for Processor Values”

```
46  #define  BIG_ENDIAN_TPM       NO
47  #define  LITTLE_ENDIAN_TPM    YES
48  #define  NO_AUTO_ALIGN        NO
```

From the ISO/IEC 11889-2, Table B.3, “Defines for Implemented Algorithms Implemented”

```
49  #define  ALG_RSA              ALG_YES
50  #define  ALG_SHA1             ALG_YES
51  #define  ALG_HMAC             ALG_YES
52  #define  ALG_AES              ALG_YES
53  #define  ALG_MGF1             ALG_YES
54  #define  ALG_XOR              ALG_YES
55  #define  ALG_KEYEDHASH        ALG_YES
56  #define  ALG_SHA256           ALG_YES
57  #define  ALG_SHA384           ALG_YES
58  #define  ALG_SHA512           ALG_NO
59  #define  ALG_SM3_256          ALG_NO
60  #define  ALG_SM4              ALG_NO
61  #define  ALG_RSASSA           (ALG_YES*ALG_RSA)
62  #define  ALG_RSAES            (ALG_YES*ALG_RSA)
63  #define  ALG_RSAPSS           (ALG_YES*ALG_RSA)
64  #define  ALG_OAEP             (ALG_YES*ALG_RSA)
65  #define  ALG_ECC              ALG_YES
66  #define  ALG_ECDH             (ALG_YES*ALG_ECC)
67  #define  ALG_ECDSA            (ALG_YES*ALG_ECC)
68  #define  ALG_ECDAA            (ALG_YES*ALG_ECC)
69  #define  ALG_SM2              (ALG_NO*ALG_ECC)
70  #define  ALG_ECSCHNORR        (ALG_YES*ALG_ECC)
71  #define  ALG_ECMQV            (ALG_NO*ALG_ECC)
72  #define  ALG_SYMCIPHER        ALG_YES
73  #define  ALG_CAMELLIA         ALG_YES
74  #define  ALG_KDF1_SP800_56a   (ALG_YES*ALG_ECC)
75  #define  ALG_KDF2             ALG_NO
76  #define  ALG_KDF1_SP800_108   ALG_YES
77  #define  ALG_CTR              ALG_YES
78  #define  ALG_OFB              ALG_YES
79  #define  ALG_CBC              ALG_YES
80  #define  ALG_CFB              ALG_YES
81  #define  ALG_ECB              ALG_YES
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

From the ISO/IEC 11889-2, Table B.4, “Defines for Implemented Commands Implemented”

```
 82  #define  CC_ActivateCredential           CC_YES
 83  #define  CC_Certify                      CC_YES
 84  #define  CC_CertifyCreation              CC_YES
 85  #define  CC_ChangeEPS                    CC_YES
 86  #define  CC_ChangePPS                    CC_YES
 87  #define  CC_Clear                        CC_YES
 88  #define  CC_ClearControl                 CC_YES
 89  #define  CC_ClockRateAdjust              CC_YES
 90  #define  CC_ClockSet                     CC_YES
 91  #define  CC_Commit                       ALG_ECC
 92  #define  CC_ContextLoad                  CC_YES
 93  #define  CC_ContextSave                  CC_YES
 94  #define  CC_Create                       CC_YES
 95  #define  CC_CreatePrimary                CC_YES
 96  #define  CC_DictionaryAttackLockReset    CC_YES
 97  #define  CC_DictionaryAttackParameters   CC_YES
 98  #define  CC_Duplicate                    CC_YES
 99  #define  CC_ECC_Parameters               ALG_ECC
100  #define  CC_ECDH_KeyGen                  ALG_ECC
101  #define  CC_ECDH_ZGen                    ALG_ECC
102  #define  CC_EncryptDecrypt               CC_YES
103  #define  CC_EventSequenceComplete        CC_YES
104  #define  CC_EvictControl                 CC_YES
105  #define  CC_FieldUpgradeData             CC_NO
106  #define  CC_FieldUpgradeStart            CC_NO
107  #define  CC_FirmwareRead                 CC_NO
108  #define  CC_FlushContext                 CC_YES
109  #define  CC_GetCapability                CC_YES
110  #define  CC_GetCommandAuditDigest        CC_YES
111  #define  CC_GetRandom                    CC_YES
112  #define  CC_GetSessionAuditDigest        CC_YES
113  #define  CC_GetTestResult                CC_YES
114  #define  CC_GetTime                      CC_YES
115  #define  CC_Hash                         CC_YES
116  #define  CC_HashSequenceStart            CC_YES
117  #define  CC_HierarchyChangeAuth          CC_YES
118  #define  CC_HierarchyControl             CC_YES
119  #define  CC_HMAC                         CC_YES
120  #define  CC_HMAC_Start                   CC_YES
121  #define  CC_Import                       CC_YES
122  #define  CC_IncrementalSelfTest          CC_YES
123  #define  CC_Load                         CC_YES
124  #define  CC_LoadExternal                 CC_YES
125  #define  CC_MakeCredential               CC_YES
126  #define  CC_NV_Certify                   CC_YES
127  #define  CC_NV_ChangeAuth                CC_YES
128  #define  CC_NV_DefineSpace               CC_YES
129  #define  CC_NV_Extend                    CC_YES
130  #define  CC_NV_GlobalWriteLock           CC_YES
131  #define  CC_NV_Increment                 CC_YES
132  #define  CC_NV_Read                      CC_YES
133  #define  CC_NV_ReadLock                  CC_YES
134  #define  CC_NV_ReadPublic                CC_YES
135  #define  CC_NV_SetBits                   CC_YES
136  #define  CC_NV_UndefineSpace             CC_YES
137  #define  CC_NV_UndefineSpaceSpecial      CC_YES
138  #define  CC_NV_Write                     CC_YES
139  #define  CC_NV_WriteLock                 CC_YES
140  #define  CC_ObjectChangeAuth             CC_YES
141  #define  CC_PCR_Allocate                 CC_YES
142  #define  CC_PCR_Event                    CC_YES
143  #define  CC_PCR_Extend                   CC_YES
144  #define  CC_PCR_Read                     CC_YES
145  #define  CC_PCR_Reset                    CC_YES
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
#define  CC_PCR_SetAuthPolicy            CC_YES
#define  CC_PCR_SetAuthValue             CC_YES
#define  CC_PolicyAuthorize              CC_YES
#define  CC_PolicyAuthValue              CC_YES
#define  CC_PolicyCommandCode            CC_YES
#define  CC_PolicyCounterTimer           CC_YES
#define  CC_PolicyCpHash                 CC_YES
#define  CC_PolicyDuplicationSelect      CC_YES
#define  CC_PolicyGetDigest              CC_YES
#define  CC_PolicyLocality               CC_YES
#define  CC_PolicyNameHash               CC_YES
#define  CC_PolicyNV                     CC_YES
#define  CC_PolicyOR                     CC_YES
#define  CC_PolicyPassword               CC_YES
#define  CC_PolicyPCR                    CC_YES
#define  CC_PolicyPhysicalPresence       CC_YES
#define  CC_PolicyRestart                CC_YES
#define  CC_PolicySecret                 CC_YES
#define  CC_PolicySigned                 CC_YES
#define  CC_PolicyTicket                 CC_YES
#define  CC_PP_Commands                  CC_YES
#define  CC_Quote                        CC_YES
#define  CC_ReadClock                    CC_YES
#define  CC_ReadPublic                   CC_YES
#define  CC_Rewrap                       CC_YES
#define  CC_RSA_Decrypt                  ALG_RSA
#define  CC_RSA_Encrypt                  ALG_RSA
#define  CC_SelfTest                     CC_YES
#define  CC_SequenceComplete             CC_YES
#define  CC_SequenceUpdate               CC_YES
#define  CC_SetAlgorithmSet              CC_YES
#define  CC_SetCommandCodeAuditStatus    CC_YES
#define  CC_SetPrimaryPolicy             CC_YES
#define  CC_Shutdown                     CC_YES
#define  CC_Sign                         CC_YES
#define  CC_StartAuthSession             CC_YES
#define  CC_Startup                      CC_YES
#define  CC_StirRandom                   CC_YES
#define  CC_TestParms                    CC_YES
#define  CC_Unseal                       CC_YES
#define  CC_VerifySignature              CC_YES
#define  CC_ZGen_2Phase                  CC_YES
#define  CC_EC_Ephemeral                 CC_YES
#define  CC_PolicyNvWritten              CC_YES
```

From the ISO/IEC 11889-2, Table B.5, “Defines for RSA Algorithm Constants”

```
#define  RSA_KEY_SIZES_BITS    {1024,2048}
#define  RSA_KEY_SIZE_BITS_1024
#define  RSA_KEY_SIZE_BITS_2048
#define  MAX_RSA_KEY_BITS      2048
#define  MAX_RSA_KEY_BYTES     ((MAX_RSA_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.6, “Defines for ECC Algorithm Constants”

```
#define  ECC_CURVES            {TPM_ECC_NIST_P256,TPM_ECC_BN_P256,TPM_ECC_SM2_P256}
#define  ECC_KEY_SIZES_BITS    {256}
#define  ECC_KEY_SIZE_BITS_256
#define  MAX_ECC_KEY_BITS      256
#define  MAX_ECC_KEY_BYTES     ((MAX_ECC_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.7 “Defines for AES Algorithm Constants”

```
#define  AES_KEY_SIZES_BITS          {128,256}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
#define  AES_KEY_SIZE_BITS_128
#define  AES_KEY_SIZE_BITS_256
#define  MAX_AES_KEY_BITS            256
#define  MAX_AES_BLOCK_SIZE_BYTES    16
#define  MAX_AES_KEY_BYTES           ((MAX_AES_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.8, “Defines for SM4 Algorithm Constants”

```
#define  SM4_KEY_SIZES_BITS          {128}
#define  SM4_KEY_SIZE_BITS_128
#define  MAX_SM4_KEY_BITS            128
#define  MAX_SM4_BLOCK_SIZE_BYTES    16
#define  MAX_SM4_KEY_BYTES           ((MAX_SM4_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.9, “Defines for CAMELLIA Algorithm Constants”

```
#define  CAMELLIA_KEY_SIZES_BITS          {128}
#define  CAMELLIA_KEY_SIZE_BITS_128
#define  MAX_CAMELLIA_KEY_BITS            128
#define  MAX_CAMELLIA_BLOCK_SIZE_BYTES    16
#define  MAX_CAMELLIA_KEY_BYTES           ((MAX_CAMELLIA_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.10, “Defines for Symmetric Algorithm Constants”

```
#define  MAX_SYM_KEY_BITS       MAX_AES_KEY_BITS
#define  MAX_SYM_KEY_BYTES      MAX_AES_KEY_BYTES
#define  MAX_SYM_BLOCK_SIZE     MAX_AES_BLOCK_SIZE_BYTES
```

From the ISO/IEC 11889-2, Table B.11, “Defines for Implementation Values”

```
#define  FIELD_UPGRADE_IMPLEMENTED        NO
#define  BSIZE                            UINT16
#define  BUFFER_ALIGNMENT                 4
#define  IMPLEMENTATION_PCR               24
#define  PLATFORM_PCR                     24
#define  DRTM_PCR                         17
#define  HCRTM_PCR                        0
#define  NUM_LOCALITIES                   5
#define  MAX_HANDLE_NUM                   3
#define  MAX_ACTIVE_SESSIONS              64
#define  CONTEXT_SLOT                     UINT16
#define  CONTEXT_COUNTER                  UINT64
#define  MAX_LOADED_SESSIONS              3
#define  MAX_SESSION_NUM                  3
#define  MAX_LOADED_OBJECTS               3
#define  MIN_EVICT_OBJECTS                2
#define  PCR_SELECT_MIN                   ((PLATFORM_PCR+7)/8)
#define  PCR_SELECT_MAX                   ((IMPLEMENTATION_PCR+7)/8)
#define  NUM_POLICY_PCR_GROUP             1
#define  NUM_AUTHVALUE_PCR_GROUP          1
#define  MAX_CONTEXT_SIZE                 2048
#define  MAX_DIGEST_BUFFER                1024
#define  MAX_NV_INDEX_SIZE                2048
#define  MAX_NV_BUFFER_SIZE               1024
#define  MAX_CAP_BUFFER                   1024
#define  NV_MEMORY_SIZE                   16384
#define  NUM_STATIC_PCR                   16
#define  MAX_ALG_LIST_SIZE                64
#define  TIMER_PRESCALE                   100000
#define  PRIMARY_SEED_SIZE                32
#define  CONTEXT_ENCRYPT_ALG              TPM_ALG_AES
#define  CONTEXT_ENCRYPT_KEY_BITS         MAX_SYM_KEY_BITS
#define  CONTEXT_ENCRYPT_KEY_BYTES        ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
#define  CONTEXT_INTEGRITY_HASH_ALG       TPM_ALG_SHA256
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
253   #define  CONTEXT_INTEGRITY_HASH_SIZE      SHA256_DIGEST_SIZE
254   #define  PROOF_SIZE                       CONTEXT_INTEGRITY_HASH_SIZE
255   #define  NV_CLOCK_UPDATE_INTERVAL         12
256   #define  NUM_POLICY_PCR                   1
257   #define  MAX_COMMAND_SIZE                 4096
258   #define  MAX_RESPONSE_SIZE                4096
259   #define  ORDERLY_BITS                     8
260   #define  MAX_ORDERLY_COUNT                ((1<<ORDERLY_BITS)-1)
261   #define  ALG_ID_FIRST                     TPM_ALG_FIRST
262   #define  ALG_ID_LAST                      TPM_ALG_LAST
263   #define  MAX_SYM_DATA                     128
264   #define  MAX_RNG_ENTROPY_SIZE             64
265   #define  RAM_INDEX_SPACE                  512
266   #define  RSA_DEFAULT_PUBLIC_EXPONENT      0x00010001
267   #define  ENABLE_PCR_NO_INCREMENT          YES
268   #define  CRT_FORMAT_RSA                   YES
269   #define  PRIVATE_VENDOR_SPECIFIC_BYTES    \
270       ((MAX_RSA_KEY_BYTES/2)*(3+CRT_FORMAT_RSA*2))
```

From the ISO/IEC 11889-2, Table 8, “Definition of (UINT16) TPM_ALG_ID Constants <IN/OUT, S>”

```
271   typedef  UINT16             TPM_ALG_ID;
272   #define  TPM_ALG_ERROR               (TPM_ALG_ID)(0x0000)
273   #define  ALG_ERROR_VALUE             0x0000
274   #define  TPM_ALG_FIRST               (TPM_ALG_ID)(0x0001)
275   #define  ALG_FIRST_VALUE             0x0001
276   #if defined ALG_RSA && ALG_RSA == YES
277   #define  TPM_ALG_RSA                 (TPM_ALG_ID)(0x0001)
278   #endif
279   #define  ALG_RSA_VALUE               0x0001
280   #define  TPM_ALG_SHA                 (TPM_ALG_ID)(0x0004)
281   #define  ALG_SHA_VALUE               0x0004
282   #if defined ALG_SHA1 && ALG_SHA1 == YES
283   #define  TPM_ALG_SHA1                (TPM_ALG_ID)(0x0004)
284   #endif
285   #define  ALG_SHA1_VALUE              0x0004
286   #if defined ALG_HMAC && ALG_HMAC == YES
287   #define  TPM_ALG_HMAC                (TPM_ALG_ID)(0x0005)
288   #endif
289   #define  ALG_HMAC_VALUE              0x0005
290   #if defined ALG_AES && ALG_AES == YES
291   #define  TPM_ALG_AES                 (TPM_ALG_ID)(0x0006)
292   #endif
293   #define  ALG_AES_VALUE               0x0006
294   #if defined ALG_MGF1 && ALG_MGF1 == YES
295   #define  TPM_ALG_MGF1                (TPM_ALG_ID)(0x0007)
296   #endif
297   #define  ALG_MGF1_VALUE              0x0007
298   #if defined ALG_KEYEDHASH && ALG_KEYEDHASH == YES
299   #define  TPM_ALG_KEYEDHASH           (TPM_ALG_ID)(0x0008)
300   #endif
301   #define  ALG_KEYEDHASH_VALUE         0x0008
302   #if defined ALG_XOR && ALG_XOR == YES
303   #define  TPM_ALG_XOR                 (TPM_ALG_ID)(0x000A)
304   #endif
305   #define  ALG_XOR_VALUE               0x000A
306   #if defined ALG_SHA256 && ALG_SHA256 == YES
307   #define  TPM_ALG_SHA256              (TPM_ALG_ID)(0x000B)
308   #endif
309   #define  ALG_SHA256_VALUE            0x000B
310   #if defined ALG_SHA384 && ALG_SHA384 == YES
311   #define  TPM_ALG_SHA384              (TPM_ALG_ID)(0x000C)
312   #endif
313   #define  ALG_SHA384_VALUE            0x000C
314   #if defined ALG_SHA512 && ALG_SHA512 == YES
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
315   #define  TPM_ALG_SHA512              (TPM_ALG_ID)(0x000D)
316   #endif
317   #define  ALG_SHA512_VALUE            0x000D
318   #define  TPM_ALG_NULL                (TPM_ALG_ID)(0x0010)
319   #define  ALG_NULL_VALUE              0x0010
320   #if defined ALG_SM3_256 && ALG_SM3_256 == YES
321   #define  TPM_ALG_SM3_256             (TPM_ALG_ID)(0x0012)
322   #endif
323   #define  ALG_SM3_256_VALUE           0x0012
324   #if defined ALG_SM4 && ALG_SM4 == YES
325   #define  TPM_ALG_SM4                 (TPM_ALG_ID)(0x0013)
326   #endif
327   #define  ALG_SM4_VALUE               0x0013
328   #if defined ALG_RSASSA && ALG_RSASSA == YES
329   #define  TPM_ALG_RSASSA              (TPM_ALG_ID)(0x0014)
330   #endif
331   #define  ALG_RSASSA_VALUE            0x0014
332   #if defined ALG_RSAES && ALG_RSAES == YES
333   #define  TPM_ALG_RSAES               (TPM_ALG_ID)(0x0015)
334   #endif
335   #define  ALG_RSAES_VALUE             0x0015
336   #if defined ALG_RSAPSS && ALG_RSAPSS == YES
337   #define  TPM_ALG_RSAPSS              (TPM_ALG_ID)(0x0016)
338   #endif
339   #define  ALG_RSAPSS_VALUE            0x0016
340   #if defined ALG_OAEP && ALG_OAEP == YES
341   #define  TPM_ALG_OAEP                (TPM_ALG_ID)(0x0017)
342   #endif
343   #define  ALG_OAEP_VALUE              0x0017
344   #if defined ALG_ECDSA && ALG_ECDSA == YES
345   #define  TPM_ALG_ECDSA               (TPM_ALG_ID)(0x0018)
346   #endif
347   #define  ALG_ECDSA_VALUE             0x0018
348   #if defined ALG_ECDH && ALG_ECDH == YES
349   #define  TPM_ALG_ECDH                (TPM_ALG_ID)(0x0019)
350   #endif
351   #define  ALG_ECDH_VALUE              0x0019
352   #if defined ALG_ECDAA && ALG_ECDAA == YES
353   #define  TPM_ALG_ECDAA               (TPM_ALG_ID)(0x001A)
354   #endif
355   #define  ALG_ECDAA_VALUE             0x001A
356   #if defined ALG_SM2 && ALG_SM2 == YES
357   #define  TPM_ALG_SM2                 (TPM_ALG_ID)(0x001B)
358   #endif
359   #define  ALG_SM2_VALUE               0x001B
360   #if defined ALG_ECSCHNORR && ALG_ECSCHNORR == YES
361   #define  TPM_ALG_ECSCHNORR           (TPM_ALG_ID)(0x001C)
362   #endif
363   #define  ALG_ECSCHNORR_VALUE         0x001C
364   #if defined ALG_ECMQV && ALG_ECMQV == YES
365   #define  TPM_ALG_ECMQV               (TPM_ALG_ID)(0x001D)
366   #endif
367   #define  ALG_ECMQV_VALUE             0x001D
368   #if defined ALG_KDF1_SP800_56a && ALG_KDF1_SP800_56a == YES
369   #define  TPM_ALG_KDF1_SP800_56a      (TPM_ALG_ID)(0x0020)
370   #endif
371   #define  ALG_KDF1_SP800_56a_VALUE    0x0020
372   #if defined ALG_KDF2 && ALG_KDF2 == YES
373   #define  TPM_ALG_KDF2                (TPM_ALG_ID)(0x0021)
374   #endif
375   #define  ALG_KDF2_VALUE              0x0021
376   #if defined ALG_KDF1_SP800_108 && ALG_KDF1_SP800_108 == YES
377   #define  TPM_ALG_KDF1_SP800_108      (TPM_ALG_ID)(0x0022)
378   #endif
379   #define  ALG_KDF1_SP800_108_VALUE    0x0022
380   #if defined ALG_ECC && ALG_ECC == YES
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
#define  TPM_ALG_ECC                (TPM_ALG_ID)(0x0023)
#endif
#define  ALG_ECC_VALUE              0x0023
#if defined ALG_SYMCIPHER && ALG_SYMCIPHER == YES
#define  TPM_ALG_SYMCIPHER          (TPM_ALG_ID)(0x0025)
#endif
#define  ALG_SYMCIPHER_VALUE        0x0025
#if defined ALG_CAMELLIA && ALG_CAMELLIA == YES
#define  TPM_ALG_CAMELLIA           (TPM_ALG_ID)(0x0026)
#endif
#define  ALG_CAMELLIA_VALUE         0x0026
#if defined ALG_CTR && ALG_CTR == YES
#define  TPM_ALG_CTR                (TPM_ALG_ID)(0x0040)
#endif
#define  ALG_CTR_VALUE              0x0040
#if defined ALG_OFB && ALG_OFB == YES
#define  TPM_ALG_OFB                (TPM_ALG_ID)(0x0041)
#endif
#define  ALG_OFB_VALUE              0x0041
#if defined ALG_CBC && ALG_CBC == YES
#define  TPM_ALG_CBC                (TPM_ALG_ID)(0x0042)
#endif
#define  ALG_CBC_VALUE              0x0042
#if defined ALG_CFB && ALG_CFB == YES
#define  TPM_ALG_CFB                (TPM_ALG_ID)(0x0043)
#endif
#define  ALG_CFB_VALUE              0x0043
#if defined ALG_ECB && ALG_ECB == YES
#define  TPM_ALG_ECB                (TPM_ALG_ID)(0x0044)
#endif
#define  ALG_ECB_VALUE              0x0044
#define  TPM_ALG_LAST               (TPM_ALG_ID)(0x0044)
#define  ALG_LAST_VALUE             0x0044
```

From the ISO/IEC 11889-2, Table 9, “Definition of (UINT16) {ECC} TPM_ECC_CURVE Constants <IN/OUT, S>“

```
typedef  UINT16             TPM_ECC_CURVE;
#define  TPM_ECC_NONE         (TPM_ECC_CURVE)(0x0000)
#define  TPM_ECC_NIST_P192    (TPM_ECC_CURVE)(0x0001)
#define  TPM_ECC_NIST_P224    (TPM_ECC_CURVE)(0x0002)
#define  TPM_ECC_NIST_P256    (TPM_ECC_CURVE)(0x0003)
#define  TPM_ECC_NIST_P384    (TPM_ECC_CURVE)(0x0004)
#define  TPM_ECC_NIST_P521    (TPM_ECC_CURVE)(0x0005)
#define  TPM_ECC_BN_P256      (TPM_ECC_CURVE)(0x0010)
#define  TPM_ECC_BN_P638      (TPM_ECC_CURVE)(0x0011)
#define  TPM_ECC_SM2_P256     (TPM_ECC_CURVE)(0x0020)
```

From the ISO/IEC 11889-2, Table 12, “Definition of (UINT32) TPM_CC Constants (Numeric Order) <IN/OUT, S>“

```
typedef  UINT32             TPM_CC;
#define  TPM_CC_FIRST                          (TPM_CC)(0x0000011F)
#define  TPM_CC_PP_FIRST                       (TPM_CC)(0x0000011F)
#if defined CC_NV_UndefineSpaceSpecial && CC_NV_UndefineSpaceSpecial == YES
#define  TPM_CC_NV_UndefineSpaceSpecial        (TPM_CC)(0x0000011F)
#endif
#if defined CC_EvictControl && CC_EvictControl == YES
#define  TPM_CC_EvictControl                   (TPM_CC)(0x00000120)
#endif
#if defined CC_HierarchyControl && CC_HierarchyControl == YES
#define  TPM_CC_HierarchyControl               (TPM_CC)(0x00000121)
#endif
#if defined CC_NV_UndefineSpace && CC_NV_UndefineSpace == YES
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
437   #define  TPM_CC_NV_UndefineSpace              (TPM_CC)(0x00000122)
438   #endif
439   #if defined CC_ChangeEPS && CC_ChangeEPS == YES
440   #define  TPM_CC_ChangeEPS                     (TPM_CC)(0x00000124)
441   #endif
442   #if defined CC_ChangePPS && CC_ChangePPS == YES
443   #define  TPM_CC_ChangePPS                     (TPM_CC)(0x00000125)
444   #endif
445   #if defined CC_Clear && CC_Clear == YES
446   #define  TPM_CC_Clear                         (TPM_CC)(0x00000126)
447   #endif
448   #if defined CC_ClearControl && CC_ClearControl == YES
449   #define  TPM_CC_ClearControl                  (TPM_CC)(0x00000127)
450   #endif
451   #if defined CC_ClockSet && CC_ClockSet == YES
452   #define  TPM_CC_ClockSet                      (TPM_CC)(0x00000128)
453   #endif
454   #if defined CC_HierarchyChangeAuth && CC_HierarchyChangeAuth == YES
455   #define  TPM_CC_HierarchyChangeAuth           (TPM_CC)(0x00000129)
456   #endif
457   #if defined CC_NV_DefineSpace && CC_NV_DefineSpace == YES
458   #define  TPM_CC_NV_DefineSpace                (TPM_CC)(0x0000012A)
459   #endif
460   #if defined CC_PCR_Allocate && CC_PCR_Allocate == YES
461   #define  TPM_CC_PCR_Allocate                  (TPM_CC)(0x0000012B)
462   #endif
463   #if defined CC_PCR_SetAuthPolicy && CC_PCR_SetAuthPolicy == YES
464   #define  TPM_CC_PCR_SetAuthPolicy             (TPM_CC)(0x0000012C)
465   #endif
466   #if defined CC_PP_Commands && CC_PP_Commands == YES
467   #define  TPM_CC_PP_Commands                   (TPM_CC)(0x0000012D)
468   #endif
469   #if defined CC_SetPrimaryPolicy && CC_SetPrimaryPolicy == YES
470   #define  TPM_CC_SetPrimaryPolicy              (TPM_CC)(0x0000012E)
471   #endif
472   #if defined CC_FieldUpgradeStart && CC_FieldUpgradeStart == YES
473   #define  TPM_CC_FieldUpgradeStart             (TPM_CC)(0x0000012F)
474   #endif
475   #if defined CC_ClockRateAdjust && CC_ClockRateAdjust == YES
476   #define  TPM_CC_ClockRateAdjust               (TPM_CC)(0x00000130)
477   #endif
478   #if defined CC_CreatePrimary && CC_CreatePrimary == YES
479   #define  TPM_CC_CreatePrimary                 (TPM_CC)(0x00000131)
480   #endif
481   #if defined CC_NV_GlobalWriteLock && CC_NV_GlobalWriteLock == YES
482   #define  TPM_CC_NV_GlobalWriteLock            (TPM_CC)(0x00000132)
483   #endif
484   #define  TPM_CC_PP_LAST                       (TPM_CC)(0x00000132)
485   #if defined CC_GetCommandAuditDigest && CC_GetCommandAuditDigest == YES
486   #define  TPM_CC_GetCommandAuditDigest         (TPM_CC)(0x00000133)
487   #endif
488   #if defined CC_NV_Increment && CC_NV_Increment == YES
489   #define  TPM_CC_NV_Increment                  (TPM_CC)(0x00000134)
490   #endif
491   #if defined CC_NV_SetBits && CC_NV_SetBits == YES
492   #define  TPM_CC_NV_SetBits                    (TPM_CC)(0x00000135)
493   #endif
494   #if defined CC_NV_Extend && CC_NV_Extend == YES
495   #define  TPM_CC_NV_Extend                     (TPM_CC)(0x00000136)
496   #endif
497   #if defined CC_NV_Write && CC_NV_Write == YES
498   #define  TPM_CC_NV_Write                      (TPM_CC)(0x00000137)
499   #endif
500   #if defined CC_NV_WriteLock && CC_NV_WriteLock == YES
501   #define  TPM_CC_NV_WriteLock                  (TPM_CC)(0x00000138)
502   #endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
503   #if defined CC_DictionaryAttackLockReset && CC_DictionaryAttackLockReset == YES
504   #define  TPM_CC_DictionaryAttackLockReset     (TPM_CC)(0x00000139)
505   #endif
506   #if defined CC_DictionaryAttackParameters && CC_DictionaryAttackParameters == YES
507   #define  TPM_CC_DictionaryAttackParameters    (TPM_CC)(0x0000013A)
508   #endif
509   #if defined CC_NV_ChangeAuth && CC_NV_ChangeAuth == YES
510   #define  TPM_CC_NV_ChangeAuth                 (TPM_CC)(0x0000013B)
511   #endif
512   #if defined CC_PCR_Event && CC_PCR_Event == YES
513   #define  TPM_CC_PCR_Event                     (TPM_CC)(0x0000013C)
514   #endif
515   #if defined CC_PCR_Reset && CC_PCR_Reset == YES
516   #define  TPM_CC_PCR_Reset                     (TPM_CC)(0x0000013D)
517   #endif
518   #if defined CC_SequenceComplete && CC_SequenceComplete == YES
519   #define  TPM_CC_SequenceComplete              (TPM_CC)(0x0000013E)
520   #endif
521   #if defined CC_SetAlgorithmSet && CC_SetAlgorithmSet == YES
522   #define  TPM_CC_SetAlgorithmSet               (TPM_CC)(0x0000013F)
523   #endif
524   #if defined CC_SetCommandCodeAuditStatus && CC_SetCommandCodeAuditStatus == YES
525   #define  TPM_CC_SetCommandCodeAuditStatus     (TPM_CC)(0x00000140)
526   #endif
527   #if defined CC_FieldUpgradeData && CC_FieldUpgradeData == YES
528   #define  TPM_CC_FieldUpgradeData              (TPM_CC)(0x00000141)
529   #endif
530   #if defined CC_IncrementalSelfTest && CC_IncrementalSelfTest == YES
531   #define  TPM_CC_IncrementalSelfTest           (TPM_CC)(0x00000142)
532   #endif
533   #if defined CC_SelfTest && CC_SelfTest == YES
534   #define  TPM_CC_SelfTest                      (TPM_CC)(0x00000143)
535   #endif
536   #if defined CC_Startup && CC_Startup == YES
537   #define  TPM_CC_Startup                       (TPM_CC)(0x00000144)
538   #endif
539   #if defined CC_Shutdown && CC_Shutdown == YES
540   #define  TPM_CC_Shutdown                      (TPM_CC)(0x00000145)
541   #endif
542   #if defined CC_StirRandom && CC_StirRandom == YES
543   #define  TPM_CC_StirRandom                    (TPM_CC)(0x00000146)
544   #endif
545   #if defined CC_ActivateCredential && CC_ActivateCredential == YES
546   #define  TPM_CC_ActivateCredential            (TPM_CC)(0x00000147)
547   #endif
548   #if defined CC_Certify && CC_Certify == YES
549   #define  TPM_CC_Certify                       (TPM_CC)(0x00000148)
550   #endif
551   #if defined CC_PolicyNV && CC_PolicyNV == YES
552   #define  TPM_CC_PolicyNV                      (TPM_CC)(0x00000149)
553   #endif
554   #if defined CC_CertifyCreation && CC_CertifyCreation == YES
555   #define  TPM_CC_CertifyCreation               (TPM_CC)(0x0000014A)
556   #endif
557   #if defined CC_Duplicate && CC_Duplicate == YES
558   #define  TPM_CC_Duplicate                     (TPM_CC)(0x0000014B)
559   #endif
560   #if defined CC_GetTime && CC_GetTime == YES
561   #define  TPM_CC_GetTime                       (TPM_CC)(0x0000014C)
562   #endif
563   #if defined CC_GetSessionAuditDigest && CC_GetSessionAuditDigest == YES
564   #define  TPM_CC_GetSessionAuditDigest         (TPM_CC)(0x0000014D)
565   #endif
566   #if defined CC_NV_Read && CC_NV_Read == YES
567   #define  TPM_CC_NV_Read                       (TPM_CC)(0x0000014E)
568   #endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
#if defined CC_NV_ReadLock && CC_NV_ReadLock == YES
#define  TPM_CC_NV_ReadLock                    (TPM_CC)(0x0000014F)
#endif
#if defined CC_ObjectChangeAuth && CC_ObjectChangeAuth == YES
#define  TPM_CC_ObjectChangeAuth               (TPM_CC)(0x00000150)
#endif
#if defined CC_PolicySecret && CC_PolicySecret == YES
#define  TPM_CC_PolicySecret                   (TPM_CC)(0x00000151)
#endif
#if defined CC_Rewrap && CC_Rewrap == YES
#define  TPM_CC_Rewrap                         (TPM_CC)(0x00000152)
#endif
#if defined CC_Create && CC_Create == YES
#define  TPM_CC_Create                         (TPM_CC)(0x00000153)
#endif
#if defined CC_ECDH_ZGen && CC_ECDH_ZGen == YES
#define  TPM_CC_ECDH_ZGen                      (TPM_CC)(0x00000154)
#endif
#if defined CC_HMAC && CC_HMAC == YES
#define  TPM_CC_HMAC                           (TPM_CC)(0x00000155)
#endif
#if defined CC_Import && CC_Import == YES
#define  TPM_CC_Import                         (TPM_CC)(0x00000156)
#endif
#if defined CC_Load && CC_Load == YES
#define  TPM_CC_Load                           (TPM_CC)(0x00000157)
#endif
#if defined CC_Quote && CC_Quote == YES
#define  TPM_CC_Quote                          (TPM_CC)(0x00000158)
#endif
#if defined CC_RSA_Decrypt && CC_RSA_Decrypt == YES
#define  TPM_CC_RSA_Decrypt                    (TPM_CC)(0x00000159)
#endif
#if defined CC_HMAC_Start && CC_HMAC_Start == YES
#define  TPM_CC_HMAC_Start                     (TPM_CC)(0x0000015B)
#endif
#if defined CC_SequenceUpdate && CC_SequenceUpdate == YES
#define  TPM_CC_SequenceUpdate                 (TPM_CC)(0x0000015C)
#endif
#if defined CC_Sign && CC_Sign == YES
#define  TPM_CC_Sign                           (TPM_CC)(0x0000015D)
#endif
#if defined CC_Unseal && CC_Unseal == YES
#define  TPM_CC_Unseal                         (TPM_CC)(0x0000015E)
#endif
#if defined CC_PolicySigned && CC_PolicySigned == YES
#define  TPM_CC_PolicySigned                   (TPM_CC)(0x00000160)
#endif
#if defined CC_ContextLoad && CC_ContextLoad == YES
#define  TPM_CC_ContextLoad                    (TPM_CC)(0x00000161)
#endif
#if defined CC_ContextSave && CC_ContextSave == YES
#define  TPM_CC_ContextSave                    (TPM_CC)(0x00000162)
#endif
#if defined CC_ECDH_KeyGen && CC_ECDH_KeyGen == YES
#define  TPM_CC_ECDH_KeyGen                    (TPM_CC)(0x00000163)
#endif
#if defined CC_EncryptDecrypt && CC_EncryptDecrypt == YES
#define  TPM_CC_EncryptDecrypt                 (TPM_CC)(0x00000164)
#endif
#if defined CC_FlushContext && CC_FlushContext == YES
#define  TPM_CC_FlushContext                   (TPM_CC)(0x00000165)
#endif
#if defined CC_LoadExternal && CC_LoadExternal == YES
#define  TPM_CC_LoadExternal                   (TPM_CC)(0x00000167)
#endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
635   #if defined CC_MakeCredential && CC_MakeCredential == YES
636   #define  TPM_CC_MakeCredential                (TPM_CC)(0x00000168)
637   #endif
638   #if defined CC_NV_ReadPublic && CC_NV_ReadPublic == YES
639   #define  TPM_CC_NV_ReadPublic                 (TPM_CC)(0x00000169)
640   #endif
641   #if defined CC_PolicyAuthorize && CC_PolicyAuthorize == YES
642   #define  TPM_CC_PolicyAuthorize               (TPM_CC)(0x0000016A)
643   #endif
644   #if defined CC_PolicyAuthValue && CC_PolicyAuthValue == YES
645   #define  TPM_CC_PolicyAuthValue               (TPM_CC)(0x0000016B)
646   #endif
647   #if defined CC_PolicyCommandCode && CC_PolicyCommandCode == YES
648   #define  TPM_CC_PolicyCommandCode             (TPM_CC)(0x0000016C)
649   #endif
650   #if defined CC_PolicyCounterTimer && CC_PolicyCounterTimer == YES
651   #define  TPM_CC_PolicyCounterTimer            (TPM_CC)(0x0000016D)
652   #endif
653   #if defined CC_PolicyCpHash && CC_PolicyCpHash == YES
654   #define  TPM_CC_PolicyCpHash                  (TPM_CC)(0x0000016E)
655   #endif
656   #if defined CC_PolicyLocality && CC_PolicyLocality == YES
657   #define  TPM_CC_PolicyLocality                (TPM_CC)(0x0000016F)
658   #endif
659   #if defined CC_PolicyNameHash && CC_PolicyNameHash == YES
660   #define  TPM_CC_PolicyNameHash                (TPM_CC)(0x00000170)
661   #endif
662   #if defined CC_PolicyOR && CC_PolicyOR == YES
663   #define  TPM_CC_PolicyOR                      (TPM_CC)(0x00000171)
664   #endif
665   #if defined CC_PolicyTicket && CC_PolicyTicket == YES
666   #define  TPM_CC_PolicyTicket                  (TPM_CC)(0x00000172)
667   #endif
668   #if defined CC_ReadPublic && CC_ReadPublic == YES
669   #define  TPM_CC_ReadPublic                    (TPM_CC)(0x00000173)
670   #endif
671   #if defined CC_RSA_Encrypt && CC_RSA_Encrypt == YES
672   #define  TPM_CC_RSA_Encrypt                   (TPM_CC)(0x00000174)
673   #endif
674   #if defined CC_StartAuthSession && CC_StartAuthSession == YES
675   #define  TPM_CC_StartAuthSession              (TPM_CC)(0x00000176)
676   #endif
677   #if defined CC_VerifySignature && CC_VerifySignature == YES
678   #define  TPM_CC_VerifySignature               (TPM_CC)(0x00000177)
679   #endif
680   #if defined CC_ECC_Parameters && CC_ECC_Parameters == YES
681   #define  TPM_CC_ECC_Parameters                (TPM_CC)(0x00000178)
682   #endif
683   #if defined CC_FirmwareRead && CC_FirmwareRead == YES
684   #define  TPM_CC_FirmwareRead                  (TPM_CC)(0x00000179)
685   #endif
686   #if defined CC_GetCapability && CC_GetCapability == YES
687   #define  TPM_CC_GetCapability                 (TPM_CC)(0x0000017A)
688   #endif
689   #if defined CC_GetRandom && CC_GetRandom == YES
690   #define  TPM_CC_GetRandom                     (TPM_CC)(0x0000017B)
691   #endif
692   #if defined CC_GetTestResult && CC_GetTestResult == YES
693   #define  TPM_CC_GetTestResult                 (TPM_CC)(0x0000017C)
694   #endif
695   #if defined CC_Hash && CC_Hash == YES
696   #define  TPM_CC_Hash                          (TPM_CC)(0x0000017D)
697   #endif
698   #if defined CC_PCR_Read && CC_PCR_Read == YES
699   #define  TPM_CC_PCR_Read                      (TPM_CC)(0x0000017E)
700   #endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
701   #if defined CC_PolicyPCR && CC_PolicyPCR == YES
702   #define  TPM_CC_PolicyPCR                    (TPM_CC)(0x0000017F)
703   #endif
704   #if defined CC_PolicyRestart && CC_PolicyRestart == YES
705   #define  TPM_CC_PolicyRestart                (TPM_CC)(0x00000180)
706   #endif
707   #if defined CC_ReadClock && CC_ReadClock == YES
708   #define  TPM_CC_ReadClock                    (TPM_CC)(0x00000181)
709   #endif
710   #if defined CC_PCR_Extend && CC_PCR_Extend == YES
711   #define  TPM_CC_PCR_Extend                   (TPM_CC)(0x00000182)
712   #endif
713   #if defined CC_PCR_SetAuthValue && CC_PCR_SetAuthValue == YES
714   #define  TPM_CC_PCR_SetAuthValue             (TPM_CC)(0x00000183)
715   #endif
716   #if defined CC_NV_Certify && CC_NV_Certify == YES
717   #define  TPM_CC_NV_Certify                   (TPM_CC)(0x00000184)
718   #endif
719   #if defined CC_EventSequenceComplete && CC_EventSequenceComplete == YES
720   #define  TPM_CC_EventSequenceComplete        (TPM_CC)(0x00000185)
721   #endif
722   #if defined CC_HashSequenceStart && CC_HashSequenceStart == YES
723   #define  TPM_CC_HashSequenceStart            (TPM_CC)(0x00000186)
724   #endif
725   #if defined CC_PolicyPhysicalPresence && CC_PolicyPhysicalPresence == YES
726   #define  TPM_CC_PolicyPhysicalPresence       (TPM_CC)(0x00000187)
727   #endif
728   #if defined CC_PolicyDuplicationSelect && CC_PolicyDuplicationSelect == YES
729   #define  TPM_CC_PolicyDuplicationSelect      (TPM_CC)(0x00000188)
730   #endif
731   #if defined CC_PolicyGetDigest && CC_PolicyGetDigest == YES
732   #define  TPM_CC_PolicyGetDigest              (TPM_CC)(0x00000189)
733   #endif
734   #if defined CC_TestParms && CC_TestParms == YES
735   #define  TPM_CC_TestParms                    (TPM_CC)(0x0000018A)
736   #endif
737   #if defined CC_Commit && CC_Commit == YES
738   #define  TPM_CC_Commit                       (TPM_CC)(0x0000018B)
739   #endif
740   #if defined CC_PolicyPassword && CC_PolicyPassword == YES
741   #define  TPM_CC_PolicyPassword               (TPM_CC)(0x0000018C)
742   #endif
743   #if defined CC_ZGen_2Phase && CC_ZGen_2Phase == YES
744   #define  TPM_CC_ZGen_2Phase                  (TPM_CC)(0x0000018D)
745   #endif
746   #if defined CC_EC_Ephemeral && CC_EC_Ephemeral == YES
747   #define  TPM_CC_EC_Ephemeral                 (TPM_CC)(0x0000018E)
748   #endif
749   #if defined CC_PolicyNvWritten && CC_PolicyNvWritten == YES
750   #define  TPM_CC_PolicyNvWritten              (TPM_CC)(0x0000018F)
751   #endif
752   #define  TPM_CC_LAST                         (TPM_CC)(0x0000018F)
753   #ifndef MAX
754   #define MAX(a, b) ((a) > (b) ? (a) : (b))
755   #endif
756   #define MAX_HASH_BLOCK_SIZE  (                  \
757       MAX(ALG_SHA1 * SHA1_BLOCK_SIZE,             \
758       MAX(ALG_SHA256 * SHA256_BLOCK_SIZE,         \
759       MAX(ALG_SHA384 * SHA384_BLOCK_SIZE,         \
760       MAX(ALG_SM3_256 * SM3_256_BLOCK_SIZE,       \
761       MAX(ALG_SHA512 * SHA512_BLOCK_SIZE,         \
762       0 ))))))
763   #define MAX_DIGEST_SIZE      (                  \
764       MAX(ALG_SHA1 * SHA1_DIGEST_SIZE,            \
765       MAX(ALG_SHA256 * SHA256_DIGEST_SIZE,        \
766       MAX(ALG_SHA384 * SHA384_DIGEST_SIZE,        \
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
767         MAX(ALG_SM3_256 * SM3_256_DIGEST_SIZE,       \
768         MAX(ALG_SHA512 * SHA512_DIGEST_SIZE,         \
769         0 ))))))
770     #if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
771     #error "Hash data not valid"
772     #endif
773     #define HASH_COUNT (ALG_SHA1+ALG_SHA256+ALG_SHA384+ALG_SM3_256+ALG_SHA512)
774     #endif  // _IMPLEMENTATION_H
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Annex B
(informative)
Cryptographic Library Interface

B.1 Introduction

The files in Annex B provide cryptographic support functions for the TPM.

When possible, the functions in these files make calls to functions that are provided by a cryptographic library (for Annex B , it is OpenSSL). In many cases, there is a mismatch between the function performed by the cryptographic library and the function needed by the TPM. In those cases, a function is provided in the code in Annex B .

There are cases where the cryptographic library could have been used for a specific function but not all functions of the same group.

EXAMPLE 1 The OpenSSL version of CFB was not suitable for the requirements of the TPM. Rather than have one symmetric mode be provided in this code with the remaining modes provided by OpenSSL, all the symmetric modes are provided in this code.

The provided cryptographic code is believed to be functionally correct but it might not be conformant with all applicable standards. Still, the implementation meets the major objective of the implementation, which is to demonstrate proper TPM behavior. It is not an objective of this implementation to be submitted for certification.

EXAMPLE 2 The RSA key generation schemes produces serviceable RSA keys but the method is not compliant with FIPS 186-3.

B.2 Integer Format

The big integers passed to/from the function interfaces in the crypto engine are in BYTE buffers that have the same format used in ISO/IEC 11889-1 that states:

"An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer."

B.3 CryptoEngine.h

B.3.1. Introduction

This file contains constant definition shared by CryptUtil() and the parts of the Crypto Engine.

```
#ifndef _CRYPT_PRI_H
#define _CRYPT_PRI_H
#include    <stddef.h>
#include    "TpmBuildSwitches.h"
#include    "BaseTypes.h"
#include    "TpmError.h"
#include    "swap.h"
#include    "Implementation.h"
#include    "TPMB.h"
#include    "bool.h"
#include    "Platform.h"
#ifndef NULL
#define NULL    0
#endif
typedef UINT16  NUMBYTES;       // When a size is a number of bytes
typedef UINT32  NUMDIGITS;      // When a size is a number of "digits"
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.3.2. General Purpose Macros

```
#ifndef MAX
#   define MAX(a, b) ((a) > (b) ? (a) : b)
#endif
```

This is the definition of a bit array with one bit per algorithm

```
typedef BYTE    ALGORITHM_VECTOR[(ALG_LAST_VALUE + 7) / 8];
```

B.3.3. Self-test

This structure is used to contain self-test tracking information for the crypto engine. 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;
#ifdef TPM_ALG_RSA
    UINT32      rsa;
#endif
#ifdef  TPM_ALG_ECC
    UINT32      ecc;
#endif
} CRYPTO_SELF_TEST_STATE;
```

B.3.4. Hash-related Structures

```
typedef struct {
    const TPM_ALG_ID      alg;
    const NUMBYTES        digestSize;
    const NUMBYTES        blockSize;
    const NUMBYTES        derSize;
    const BYTE            der[20];
} HASH_INFO;
```

This value will change with each implementation. The value of 16 is used to account for any slop in the context values. 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 ALIGNED_SIZE(x, b) ((((x) + (b) - 1) / (b)) * (b))
#define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
#define MAX_HASH_STATE_SIZE_ALIGNED                                          \
                    ALIGNED_SIZE(MAX_HASH_STATE_SIZE, CRYPTO_ALIGNMENT)
```

This is an byte array that will hold any of the hash contexts.

```
typedef CRYPTO_ALIGNED BYTE ALIGNED_HASH_STATE[MAX_HASH_STATE_SIZE_ALIGNED];
```

Macro to align an address to the next higher size

```
#define AlignPointer(address, align)                                         \
   ((((intptr_t)&(address)) + (align - 1)) & ~(align - 1))
```

Macro to test alignment

```
#define IsAddressAligned(address, align)                                     \
                    (((intptr_t)(address) & (align - 1)) == 0)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

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.

```
48  typedef struct _HASH_STATE
49  {
50      ALIGNED_HASH_STATE       state;
51      TPM_ALG_ID               hashAlg;
52  } CPRI_HASH_STATE, *PCPRI_HASH_STATE;
53  extern const HASH_INFO   g_hashData[HASH_COUNT + 1];
```

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 compile-time check to make sure that this is true.

```
54  typedef struct {
55      ALIGNED_HASH_STATE      buffer;
56      TPM_ALG_ID              hashAlg;
57  } EXPORT_HASH_STATE;
58  typedef enum {
59      IMPORT_STATE,       // Converts externally formatted state to internal
60      EXPORT_STATE        // Converts internal formatted state to external
61  } IMPORT_EXPORT;
```

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 GET_PUT enum is used by _cpri__DrbgGetPutState() to indicate the direction of data flow.

```
62  typedef enum {
63      GET_STATE,      // Get the state to save to NV
64      PUT_STATE       // Restore the state from NV
65  } GET_PUT;
```

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

```
66  #define DRBG_KEY_SIZE_BITS      MAX_AES_KEY_BITS
67  #define DRBG_IV_SIZE_BITS       (MAX_AES_BLOCK_SIZE_BYTES * 8)
68  #define DRBG_ALGORITHM          TPM_ALG_AES
69  #if ((DRBG_KEY_SIZE_BITS % 8) != 0) || ((DRBG_IV_SIZE_BITS % 8) != 0)
70  #error "Key size and IV for DRBG must be even multiples of 8"
71  #endif
72  #if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
73  #error "Key size for DRBG must be even multiple of the cypher block size"
74  #endif
75  typedef UINT32    DRBG_SEED[(DRBG_KEY_SIZE_BITS + DRBG_IV_SIZE_BITS) / 32];
76  typedef struct {
77      UINT64      reseedCounter;
78      UINT32      magic;
79      DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
80      UINT32      lastValue[4];   // used when the TPM does continuous self-test
81                                  // for FIPS compliance of DRBG
82  } DRBG_STATE, *pDRBG_STATE;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

B.3.5. Asymmetric Structures and Values

```
#ifdef TPM_ALG_ECC
```

B.3.6. ECC-related Structures

This structure replicates the structure definition in TPM_Types.h. It is duplicated to avoid inclusion of all of TPM_Types.h This structure is similar to the RSA_KEY structure below. The purpose of these structures is to reduce the overhead of a function call and to make the code less dependent on key types as much as possible.

```
typedef struct {
    UINT32                  curveID;       // The curve identifier
    TPMS_ECC_POINT         *publicPoint;   // Pointer to the public point
    TPM2B_ECC_PARAMETER    *privateKey;    // Pointer to the private key
} ECC_KEY;
#endif // TPM_ALG_ECC
#ifdef TPM_ALG_RSA
```

B.3.7. RSA-related Structures

This structure is a succinct representation of the cryptographic components of an RSA key.

```
typedef struct {
    UINT32        exponent;      // The public exponent pointer
    TPM2B        *publicKey;     // Pointer to the public modulus
    TPM2B        *privateKey;    // The private exponent (not a prime)
} RSA_KEY;
#endif // TPM_ALG_RSA
#ifdef TPM_ALG_RSA
#   ifdef TPM_ALG_ECC
#       if   MAX_RSA_KEY_BYTES > MAX_ECC_KEY_BYTES
#           define  MAX_NUMBER_SIZE         MAX_RSA_KEY_BYTES
#       else
#           define  MAX_NUMBER_SIZE         MAX_ECC_KEY_BYTES
#       endif
#   else // RSA but no ECC
#       define MAX_NUMBER_SIZE             MAX_RSA_KEY_BYTES
#   endif
#elif defined TPM_ALG_ECC
#   define MAX_NUMBER_SIZE                 MAX_ECC_KEY_BYTES
#else
#   error No assymmetric algorithm implemented.
#endif
typedef INT16     CRYPT_RESULT;
#define CRYPT_RESULT_MIN    INT16_MIN
#define CRYPT_RESULT_MAX    INT16_MAX
```

Table B.1

< 0	recoverable error
0	success
> 0	command specific return value (generally a digest size)

```
#define CRYPT_FAIL          ((CRYPT_RESULT)  1)
#define CRYPT_SUCCESS       ((CRYPT_RESULT)  0)
#define CRYPT_NO_RESULT     ((CRYPT_RESULT) -1)
#define CRYPT_SCHEME        ((CRYPT_RESULT) -2)
#define CRYPT_PARAMETER     ((CRYPT_RESULT) -3)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
120   #define CRYPT_UNDERFLOW     ((CRYPT_RESULT) -4)
121   #define CRYPT_POINT         ((CRYPT_RESULT) -5)
122   #define CRYPT_CANCEL        ((CRYPT_RESULT) -6)
123   typedef UINT64              HASH_CONTEXT[MAX_HASH_STATE_SIZE/sizeof(UINT64)];
124   #include    "CpriCryptPri_fp.h"
125   #ifdef  TPM_ALG_ECC
126   #   include "CpriDataEcc.h"
127   #   include "CpriECC_fp.h"
128   #endif
129   #include    "MathFunctions_fp.h"
130   #include    "CpriRNG_fp.h"
131   #include    "CpriHash_fp.h"
132   #include    "CpriSym_fp.h"
133   #ifdef  TPM_ALG_RSA
134   #   include    "CpriRSA_fp.h"
135   #endif
136   #endif // !_CRYPT_PRI_H
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.4 OsslCryptoEngine.h

B.4.1. Introduction

This is the header file used by the components of the CryptoEngine(). This file should not be included in any file other than the files in the crypto engine.

Vendors may replace the implementation in this file by a local crypto engine. The implementation in this file is based on OpenSSL() library. Integer format: the big integers passed in/out the function interfaces in this library by a byte buffer (BYTE *) adopt the same format used in ISO/IEC 11889-1: An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer.

B.4.2. Defines

```
#ifndef _OSSL_CRYPTO_ENGINE_H
#define _OSSL_CRYPTO_ENGINE_H
#include <openssl/aes.h>
#include <openssl/camellia.h>
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <openssl/ec.h>
#include <openssl/rand.h>
#include <openssl/bn.h>
#include <openSSL/ec_lcl.h>
#define     CRYPTO_ENGINE
#include "CryptoEngine.h"
#include "CpriMisc_fp.h"
#define MAX_ECC_PARAMETER_BYTES 32
#define MAX_2B_BYTES MAX((MAX_RSA_KEY_BYTES * ALG_RSA),             \
                         MAX((MAX_ECC_PARAMETER_BYTES * ALG_ECC),  \
                             MAX_DIGEST_SIZE))
#define assert2Bsize(a) pAssert((a).size <= <K>sizeof((a).buffer))
#ifdef TPM_ALG_RSA
#   ifdef   RSA_KEY_SIEVE
#       include     "RsaKeySieve.h"
#       include     "RsaKeySieve_fp.h"
#   endif
#   include    "CpriRSA_fp.h"
#endif
```

This is a structure to hold the parameters for the version of KDFa() used by the CryptoEngine(). This structure allows the state to be passed between multiple functions that use the same pseudo-random sequence.

```
typedef struct {
    CPRI_HASH_STATE          iPadCtx;
    CPRI_HASH_STATE          oPadCtx;
    TPM2B                   *extra;
    UINT32                  *outer;
    TPM_ALG_ID               hashAlg;
    UINT16                   keySizeInBits;
} KDFa_CONTEXT;
#endif // _OSSL_CRYPTO_ENGINE_H
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

B.5 MathFunctions.c

B.5.1. Introduction

This file contains implementation of some of the big number primitives. This is used in order to reduce the overhead in dealing with data conversions to standard big number format.

The simulator code uses the canonical form whenever possible in order to make the code in ISO/IEC 11889-3 more accessible. The canonical data formats are simple and not well suited for complex big number computations. This library provides functions that are found in typical big number libraries but they are written to handle the canonical data format of the reference TPM.

In some cases, data is converted to a big number format used by a standard library, such as OpenSSL(). This is done when the computations are complex enough warrant conversion. Vendors may replace the implementation in this file with a library that provides equivalent functions. A vendor may also rewrite the TPM code so that it uses a standard big number format instead of the canonical form and use the standard libraries instead of the code in this file.

The implementation in this file makes use of the OpenSSL() library.

Integer format: integers passed through the function interfaces in this library adopt the same format used in ISO/IEC 11889-1 that states:

"An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer."

An additional value is needed to indicate the number of significant bytes.

```
1   #include "OsslCryptoEngine.h"
```

B.5.2. Externally Accessible Functions

B.5.2.1. _math__Normalize2B()

This function will normalize the value in a TPM2B. If there are **leading** bytes of zero, the first non-zero byte is shifted up.

Table B.2

Return Value	Meaning
0	no significant bytes, value is zero
>0	number of significant bytes

```
2   LIB_EXPORT UINT16
3   _math__Normalize2B(
4       TPM2B           *b              // IN/OUT: number to normalize
5       )
6   {
7       UINT16      from;
8       UINT16      to;
9       UINT16      size = b->size;
10
11      for(from = 0; b->buffer[from] == 0 && from < size; from++);
12      b->size -= from;
13      for(to = 0; from < size; to++, from++ )
14          b->buffer[to] = b->buffer[from];
15      return b->size;
16  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.5.2.2. _math__Denormalize2B()

This function is used to adjust a TPM2B so that the number has the desired number of bytes. This is accomplished by adding bytes of zero at the start of the number.

Table B.3

Return Value	Meaning
TRUE	number de-normalized
FALSE	number already larger than the desired size

```
17  LIB_EXPORT BOOL
18  _math__Denormalize2B(
19      TPM2B           *in,            // IN:OUT TPM2B number to de-normalize
20      UINT32           size           // IN: the desired size
21      )
22  {
23      UINT32       to;
24      UINT32       from;
25      // If the current size is greater than the requested size, see if this can be
26      // normalized to a value smaller than the requested size and then de-normalize
27      if(in->size > size)
28      {
29          _math__Normalize2B(in);
30          if(in->size > size)
31              return FALSE;
32      }
33      // If the size is already what is requested, leave
34      if(in->size == size)
35          return TRUE;
36
37      // move the bytes to the 'right'
38      for(from = in->size, to = size; from > 0;)
39          in->buffer[--to] = in->buffer[--from];
40
41      // 'to' will always be greater than 0 because we checked for equal above.
42      for(; to > 0;)
43          in->buffer[--to] = 0;
44
45      in->size = (UINT16)size;
46      return TRUE;
47  }
```

B.5.2.3. _math__sub()

This function to subtract one unsigned value from another *c* = *a* - *b*. *c* may be the same as *a* or *b*.

Table B.4

Return Value	Meaning
1	if (a > b) so no borrow
0	if (a = b) so no borrow and b == a
-1	if (a < b) so there was a borrow

```
48  LIB_EXPORT int
49  _math__sub(
50      const UINT32     aSize,         // IN: size of a
51      const BYTE      *a,             // IN: a
52      const UINT32     bSize,         // IN: size of b
53      const BYTE      *b,             // IN: b
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
    UINT16          *cSize,         // OUT: set to MAX(aSize, bSize)
    BYTE            *c              // OUT: the difference
    )
{
    int             borrow = 0;
    int             notZero = 0;
    int             i;
    int             i2;

    // set c to the longer of a or b
    *cSize = (UINT16)((aSize > bSize) ? aSize : bSize);
    // pick the shorter of a and b
    i = (aSize > bSize) ? bSize : aSize;
    i2 = *cSize - i;
    a = &a[aSize - 1];
    b = &b[bSize - 1];
    c = &c[*cSize - 1];
    for(; i > 0; i--)
    {
        borrow = *a-- - *b-- + borrow;
        *c-- = (BYTE)borrow;
        notZero = notZero || borrow;
        borrow >>= 8;
    }
    if(aSize > bSize)
    {
        for(;i2 > 0; i2--)
        {
            borrow = *a-- + borrow;
            *c-- = (BYTE)borrow;
            notZero = notZero || borrow;
            borrow >>= 8;
        }
    }
    else if(aSize < bSize)
    {
        for(;i2 > 0; i2--)
        {
            borrow = 0 - *b-- + borrow;
            *c-- = (BYTE)borrow;
            notZero = notZero || borrow;
            borrow >>= 8;
        }
    }
    // if there is a borrow, then b > a
    if(borrow)
        return -1;
    // either a > b or they are the same
    return notZero;
}
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

B.5.2.4. _math__Inc()

This function increments a large, big-endian number value by one.

Table B.5

Return Value	Meaning
0	result is zero
!0	result is not zero

```
LIB_EXPORT int
_math__Inc(
```

© ISO/IEC 2015 – All rights reserved

```
106         UINT32           aSize,          // IN: size of a
107         BYTE            *a               // IN: a
108         )
109     {
110
111         for(a = &a[aSize-1];aSize > 0; aSize--)
112         {
113             if((*a-- += 1) != 0)
114                 return 1;
115         }
116         return 0;
117     }
```

B.5.2.5. _math__Dec

This function decrements a large, ENDIAN value by one.

```
118     LIB_EXPORT void
119     _math__Dec(
120         UINT32           aSize,          // IN: size of a
121         BYTE            *a               // IN: a
122         )
123     {
124         for(a = &a[aSize-1]; aSize > 0; aSize--)
125         {
126             if((*a-- -= 1) != 0xff)
127                 return;
128         }
129         return;
130     }
```

B.5.2.6. _math__Mul()

This function is used to multiply two large integers: *p* = *a** *b*. If the size of *p* is not specified (*pSize* == NULL), the size of the results *p* is assumed to be *aSize* + *bSize* and the results are de-normalized so that the resulting size is exactly *aSize* + *bSize*. If *pSize* is provided, then the actual size of the result is returned. The initial value for *pSize* must be at least *aSize* + *pSize*.

Table B.6

Return Value	Meaning
< 0	indicates an error
>= 0	the size of the product

```
131     LIB_EXPORT int
132     _math__Mul(
133         const UINT32     aSize,          // IN: size of a
134         const BYTE      *a,              // IN: a
135         const UINT32     bSize,          // IN: size of b
136         const BYTE      *b,              // IN: b
137         UINT32          *pSize,          // IN/OUT: size of the product
138         BYTE            *p               // OUT: product. length of product = aSize +
139                                          //      bSize
140         )
141     {
142         BIGNUM          *bnA;
143         BIGNUM          *bnB;
144         BIGNUM          *bnP;
145         BN_CTX          *context;
146         int              retVal = 0;
147
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
148
149         // First check that pSize is large enough if present
150         if((pSize != NULL) && (*pSize < (aSize + bSize)))
151             return CRYPT_PARAMETER;
152         pAssert(pSize == NULL || *pSize <= MAX_2B_BYTES);
153         //
154         // Allocate space for BIGNUM context
155         //
156         context = BN_CTX_new();
157         if(context == NULL)
158             FAIL(FATAL_ERROR_ALLOCATION);
159         bnA = BN_CTX_get(context);
160         bnB = BN_CTX_get(context);
161         bnP = BN_CTX_get(context);
162         if (bnP == NULL)
163             FAIL(FATAL_ERROR_ALLOCATION);
164
165         // Convert the inputs to BIGNUMs
166         //
167         if (BN_bin2bn(a, aSize, bnA) == NULL || BN_bin2bn(b, bSize, bnB) == NULL)
168             FAIL(FATAL_ERROR_INTERNAL);
169
170         // Perform the multiplication
171         //
172         if (BN_mul(bnP, bnA, bnB, context) != 1)
173             FAIL(FATAL_ERROR_INTERNAL);
174
175
176         // If the size of the results is allowed to float, then set the return
177         // size. Otherwise, it might be necessary to de-normalize the results
178         retVal = BN_num_bytes(bnP);
179         if(pSize == NULL)
180         {
181             BN_bn2bin(bnP, &p[aSize + bSize - retVal]);
182             memset(p, 0, aSize + bSize - retVal);
183             retVal = aSize + bSize;
184         }
185         else
186         {
187             BN_bn2bin(bnP, p);
188             *pSize = retVal;
189         }
190
191         BN_CTX_end(context);
192         BN_CTX_free(context);
193         return retVal;
194     }
```

B.5.2.7. _math__Div()

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 B.7

Return Value	Meaning
CRYPT_SUCCESS	operation complete
CRYPT_UNDERFLOW	*q* or *r* is too small to receive the result

```
195     LIB_EXPORT CRYPT_RESULT
196     _math__Div(
197         const TPM2B     *n,             // IN: numerator
198         const TPM2B     *d,             // IN: denominator
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
199         TPM2B           *q,             // OUT: quotient
200         TPM2B           *r              // OUT: remainder
201         )
202     {
203         BIGNUM          *bnN;
204         BIGNUM          *bnD;
205         BIGNUM          *bnQ;
206         BIGNUM          *bnR;
207         BN_CTX          *context;
208         CRYPT_RESULT     retVal = CRYPT_SUCCESS;
209
210         // Get structures for the big number representations
211         context = BN_CTX_new();
212         if(context == NULL)
213             FAIL(FATAL_ERROR_ALLOCATION);
214         BN_CTX_start(context);
215         bnN = BN_CTX_get(context);
216         bnD = BN_CTX_get(context);
217         bnQ = BN_CTX_get(context);
218         bnR = BN_CTX_get(context);
219
220         // Errors in BN_CTX_get() are sticky so only need to check the last allocation
221         if (   bnR == NULL
222             || BN_bin2bn(n->buffer, n->size, bnN) == NULL
223             || BN_bin2bn(d->buffer, d->size, bnD) == NULL)
224                 FAIL(FATAL_ERROR_INTERNAL);
225
226         // Check for divide by zero.
227         if(BN_num_bits(bnD) == 0)
228             FAIL(FATAL_ERROR_DIVIDE_ZERO);
229
230         // Perform the division
231         if (BN_div(bnQ, bnR, bnN, bnD, context) != 1)
232             FAIL(FATAL_ERROR_INTERNAL);
233
234
235         // Convert the BIGNUM result back to our format
236         if(q != NULL)   // If the quotient is being returned
237         {
238             if(!BnTo2B(q, bnQ, q->size))
239             {
240                 retVal = CRYPT_UNDERFLOW;
241                 goto Done;
242             }
243          }
244         if(r != NULL)   // If the remainder is being returned
245         {
246             if(!BnTo2B(r, bnR, r->size))
247                 retVal = CRYPT_UNDERFLOW;
248         }
249
250     Done:
251         BN_CTX_end(context);
252         BN_CTX_free(context);
253
254         return retVal;
255     }
```

B.5.2.8. _math__uComp()

This function compare two unsigned values.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

Table B.8

Return Value	Meaning
1	if (a > b)
0	if (a = b)
-1	if (a < b)

```
256  LIB_EXPORT int
257  _math__uComp(
258      const UINT32     aSize,          // IN: size of a
259      const BYTE      *a,              // IN: a
260      const UINT32     bSize,          // IN: size of b
261      const BYTE      *b               // IN: b
262      )
263  {
264      int             borrow = 0;
265      int             notZero = 0;
266      int             i;
267      // If a has more digits than b, then a is greater than b if
268      // any of the more significant bytes is non zero
269      if((i = (int)aSize - (int)bSize) > 0)
270          for(; i > 0; i--)
271              if(*a++) // means a > b
272                  return 1;
273      // If b has more digits than a, then b is greater if any of the
274      // more significant bytes is non zero
275      if(i < 0)  <Q>// Means that b is longer than a
276          for(; i < 0; i++)
277              if(*b++) // means that b > a
278                  return -1;
279      // Either the vales are the same size or the upper bytes of a or b are
280      // all zero, so compare the rest
281      i = (aSize > bSize) ? bSize : aSize;
282      a = &a[i-1];
283      b = &b[i-1];
284      for(; i > 0; i--)
285      {
286          borrow = *a-- - *b-- + borrow;
287          notZero = notZero || borrow;
288          borrow >>= 8;
289      }
290      // if there is a borrow, then b > a
291      if(borrow)
292          return -1;
293      // either a > b or they are the same
294      return notZero;
295  }
```

B.5.2.9. _math__Comp()

Compare two signed integers:

Table B.9

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
296  LIB_EXPORT int
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
297 _math__Comp(
298     const UINT32     aSize,          // IN: size of a
299     const BYTE      *a,              // IN: a buffer
300     const UINT32     bSize,          // IN: size of b
301     const BYTE      *b               // IN: b buffer
302     )
303 {
304     int      signA, signB;       // sign of a and b
305 
306     // For positive or 0, sign_a is 1
307     // for negative, sign_a is 0
308     signA = ((a[0] & 0x80) == 0) ? 1 : 0;
309 
310     // For positive or 0, sign_b is 1
311     // for negative, sign_b is 0
312     signB = ((b[0] & 0x80) == 0) ? 1 : 0;
313 
314     if(signA != signB)
315     {
316         return signA - signB;
317     }
318 
319     if(signA == 1)
320         // do unsigned compare function
321         return _math__uComp(aSize, a, bSize, b);
322     else
323         // do unsigned compare the other way
324         return 0 - _math__uComp(aSize, a, bSize, b);
325 }
```

B.5.2.10. _math__ModExp

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 B.10

Return Value	Meaning
CRYPT_SUCCESS	exponentiation succeeded
CRYPT_PARAMETER	number to exponentiate is larger than the modulus
CRYPT_UNDERFLOW	result will not fit into the provided buffer

```
326 LIB_EXPORT CRYPT_RESULT
327 _math__ModExp(
328     UINT32           cSize,          // IN: size of the results
329     BYTE            *c,              // OUT: results buffer
330     const UINT32     mSize,          // IN: size of number to be exponentiated
331     const BYTE      *m,              // IN: number to be exponentiated
332     const UINT32     eSize,          // IN: size of power
333     const BYTE      *e,              // IN: power
334     const UINT32     nSize,          // IN: modulus size
335     const BYTE      *n               // IN: modulus
336     )
337 {
338     CRYPT_RESULT     retVal = CRYPT_SUCCESS;
339     BN_CTX          *context;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
340         BIGNUM          *bnC;
341         BIGNUM          *bnM;
342         BIGNUM          *bnE;
343         BIGNUM          *bnN;
344         INT32            i;
345
346         context = BN_CTX_new();
347         if(context == NULL)
348             FAIL(FATAL_ERROR_ALLOCATION);
349         BN_CTX_start(context);
350         bnC = BN_CTX_get(context);
351         bnM = BN_CTX_get(context);
352         bnE = BN_CTX_get(context);
353         bnN = BN_CTX_get(context);
354
355         // Errors for BN_CTX_get are sticky so only need to check last allocation
356         if(bnN == NULL)
357             FAIL(FATAL_ERROR_ALLOCATION);
358
359         //convert arguments
360         if (   BN_bin2bn(m, mSize, bnM) == NULL
361             || BN_bin2bn(e, eSize, bnE) == NULL
362             || BN_bin2bn(n, nSize, bnN) == NULL)
363                 FAIL(FATAL_ERROR_INTERNAL);
364
365         // Don't do exponentiation if the number being exponentiated is
366         // larger than the modulus.
367         if(BN_ucmp(bnM, bnN) >= 0)
368         {
369             retVal = CRYPT_PARAMETER;
370             goto Cleanup;
371         }
372         // Perform the exponentiation
373         if(!(BN_mod_exp(bnC, bnM, bnE, bnN, context)))
374             FAIL(FATAL_ERROR_INTERNAL);
375
376         // Convert the results
377         // Make sure that the results will fit in the provided buffer.
378         if((unsigned)BN_num_bytes(bnC) > cSize)
379         {
380             retVal = CRYPT_UNDERFLOW;
381             goto Cleanup;
382         }
383         i = cSize - BN_num_bytes(bnC);
384         BN_bn2bin(bnC, &c[i]);
385         memset(c, 0, i);
386
387     Cleanup:
388         // Free up allocated BN values
389         BN_CTX_end(context);
390         BN_CTX_free(context);
391         return retVal;
392     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

B.5.2.11. _math__IsPrime()

Check if an 32-bit integer is a prime.

© ISO/IEC 2015 – All rights reserved

Table B.11

Return Value	Meaning
TRUE	if the integer is probably a prime
FALSE	if the integer is definitely not a prime

```
393  LIB_EXPORT BOOL
394  _math__IsPrime(
395      const UINT32     prime
396      )
397  {
398      int     isPrime;
399      BIGNUM  *p;
400
401      // Assume the size variables are not overflow, which should not happen in
402      // the contexts that this function will be called.
403      if((p = BN_new()) == NULL)
404          FAIL(FATAL_ERROR_ALLOCATION);
405      if(!BN_set_word(p, prime))
406          FAIL(FATAL_ERROR_INTERNAL);
407
408      //
409      // BN_is_prime returning -1 means that it ran into an error.
410      // It should only return 0 or 1
411      //
412      if((isPrime = BN_is_prime_ex(p, BN_prime_checks, NULL, NULL)) < 0)
413          FAIL(FATAL_ERROR_INTERNAL);
414
415      if(p != NULL)
416          BN_clear_free(p);
417      return (isPrime == 1);
418  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.6 CpriCryptPri.c

B.6.1. Introduction

This file contains the interface to the initialization, startup and shutdown functions of the crypto library.

B.6.2. Includes and Locals

```
#include "OsslCryptoEngine.h"
static void Trap(const char *function, int line, int code);
FAIL_FUNCTION       TpmFailFunction = (FAIL_FUNCTION)&Trap;
```

B.6.3. Functions

B.6.3.1. TpmFail()

This is a shim function that is called when a failure occurs. It simply relays the call to the callback pointed to by TpmFailFunction(). It is only defined for the sake of NO_RETURN specifier that cannot be added to a function pointer with some compilers.

```
void
TpmFail(
    const char          *function,
    int                  line,
    int                  code)
{
    TpmFailFunction(function, line, code);
}
```

B.6.3.2. FAILURE_TRAP()

This function is called if the caller to _cpri__InitCryptoUnits() doesn't provide a call back address.

```
static void
Trap(
    const char      *function,
    int              line,
    int              code
    )
{
    UNREFERENCED(function);
    UNREFERENCED(line);
    UNREFERENCED(code);
    abort();
}
```

B.6.3.3. _cpri__InitCryptoUnits()

This function calls the initialization functions of the other crypto modules that are part of the crypto engine for this implementation. This function should be called as a result of _TPM_Init(). The parameter to this function is a call back function it TMP. lib that is called when the crypto engine has a failure.

```
LIB_EXPORT CRYPT_RESULT
_cpri__InitCryptoUnits(
    FAIL_FUNCTION    failFunction
    )
{
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
29      TpmFailFunction = failFunction;
30
31      _cpri__RngStartup();
32      _cpri__HashStartup();
33      _cpri__SymStartup();
34
35  #ifdef TPM_ALG_RSA
36      _cpri__RsaStartup();
37  #endif
38
39  #ifdef TPM_ALG_ECC
40      _cpri__EccStartup();
41  #endif
42
43      return CRYPT_SUCCESS;
44  }
```

B.6.3.4. _cpri__StopCryptoUnits()

This function calls the shutdown functions of the other crypto modules that are part of the crypto engine for this implementation.

```
45  LIB_EXPORT void
46  _cpri__StopCryptoUnits(
47      void
48      )
49  {
50      return;
51  }
```

B.6.3.5. _cpri__Startup()

This function calls the startup functions of the other crypto modules that are part of the crypto engine for this implementation. This function should be called during processing of TPM2_Startup().

```
52  LIB_EXPORT BOOL
53  _cpri__Startup(
54      void
55      )
56  {
57
58      return(   _cpri__HashStartup()
59             && _cpri__RngStartup()
60  #ifdef TPM_ALG_RSA
61             && _cpri__RsaStartup()
62  #endif // TPM_ALG_RSA
63  #ifdef TPM_ALG_ECC
64             && _cpri__EccStartup()
65  #endif // TPM_ALG_ECC
66             && _cpri__SymStartup());
67  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.7 CpriRNG.c

B.7.1. Introduction

This file contains the interface to the OpenSSL() random number functions.

B.7.2. Defines

```
1   //#define __TPM_RNG_FOR_DEBUG__
```

B.7.3. Includes and Values

```
2   #include "OsslCryptoEngine.h"
3   int         s_entropyFailure;
```

B.7.4. Functions

B.7.4.1. _cpri__RngStartup()

This function is called to initialize the random number generator. It collects entropy from the platform to seed the OpenSSL() random number generator.

```
4   LIB_EXPORT BOOL
5   _cpri__RngStartup(void)
6   {
7       UINT32       entropySize;
8       BYTE         entropy[MAX_RNG_ENTROPY_SIZE];
9       INT32        returnedSize = 0;
10
11      // Initialize the entropy source
12      s_entropyFailure = FALSE;
13      _plat__GetEntropy(NULL, 0);
14
15      // Collect entropy until we have enough
16      for(entropySize  = 0;
17          entropySize < MAX_RNG_ENTROPY_SIZE && returnedSize >= 0;
18          entropySize += returnedSize)
19      {
20          returnedSize = _plat__GetEntropy(&entropy[entropySize],
21                                           MAX_RNG_ENTROPY_SIZE - entropySize);
22      }
23      // Got some entropy on the last call and did not get an error
24      if(returnedSize > 0)
25      {
26          // Seed OpenSSL with entropy
27          RAND_seed(entropy, entropySize);
28      }
29      else
30      {
31          s_entropyFailure = TRUE;
32      }
33      return s_entropyFailure == FALSE;
34  }
```

B.7.4.2. _cpri__DrbgGetPutState()

This function is used to set the state of the RNG *(direction* == PUT_STATE) or to recover the state of the RNG *(direction* == GET_STATE).

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

NOTE This not currently supported on OpenSSL() version.

```
35  LIB_EXPORT CRYPT_RESULT
36  _cpri__DrbgGetPutState(
37      GET_PUT          direction,
38      int              bufferSize,
39      BYTE            *buffer
40      )
41  {
42      UNREFERENCED_PARAMETER(direction);
43      UNREFERENCED_PARAMETER(bufferSize);
44      UNREFERENCED_PARAMETER(buffer);
45
46      return CRYPT_SUCCESS;       // Function is not implemented
47  }
```

B.7.4.3. _cpri__StirRandom()

This function is called to add external entropy to the OpenSSL() random number generator.

```
48  LIB_EXPORT CRYPT_RESULT
49  _cpri__StirRandom(
50      INT32            entropySize,
51      BYTE            *entropy
52      )
53  {
54      if (entropySize >= 0)
55      {
56          RAND_add((const void *)entropy, (int) entropySize, 0.0);
57
58      }
59      return CRYPT_SUCCESS;
60  }
```

B.7.4.4. _cpri__GenerateRandom()

This function is called to get a string of random bytes from the OpenSSL() random number generator. The return value is the number of bytes placed in the buffer. If the number of bytes returned is not equal to the number of bytes requested (*randomSize*) it is indicative of a failure of the OpenSSL() random number generator and is probably fatal.

```
61  LIB_EXPORT UINT16
62  _cpri__GenerateRandom(
63      INT32            randomSize,
64      BYTE            *buffer
65      )
66  {
67      //
68      // We don't do negative sizes or ones that are too large
69      if (randomSize < 0 || randomSize > UINT16_MAX)
70          return 0;
71      // RAND_bytes uses 1 for success and we use 0
72      if(RAND_bytes(buffer, randomSize) == 1)
73          return (UINT16)randomSize;
74      else
75          return 0;
76  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

B.7.4.5. _cpri__GenerateSeededRandom()

This function is used to generate a pseudo-random number from some seed values. This function returns the same result each time it is called with the same parameters

```
77  LIB_EXPORT UINT16
78  _cpri__GenerateSeededRandom(
79      INT32            randomSize,     // IN: the size of the request
80      BYTE            *random,         // OUT: receives the data
81      TPM_ALG_ID       hashAlg,        // IN: used by KDF version but not here
82      TPM2B           *seed,           // IN: the seed value
83      const char      *label,          // IN: a label string (optional)
84      TPM2B           *partyU,         // IN: other data (oprtional)
85      TPM2B           *partyV          // IN: still more (optional)
86      )
87  {
88
89      return (_cpri__KDFa(hashAlg, seed, label, partyU, partyV,
90                          randomSize * 8, random, NULL, FALSE));
91  }
92  #endif  //%
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.8 CpriHash.c

B.8.1. Description

This file contains implementation of cryptographic functions for hashing.

B.8.2. Includes, Defines, and Types

```
#include    "OsslCryptoEngine.h"
#include    "CpriHashData.c"
#define OSSL_HASH_STATE_DATA_SIZE    (MAX_HASH_STATE_SIZE - 8)
typedef struct {
    union   {
        EVP_MD_CTX  context;
        BYTE        data[OSSL_HASH_STATE_DATA_SIZE];
    } u;
    INT16           copySize;
} OSSL_HASH_STATE;
```

Temporary aliasing of SM3 to SHA256 until SM3 is available

```
#define EVP_sm3_256 EVP_sha256
```

B.8.3. Static Functions

B.8.3.1. GetHashServer()

This function returns the address of the hash server function

```
static EVP_MD *
GetHashServer(
    TPM_ALG_ID   hashAlg
)
{
    switch (hashAlg)
    {
#ifdef TPM_ALG_SHA1
    case TPM_ALG_SHA1:
        return (EVP_MD *)EVP_sha1();
        break;
#endif
#ifdef TPM_ALG_SHA256
    case TPM_ALG_SHA256:
        return (EVP_MD *)EVP_sha256();
        break;
#endif
#ifdef TPM_ALG_SHA384
    case TPM_ALG_SHA384:
        return (EVP_MD *)EVP_sha384();
        break;
#endif
#ifdef TPM_ALG_SHA512
    case TPM_ALG_SHA512:
        return (EVP_MD *)EVP_sha512();
        break;
#endif
#ifdef TPM_ALG_SM3_256
    case TPM_ALG_SM3_256:
        return (EVP_MD *)EVP_sm3_256();
        break;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
43  #endif
44      case TPM_ALG_NULL:
45          return NULL;
46      default:
47          FAIL(FATAL_ERROR_INTERNAL);
48      }
49  }
```

B.8.3.2. MarshalHashState()

This function copies an OpenSSL() hash context into a caller provided buffer.

Table B.12

Return Value	Meaning
> 0	the number of bytes of buf used.

```
50  static UINT16
51  MarshalHashState(
52      EVP_MD_CTX      *ctxt,          // IN: Context to marshal
53      BYTE            *buf            // OUT: The buffer that will receive the
54                                      //      context. This buffer is at least
55                                      //      MAX_HASH_STATE_SIZE bytes
56      )
57  {
58      // make sure everything will fit
59      pAssert(ctxt->digest->ctx_size <= OSSL_HASH_STATE_DATA_SIZE);
60
61      // Copy the context data
62      memcpy(buf, (void*) ctxt->md_data, ctxt->digest->ctx_size);
63
64      return (UINT16)ctxt->digest->ctx_size;
65  }
```

B.8.3.3. GetHashState()

This function will unmarshal a caller provided buffer into an OpenSSL() hash context. The function returns the number of bytes copied (which may be zero).

```
66  static UINT16
67  GetHashState(
68      EVP_MD_CTX      *ctxt,          // OUT: The context structure to receive the
69                                      //      result of unmarshaling.
70      TPM_ALG_ID       algType,       // IN: The hash algorithm selector
71      BYTE            *buf            // IN: Buffer containing marshaled hash data
72      )
73  {
74      EVP_MD          *evpmdAlgorithm = NULL;
75
76      pAssert(ctxt != NULL);
77
78      EVP_MD_CTX_init(ctxt);
79
80      evpmdAlgorithm = GetHashServer(algType);
81      if(evpmdAlgorithm == NULL)
82          return 0;
83
84      // This also allocates the ctxt->md_data
85      if((EVP_DigestInit_ex(ctxt, evpmdAlgorithm, NULL)) != 1)
86          FAIL(FATAL_ERROR_INTERNAL);
87
88      pAssert(ctxt->digest->ctx_size < <K>sizeof(ALIGNED_HASH_STATE));
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
89      memcpy(ctxt->md_data, buf, ctxt->digest->ctx_size);
90      return (UINT16)ctxt->digest->ctx_size;
91  }
```

B.8.3.4. GetHashInfoPointer()

This function returns a pointer to the hash info for the algorithm. If the algorithm is not supported, function returns a pointer to the data block associated with TPM_ALG_NULL.

```
92  static const HASH_INFO *
93  GetHashInfoPointer(
94      TPM_ALG_ID       hashAlg
95      )
96  {
97      UINT32 i, tableSize;
98
99      // Get the table size of g_hashData
100     tableSize = sizeof(g_hashData) / sizeof(g_hashData[0]);
101
102     for(i = 0; i < tableSize - 1; i++)
103     {
104         if(g_hashData[i].alg == hashAlg)
105             return &g_hashData[i];
106     }
107     return &g_hashData[tableSize-1];
108 }
```

B.8.4. Hash Functions

B.8.4.1. _cpri__HashStartup()

Function that is called to initialize the hash service. In this implementation, this function does nothing but it is called by the CryptUtilStartup() function and must be present.

```
109 LIB_EXPORT BOOL
110 _cpri__HashStartup(
111     void
112     )
113 {
114     // On startup, make sure that the structure sizes are compatible. It would
115     // be nice if this could be done at compile time but I couldn't figure it out.
116     CPRI_HASH_STATE *cpriState = NULL;
117 //    NUMBYTES         evpCtxSize = sizeof(EVP_MD_CTX);
118     NUMBYTES         cpriStateSize = sizeof(cpriState->state);
119 //    OSSL_HASH_STATE *osslState;
120     NUMBYTES         osslStateSize = sizeof(OSSL_HASH_STATE);
121 //    int              dataSize = sizeof(osslState->u.data);
122     pAssert(cpriStateSize >= osslStateSize);
123
124     return TRUE;
125 }
```

B.8.4.2. _cpri__GetHashAlgByIndex()

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 and *index* of 2 will return the last. All other index values will return TPM_ALG_NULL.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

Table B.13

Return Value	Meaning
TPM_ALG_xxx()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
126 LIB_EXPORT TPM_ALG_ID
127 _cpri__GetHashAlgByIndex(
128     UINT32          index          // IN: the index
129     )
130 {
131     if(index >= HASH_COUNT)
132         return TPM_ALG_NULL;
133     return g_hashData[index].alg;
134 }
```

B.8.4.3. _cpri__GetHashBlockSize()

Returns the size of the block used for the hash.

Table B.14

Return Value	Meaning
< 0	the algorithm is not a supported hash
>=	the digest size (0 for TPM_ALG_NULL)

```
135 LIB_EXPORT UINT16
136 _cpri__GetHashBlockSize(
137     TPM_ALG_ID      hashAlg        // IN: hash algorithm to look up
138     )
139 {
140     return GetHashInfoPointer(hashAlg)->blockSize;
141 }
```

B.8.4.4. _cpri__GetHashDER

This function returns a pointer to the DER string for the algorithm and indicates its size.

```
142 LIB_EXPORT UINT16
143 _cpri__GetHashDER(
144     TPM_ALG_ID      hashAlg,       // IN: the algorithm to look up
145     const BYTE      **p
146     )
147 {
148     const HASH_INFO       *q;
149     q = GetHashInfoPointer(hashAlg);
150     *p = &q->der[0];
151     return q->derSize;
152 }
```

B.8.4.5. _cpri__GetDigestSize()

Gets the digest size of the algorithm. The algorithm is required to be supported.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table B.15

Return Value	Meaning
=0	the digest size for TPM_ALG_NULL
>0	the digest size of a hash algorithm

```
153  LIB_EXPORT UINT16
154  _cpri__GetDigestSize(
155      TPM_ALG_ID       hashAlg        // IN: hash algorithm to look up
156      )
157  {
158      return GetHashInfoPointer(hashAlg)->digestSize;
159  }
```

B.8.4.6. _cpri__GetContextAlg()

This function returns the algorithm associated with a hash context

```
160  LIB_EXPORT TPM_ALG_ID
161  _cpri__GetContextAlg(
162      CPRI_HASH_STATE     *hashState      // IN: the hash context
163      )
164  {
165      return hashState->hashAlg;
166  }
```

B.8.4.7. _cpri__CopyHashState

This function is used to **clone** a CPRI_HASH_STATE. The return value is the size of the state.

```
167  LIB_EXPORT UINT16
168  _cpri__CopyHashState (
169      CPRI_HASH_STATE     *out,           // OUT: destination of the state
170      CPRI_HASH_STATE     *in             // IN: source of the state
171      )
172  {
173      OSSL_HASH_STATE    *i = (OSSL_HASH_STATE *)&in->state;
174      OSSL_HASH_STATE    *o = (OSSL_HASH_STATE *)&out->state;
175      pAssert(sizeof(i) <= <K>sizeof(in->state));
176
177      EVP_MD_CTX_init(&o->u.context);
178      EVP_MD_CTX_copy_ex(&o->u.context, &i->u.context);
179      o->copySize = i->copySize;
180      out->hashAlg = in->hashAlg;
181      return sizeof(CPRI_HASH_STATE);
182  }
```

B.8.4.8. _cpri__StartHash()

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.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table B.16

Return Value	Meaning
0	hash is TPM_ALG_NULL
>0	digest size

```
183  LIB_EXPORT UINT16
184  _cpri__StartHash(
185      TPM_ALG_ID           hashAlg,       // IN: hash algorithm
186      BOOL                 sequence,      // IN: TRUE if the state should be saved
187      CPRI_HASH_STATE     *hashState      // OUT: the state of hash stack.
188      )
189  {
190      EVP_MD_CTX       localState;
191      OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
192      BYTE            *stateData = state->u.data;
193      EVP_MD_CTX      *context;
194      EVP_MD          *evpmdAlgorithm = NULL;
195      UINT16           retVal = 0;
196
197      if(sequence)
198          context = &localState;
199      else
200          context = &state->u.context;
201
202      hashState->hashAlg = hashAlg;
203
204      EVP_MD_CTX_init(context);
205      evpmdAlgorithm = GetHashServer(hashAlg);
206      if(evpmdAlgorithm == NULL)
207          goto Cleanup;
208
209      if(EVP_DigestInit_ex(context, evpmdAlgorithm, NULL) != 1)
210          FAIL(FATAL_ERROR_INTERNAL);
211      retVal = (CRYPT_RESULT)EVP_MD_CTX_size(context);
212
213  Cleanup:
214      if(retVal > 0)
215      {
216          if (sequence)
217          {
218              if((state->copySize = MarshalHashState(context, stateData)) == 0)
219              {
220                  // If MarshalHashState returns a negative number, it is an error
221                  // code and not a hash size so copy the error code to be the return
222                  // from this function and set the actual stateSize to zero.
223                  retVal = state->copySize;
224                  state->copySize = 0;
225              }
226              // Do the cleanup
227              EVP_MD_CTX_cleanup(context);
228          }
229          else
230              state->copySize = -1;
231      }
232      else
233          state->copySize = 0;
234      return retVal;
235  }
```

B.8.4.9. _cpri__UpdateHash()

Add data to a hash or HMAC stack.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
236 LIB_EXPORT void
237 _cpri__UpdateHash(
238     CPRI_HASH_STATE     *hashState,     // IN: the hash context information
239     UINT32               dataSize,      // IN: the size of data to be added to the
240                                         //     digest
241     BYTE                *data           // IN: data to be hashed
242     )
243 {
244     EVP_MD_CTX       localContext;
245     OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
246     BYTE            *stateData = state->u.data;
247     EVP_MD_CTX      *context;
248     CRYPT_RESULT     retVal = CRYPT_SUCCESS;
249
250     // If there is no context, return
251     if(state->copySize == 0)
252         return;
253     if(state->copySize > 0)
254     {
255         context = &localContext;
256         if((retVal = GetHashState(context, hashState->hashAlg, stateData)) <= 0)
257             return;
258     }
259     else
260         context = &state->u.context;
261
262     if(EVP_DigestUpdate(context, data, dataSize) != 1)
263         FAIL(FATAL_ERROR_INTERNAL);
264     else if(   state->copySize > 0
265                && (retVal= MarshalHashState(context, stateData)) >= 0)
266     {
267         // retVal is the size of the marshaled data. Make sure that it is consistent
268         // by ensuring that we didn't get more than allowed
269         if(retVal < state->copySize)
270             FAIL(FATAL_ERROR_INTERNAL);
271         else
272             EVP_MD_CTX_cleanup(context);
273     }
274     return;
275 }
```

B.8.4.10. _cpri__CompleteHash()

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 B.17

Return Value	Meaning
0	no data returned
> 0	the number of bytes in the digest

```
276 LIB_EXPORT UINT16
277 _cpri__CompleteHash(
278     CPRI_HASH_STATE     *hashState,     // IN: the state of hash stack
279     UINT32               dOutSize,      // IN: size of digest buffer
280     BYTE                *dOut           // OUT: hash digest
281     )
282 {
283     EVP_MD_CTX       localState;
284     OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
285     BYTE            *stateData = state->u.data;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
286         EVP_MD_CTX      *context;
287         UINT16           retVal;
288         int              hLen;
289         BYTE             temp[MAX_DIGEST_SIZE];
290         BYTE            *rBuffer = dOut;
291
292         if(state->copySize == 0)
293             return 0;
294         if(state->copySize > 0)
295         {
296             context = &localState;
297             if((retVal = GetHashState(context, hashState->hashAlg, stateData)) <= 0)
298                 goto Cleanup;
299         }
300         else
301             context = &state->u.context;
302
303         hLen = EVP_MD_CTX_size(context);
304         if((unsigned)hLen > dOutSize)
305             rBuffer = temp;
306         if(EVP_DigestFinal_ex(context, rBuffer, NULL) == 1)
307         {
308             if(rBuffer != dOut)
309             {
310                 if(dOut != NULL)
311                 {
312                     memcpy(dOut, temp, dOutSize);
313                 }
314                 retVal = (UINT16)dOutSize;
315             }
316             else
317             {
318                 retVal = (UINT16)hLen;
319             }
320             state->copySize = 0;
321         }
322         else
323         {
324             retVal = 0; // Indicate that no data is returned
325         }
326     Cleanup:
327         EVP_MD_CTX_cleanup(context);
328         return retVal;
329     }
```

B.8.4.11. _cpri__ImportExportHashState()

This function is used to import or export the hash state. This function would be called to export state when a sequence object was being prepared for export

```
330     LIB_EXPORT void
331     _cpri__ImportExportHashState(
332         CPRI_HASH_STATE     *osslFmt,       // IN/OUT: the hash state formated for use
333                                             //     by openSSL
334         EXPORT_HASH_STATE   *externalFmt,   // IN/OUT: the exported hash state
335         IMPORT_EXPORT        direction      //
336         )
337     {
338         UNREFERENCED_PARAMETER(direction);
339         UNREFERENCED_PARAMETER(externalFmt);
340         UNREFERENCED_PARAMETER(osslFmt);
341         return;
342
343     #if 0
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
344         if(direction == IMPORT_STATE)
345         {
346             // don't have the import export functions yet so just copy
347             _cpri__CopyHashState(osslFmt, (CPRI_HASH_STATE *)externalFmt);
348         }
349         else
350         {
351             _cpri__CopyHashState((CPRI_HASH_STATE *)externalFmt, osslFmt);
352         }
353     #endif
354     }
```

B.8.4.12. _cpri__HashBlock()

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 B.18

Return Value	Meaning
>= 0	number of bytes in *digest* (may be zero)

```
355     LIB_EXPORT UINT16
356     _cpri__HashBlock(
357         TPM_ALG_ID       hashAlg,       // IN: The hash algorithm
358         UINT32           dataSize,      // IN: size of buffer to hash
359         BYTE            *data,          // IN: the buffer to hash
360         UINT32           digestSize,    // IN: size of the digest buffer
361         BYTE            *digest         // OUT: hash digest
362         )
363     {
364         EVP_MD_CTX       hashContext;
365         EVP_MD          *hashServer = NULL;
366         UINT16           retVal = 0;
367         BYTE             b[MAX_DIGEST_SIZE]; // temp buffer in case digestSize not
368         // a full digest
369         unsigned int     dSize = _cpri__GetDigestSize(hashAlg);
370
371
372         // If there is no digest to compute return
373         if(dSize == 0)
374             return 0;
375
376         // After the call to EVP_MD_CTX_init(), will need to call EVP_MD_CTX_cleanup()
377         EVP_MD_CTX_init(&hashContext);     // Initialize the local hash context
378         hashServer = GetHashServer(hashAlg); // Find the hash server
379
380         // It is an error if the digest size is non-zero but there is no server
381         if(   (hashServer == NULL)
382            || (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
383            || (EVP_DigestUpdate(&hashContext, data, dataSize) != 1))
384             FAIL(FATAL_ERROR_INTERNAL);
385         else
386         {
387             // If the size of the digest produced (dSize) is larger than the available
388             // buffer (digestSize), then put the digest in a temp buffer and only copy
389             // the most significant part into the available buffer.
390             if(dSize > digestSize)
391             {
392                 if(EVP_DigestFinal_ex(&hashContext, b, &dSize) != 1)
393                     FAIL(FATAL_ERROR_INTERNAL);
394                 memcpy(digest, b, digestSize);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
395             retVal = (UINT16)digestSize;
396         }
397         else
398         {
399             if((EVP_DigestFinal_ex(&hashContext, digest, &dSize)) != 1)
400                 FAIL(FATAL_ERROR_INTERNAL);
401             retVal = (UINT16) dSize;
402         }
403     }
404     EVP_MD_CTX_cleanup(&hashContext);
405     return retVal;
406 }
```

B.8.5. HMAC Functions

B.8.5.1. _cpri__StartHMAC

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 B.19

Return Value	Meaning
>= 0	number of bytes in digest produced by *hashAlg* (may be zero)

```
407 LIB_EXPORT UINT16
408 _cpri__StartHMAC(
409     TPM_ALG_ID           hashAlg,       // IN: the algorithm to use
410     BOOL                 sequence,      // IN: indicates if the state should be
411                                         //     saved
412     CPRI_HASH_STATE     *state,         // IN/OUT: the state buffer
413     UINT16               keySize,       // IN: the size of the HMAC key
414     BYTE                *key,           // IN: the HMAC key
415     TPM2B               *oPadKey        // OUT: the key prepared for the oPad round
416     )
417 {
418     CPRI_HASH_STATE  localState;
419     UINT16           blockSize = _cpri__GetHashBlockSize(hashAlg);
420     UINT16           digestSize;
421     BYTE            *pb;                // temp pointer
422     UINT32           i;
423
424     // If the key size is larger than the block size, then the hash of the key
425     // is used as the key
426     if(keySize > blockSize)
427     {
428         // large key so digest
429         if((digestSize = _cpri__StartHash(hashAlg, FALSE, &localState)) == 0)
430             return 0;
431         _cpri__UpdateHash(&localState, keySize, key);
432         _cpri__CompleteHash(&localState, digestSize, oPadKey->buffer);
433         oPadKey->size = digestSize;
434     }
435     else
436     {
437         // key size is ok
438         memcpy(oPadKey->buffer, key, keySize);
439         oPadKey->size = keySize;
440     }
441     // XOR the key with iPad (0x36)
442     pb = oPadKey->buffer;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
443         for(i = oPadKey->size; i > 0; i--)
444             *pb++ ^= 0x36;
445 
446         // if the keySize is smaller than a block, fill the rest with 0x36
447         for(i = blockSize - oPadKey->size; i >  0; i--)
448             *pb++ = 0x36;
449 
450         // Increase the oPadSize to a full block
451         oPadKey->size = blockSize;
452 
453         // Start a new hash with the HMAC key
454         // This will go in the caller's state structure and may be a sequence or not
455 
456         if((digestSize = _cpri__StartHash(hashAlg, sequence, state)) > 0)
457         {
458 
459             _cpri__UpdateHash(state, oPadKey->size, oPadKey->buffer);
460 
461             // XOR the key block with 0x5c ^ 0x36
462             for(pb = oPadKey->buffer, i = blockSize; i > 0; i--)
463                 *pb++ ^= (0x5c ^ 0x36);
464         }
465 
466         return digestSize;
467     }
```

B.8.5.2. _cpri_CompleteHMAC()

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 B.20

Return Value	Meaning
>= 0	number of bytes in *dOut* (may be zero)

```
468     LIB_EXPORT UINT16
469     _cpri__CompleteHMAC(
470         CPRI_HASH_STATE     *hashState,     // IN: the state of hash stack
471         TPM2B               *oPadKey,       // IN: the HMAC key in oPad format
472         UINT32              dOutSize,       // IN: size of digest buffer
473         BYTE                *dOut           // OUT: hash digest
474         )
475     {
476         BYTE            digest[MAX_DIGEST_SIZE];
477         CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
478         CPRI_HASH_STATE  localState;
479         UINT16          digestSize = _cpri__GetDigestSize(state->hashAlg);
480 
481 
482         _cpri__CompleteHash(hashState, digestSize, digest);
483 
484         // Using the local hash state, do a hash with the oPad
485         if(_cpri__StartHash(state->hashAlg, FALSE, &localState) != digestSize)
486             return 0;
487 
488         _cpri__UpdateHash(&localState, oPadKey->size, oPadKey->buffer);
489         _cpri__UpdateHash(&localState, digestSize, digest);
490         return _cpri__CompleteHash(&localState, dOutSize, dOut);
491     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.8.6. Mask and Key Generation Functions

B.8.6.1. _crypi_MGF1()

This function performs MGF1 using the selected hash. MGF1 is T(n) = T(n-1) || H(seed || counter). This function returns the length of the mask produced which could be zero if the digest algorithm is not supported.

Table B.21

Return Value	Meaning
0	hash algorithm not supported
> 0	should be the same as *mSize*

```
492  LIB_EXPORT CRYPT_RESULT
493  _cpri__MGF1(
494      UINT32           mSize,         // IN: length of the mask to be produced
495      BYTE            *mask,          // OUT: buffer to receive the mask
496      TPM_ALG_ID       hashAlg,       // IN: hash to use
497      UINT32           sSize,         // IN: size of the seed
498      BYTE            *seed           // IN: seed size
499      )
500  {
501      EVP_MD_CTX           hashContext;
502      EVP_MD              *hashServer = NULL;
503      CRYPT_RESULT         retVal = 0;
504      BYTE                 b[MAX_DIGEST_SIZE]; // temp buffer in case mask is not an
505      // even multiple of a full digest
506      CRYPT_RESULT         dSize = _cpri__GetDigestSize(hashAlg);
507      unsigned int         digestSize = (UINT32)dSize;
508      UINT32               remaining;
509      UINT32               counter;
510      BYTE                 swappedCounter[4];
511
512      // Parameter check
513      if(mSize > (1024*16)) // Semi-arbitrary maximum
514          FAIL(FATAL_ERROR_INTERNAL);
515
516      // If there is no digest to compute return
517      if(dSize <= 0)
518          return 0;
519
520      EVP_MD_CTX_init(&hashContext);     // Initialize the local hash context
521      hashServer = GetHashServer(hashAlg); // Find the hash server
522      if(hashServer == NULL)
523          // If there is no server, then there is no digest
524          return 0;
525
526      for(counter = 0, remaining = mSize; remaining > 0; counter++)
527      {
528          // Because the system may be either Endian...
529          UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
530
531          // Start the hash and include the seed and counter
532          if(   (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
533             || (EVP_DigestUpdate(&hashContext, seed, sSize) != 1)
534             || (EVP_DigestUpdate(&hashContext, swappedCounter, 4) != 1)
535            )
536              FAIL(FATAL_ERROR_INTERNAL);
537
538          // Handling the completion depends on how much space remains in the mask
539          // buffer. If it can hold the entire digest, put it there. If not
540          // put the digest in a temp buffer and only copy the amount that
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
541             // will fit into the mask buffer.
542             if(remaining < (<K>unsigned)dSize)
543             {
544                 if(EVP_DigestFinal_ex(&hashContext, b, &digestSize) != 1)
545                     FAIL(FATAL_ERROR_INTERNAL);
546                 memcpy(mask, b, remaining);
547                 break;
548             }
549             else
550             {
551                 if(EVP_DigestFinal_ex(&hashContext, mask, &digestSize) != 1)
552                     FAIL(FATAL_ERROR_INTERNAL);
553                 remaining -= dSize;
554                 mask = &mask[dSize];
555             }
556             retVal = (CRYPT_RESULT)mSize;
557         }
558
559         EVP_MD_CTX_cleanup(&hashContext);
560         return retVal;
561     }
```

B.8.6.2. _cpri_KDFa()

This function performs the key generation according to ISO/IEC 11889-1.

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 = 256K 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 B.22

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
562     LIB_EXPORT UINT16
563     _cpri__KDFa(
564         TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
565         TPM2B           *key,           // IN: HMAC key
566         const char      *label,         // IN: a 0-byte terminated label used in KDF
567         TPM2B           *contextU,      // IN: context U
568         TPM2B           *contextV,      // IN: context V
569         UINT32           sizeInBits,    // IN: size of generated key in bits
570         BYTE            *keyStream,     // OUT: key buffer
571         UINT32          *counterInOut,  // IN/OUT: caller may provide the iteration
572                                         //     counter for incremental operations to
573                                         //     avoid large intermediate buffers.
574         BOOL             once           // IN: TRUE if only one iteration is performed
575                                         //     FALSE if iteration count determined by
576                                         //     "sizeInBits"
577         )
578     {
579         UINT32                  counter = 0;    // counter value
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
580       INT32                   lLen = 0;       // length of the label
581       INT16                   hLen;           // length of the hash
582       INT16                   bytes;          // number of bytes to produce
583       BYTE                   *stream = keyStream;
584       BYTE                    marshaledUint32[4];
585       CPRI_HASH_STATE         hashState;
586       TPM2B_MAX_HASH_BLOCK    hmacKey;
587
588       pAssert(key != NULL && keyStream != NULL);
589       pAssert(once == FALSE || (sizeInBits & 7) == 0);
590
591       if(counterInOut != NULL)
592           counter = *counterInOut;
593
594       // Prepare label buffer.  Calculate its size and keep the last 0 byte
595       if(label != NULL)
596           for(lLen = 0; label[lLen++] != 0; );
597
598       // Get the hash size.  If it is less than or 0, either the
599       // algorithm is not supported or the hash is TPM_ALG_NULL
600       // In either case the digest size is zero.  This is the only return
601       // other than the one at the end. All other exits from this function
602       // are fatal errors. After we check that the algorithm is supported
603       // anything else that goes wrong is an implementation flaw.
604       if((hLen = (INT16) _cpri__GetDigestSize(hashAlg)) == 0)
605           return 0;
606
607       // If the size of the request is larger than the numbers will handle,
608       // it is a fatal error.
609       pAssert(((sizeInBits + 7)/ 8) <= INT16_MAX);
610
611       bytes = once ? hLen : (INT16)((sizeInBits + 7) / 8);
612
613       // Generate required bytes
614       for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
615       {
616           if(bytes < hLen)
617               hLen = bytes;
618
619           counter++;
620           // Start HMAC
621           if(_cpri__StartHMAC(hashAlg,
622                               FALSE,
623                               &hashState,
624                               key->size,
625                               &key->buffer[0],
626                               &hmacKey.b)         <= 0)
627               FAIL(FATAL_ERROR_INTERNAL);
628
629           // Adding counter
630           UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
631           _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
632
633           // Adding label
634           if(label != NULL)
635               _cpri__UpdateHash(&hashState,  lLen, (BYTE *)label);
636
637           // Adding contextU
638           if(contextU != NULL)
639               _cpri__UpdateHash(&hashState, contextU->size, contextU->buffer);
640
641           // Adding contextV
642           if(contextV != NULL)
643               _cpri__UpdateHash(&hashState, contextV->size, contextV->buffer);
644
645           // Adding size in bits
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
646             UINT32_TO_BYTE_ARRAY(sizeInBits, marshaledUint32);
647             _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
648
649             // Compute HMAC. At the start of each iteration, hLen is set
650             // to the smaller of hLen and bytes. This causes bytes to decrement
651             // exactly to zero to complete the loop
652             _cpri__CompleteHMAC(&hashState, &hmacKey.b, hLen, stream);
653         }
654
655         // Mask off bits if the required bits is not a multiple of byte size
656         if((sizeInBits % 8) != 0)
657             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
658         if(counterInOut != NULL)
659             *counterInOut = counter;
660         return (CRYPT_RESULT)((sizeInBits + 7)/8);
661     }
```

B.8.6.3. _cpri__KDFe()

KDFe() as defined in ISO/IEC 11889-1.

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 = 256K bits (32385 bytes). Any error in the processing of this command is considered fatal.

Table B.23

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
662     LIB_EXPORT UINT16
663     _cpri__KDFe(
664         TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
665         TPM2B           *Z,             // IN: Z
666         const char      *label,         // IN: a 0 terminated label using in KDF
667         TPM2B           *partyUInfo,    // IN: PartyUInfo
668         TPM2B           *partyVInfo,    // IN: PartyVInfo
669         UINT32           sizeInBits,    // IN: size of generated key in bits
670         BYTE            *keyStream      // OUT: key buffer
671         )
672     {
673         UINT32          counter = 0;       // counter value
674         UINT32          lSize = 0;
675         BYTE           *stream = keyStream;
676         CPRI_HASH_STATE         hashState;
677         INT16           hLen = (INT16) _cpri__GetDigestSize(hashAlg);
678         INT16           bytes;             // number of bytes to generate
679         BYTE            marshaledUint32[4];
680
681         pAssert(   keyStream != NULL
682                    && Z != NULL
683                    && ((sizeInBits + 7) / 8) < INT16_MAX);
684
685         if(hLen == 0)
686             return 0;
687
688         bytes = (INT16)((sizeInBits + 7) / 8);
689
690         // Prepare label buffer.  Calculate its size and keep the last 0 byte
691         if(label != NULL)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
692             for(lSize = 0; label[lSize++] != 0;);
693 
694         // Generate required bytes
695         //The inner loop of that KDF uses:
696         //  Hashi := H(counter | Z | OtherInfo) (5)
697         // Where:
698         //  Hashi   the hash generated on the i-th iteration of the loop.
699         //  H()     an approved hash function
700         //  counter a 32-bit counter that is initialized to 1 and incremented
701         //          on each iteration
702         //  Z       the X coordinate of the product of a public ECC key and a
703         //          different private ECC key.
704         //  OtherInfo   a collection of qualifying data for the KDF defined below.
705         //  In this part of ISO/IEC 11889, OtherInfo will be constructed by:
706         //      OtherInfo := Use | PartyUInfo  | PartyVInfo
707         for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
708         {
709             if(bytes < hLen)
710                 hLen = bytes;
711 
712             counter++;
713             // Start hash
714             if(_cpri__StartHash(hashAlg, FALSE,  &hashState) == 0)
715                 return 0;
716 
717             // Add counter
718             UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
719             _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
720 
721             // Add Z
722             if(Z != NULL)
723                 _cpri__UpdateHash(&hashState, Z->size, Z->buffer);
724 
725             // Add label
726             if(label != NULL)
727                 _cpri__UpdateHash(&hashState, lSize, (BYTE *)label);
728             else
729 
730                 // The SP800-108 specification requires a zero between the label
731                 // and the context.
732                 _cpri__UpdateHash(&hashState, 1, (BYTE *)"");
733 
734             // Add PartyUInfo.
735             if(partyUInfo != NULL)
736                 _cpri__UpdateHash(&hashState, partyUInfo->size, partyUInfo->buffer);
737 
738             // Add PartyVInfo
739             if(partyVInfo != NULL)
740                 _cpri__UpdateHash(&hashState, partyVInfo->size, partyVInfo->buffer);
741 
742             // Compute Hash. hLen was changed to be the smaller of bytes or hLen
743             // at the start of each iteration.
744             _cpri__CompleteHash(&hashState, hLen, stream);
745         }
746 
747         // Mask off bits if the required bits is not a multiple of byte size
748         if((sizeInBits % 8) != 0)
749             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
750 
751         return (CRYPT_RESULT)((sizeInBits + 7) / 8);
752 
753     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.9 CpriHashData.c

```
const HASH_INFO   g_hashData[HASH_COUNT + 1] = {
#if   ALG_SHA1 == YES
    {TPM_ALG_SHA1,    SHA1_DIGEST_SIZE,   SHA1_BLOCK_SIZE,
     SHA1_DER_SIZE,   SHA1_DER},
#endif
#if   ALG_SHA256 == YES
    {TPM_ALG_SHA256,    SHA256_DIGEST_SIZE,   SHA256_BLOCK_SIZE,
     SHA256_DER_SIZE,   SHA256_DER},
#endif
#if   ALG_SHA384 == YES
    {TPM_ALG_SHA384,    SHA384_DIGEST_SIZE,   SHA384_BLOCK_SIZE,
     SHA384_DER_SIZE,   SHA384_DER},
#endif
#if   ALG_SHA512 == YES
    {TPM_ALG_SHA512,    SHA512_DIGEST_SIZE,   SHA512_BLOCK_SIZE,
     SHA512_DER_SIZE,   SHA512_DER},
#endif
#if   ALG_WHIRLPOOL512 == YES
    {TPM_ALG_WHIRLPOOL512,    WHIRLPOOL512_DIGEST_SIZE,   WHIRLPOOL512_BLOCK_SIZE,
     WHIRLPOOL512_DER_SIZE,   WHIRLPOOL512_DER},
#endif
#if   ALG_SM3_256 == YES
    {TPM_ALG_SM3_256,    SM3_256_DIGEST_SIZE,   SM3_256_BLOCK_SIZE,
     SM3_256_DER_SIZE,   SM3_256_DER},
#endif
    {TPM_ALG_NULL,0,0,0,{0}}
};
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.10 CpriMisc.c

B.10.1. Includes

```
#include "OsslCryptoEngine.h"
```

B.10.2. Functions

B.10.2.1. BnTo2B()

This function is used to convert a BigNum() to a byte array of the specified size. If the number is too large to fit, then 0 is returned. Otherwise, the number is converted into the low-order bytes of the provided array and the upper bytes are set to zero.

Table B.24

Return Value	Meaning
0	failure (probably fatal)
1	conversion successful

```
BOOL
BnTo2B(
    TPM2B           *outVal,        // OUT: place for the result
    BIGNUM          *inVal,         // IN: number to convert
    UINT16           size           // IN: size of the output.
    )
{
    BYTE    *pb = outVal->buffer;

    outVal->size = size;

    size = size - (((UINT16) BN_num_bits(inVal) + 7) / 8);
    if(size < 0)
        return FALSE;
    for(;size > 0; size--)
        *pb++ = 0;
    BN_bn2bin(inVal, pb);
    return TRUE;
}
```

B.10.2.2. Copy2B()

This function copies a TPM2B structure. The compiler can't generate a copy of a TPM2B generic structure because the actual size is not known. This function performs the copy on any TPM2B pair. The size of the destination should have been checked before this call to make sure that it will hold the TPM2B being copied.

This replicates the functionality in the MemoryLib.c.

```
void
Copy2B(
    TPM2B           *out,           // OUT: The TPM2B to receive the copy
    TPM2B           *in             // IN: the TPM2B to copy
    )
{
    BYTE        *pIn = in->buffer;
    BYTE        *pOut = out->buffer;
    int          count;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
30      out->size = in->size;
31      for(count = in->size; count > 0; count--)
32          *pOut++ = *pIn++;
33      return;
34  }
```

B.10.2.3. BnFrom2B()

This function creates a BIGNUM from a TPM2B and fails if the conversion fails.

```
35  BIGNUM *
36  BnFrom2B(
37      BIGNUM          *out,           // OUT: The BIGNUM
38      const TPM2B     *in             // IN: the TPM2B to copy
39      )
40  {
41      if(BN_bin2bn(in->buffer, in->size, out) == NULL)
42          FAIL(FATAL_ERROR_INTERNAL);
43      return out;
44  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.11 CpriSym.c

B.11.1. Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These function only use the single block encryption and decryption functions of OpesnSSL().

Currently, this module only supports AES and Camellia encryption. SM4 is not implemented in the version of OpenSSL() available to the author.

B.11.2. Includes, Defines, and Typedefs

```
#include    "OsslCryptoEngine.h"
```

SM4 is not implemented in the version of OpenSSL() available to the author

```
#ifdef TPM_ALG_SM4
#error "SM4 is not available"
#endif
typedef union {
#ifdef  TPM_ALG_AES
        AES_KEY     AesKey;
#endif
#ifdef  TPM_ALG_SM4
        SM4_KEY     SM4Key;
#endif
#ifdef  TPM_ALG_CAMELLIA
        CAMELLIA_KEY     CamelliaKey;
#endif
    } keySchedule_t;
typedef void (*encryptCall_t)(
                    const void *in,
                    void *out,
                    void *keySchedule
);
#define SET_ENCRYPT_KEY(ALG, Alg)           \
    if(0 != ALG##_set_encrypt_key(          \
                    key,                    \
                    keySizeInBits,          \
                    &keySchedule.Alg##Key)) \
        FAIL(FATAL_ERROR_INTERNAL);         \
        encrypt = (encryptCall_t)&(ALG##_encrypt)  \

#define SET_DECRYPT_KEY(ALG, Alg)           \
    if(0 != ALG##_set_decrypt_key(          \
                    key,                    \
                    keySizeInBits,          \
                    &keySchedule.Alg##Key)) \
        FAIL(FATAL_ERROR_INTERNAL);         \
        decrypt = (encryptCall_t)&(ALG##_decrypt)    \

#ifdef  TPM_ALG_AES
#   define  SET_AES_ENCRYPT     SET_ENCRYPT_KEY(AES, Aes)
#   define  SET_AES_DECRYPT     SET_DECRYPT_KEY(AES, Aes)
#else
#   define  SET_AES_ENCRYPT     pAssert(0);
#   define  SET_AES_DECRYPT     pAssert(0);
#endif
#ifdef  TPM_ALG_SM4
#   define  SET_SM4_ENCRYPT     SET_ENCRYPT_KEY(SM4, SM4)
#   define  SET_SM4_DECRYPT     SET_DECRYPT_KEY(SM4, SM4)
#else
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
#   define  SET_SM4_ENCRYPT     pAssert(0);
#   define  SET_SM4_DECRYPT     pAssert(0);
#endif
#ifdef  TPM_ALG_CAMELLIA
#   define  SET_CAMELLIA_ENCRYPT     SET_ENCRYPT_KEY(CAMELLIA, Camellia)
#   define  SET_CAMELLIA_DECRYPT     SET_DECRYPT_KEY(CAMELLIA, Camellia)
#else
#   define  SET_CAMELLIA_ENCRYPT    pAssert(0);
#   define  SET_CAMELLIA_DECRYPT    pAssert(0);
#endif
#define     SELECT(algorithm, direction)        \
    switch (algorithm)                          \
    {                                           \
        case ALG_AES_VALUE:                     \
            SET_AES_##direction;                \
            break;                              \
        case ALG_SM4_VALUE:                     \
            SET_SM4_##direction;                \
            break;                              \
        case ALG_CAMELLIA_VALUE:                \
            SET_CAMELLIA_##direction;           \
            break;                              \
        default:                                \
            pAssert(0);                         \
            break;                              \
    }
```

B.11.3. Utility Functions

B.11.3.1. _cpri_SymStartup()

```
LIB_EXPORT BOOL
_cpri__SymStartup(
    void
)
{
    return TRUE;
}
```

B.11.3.2. _cpri__GetSymmetricBlockSize()

This function returns the block size of the algorithm.

Table B.25

xReturn Value	Meaning
<= 0	cipher not supported
> 0	the cipher block size in bytes

```
LIB_EXPORT INT16
_cpri__GetSymmetricBlockSize(
    TPM_ALG_ID       symmetricAlg,  // IN: the symmetric algorithm
    UINT16           keySizeInBits  // IN: the key size
    )
{
    switch (symmetricAlg)
    {
#ifdef TPM_ALG_AES
    case TPM_ALG_AES:
#endif
#ifdef TPM_ALG_CAMELLIA
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
93          // AES, Camellia and SM4 use
94          // the same block size
95      case TPM_ALG_CAMELLIA:
96  #endif
97  #ifdef TPM_ALG_SM4 // Both AES and SM4 use the same block size
98      case TPM_ALG_SM4:
99  #endif
100         if(keySizeInBits != 0)  // This is mostly to have a reference to
101             // keySizeInBits for the compiler
102             return  16;
103         else
104             return 0;
105         break;
106
107     default:
108         return 0;
109     }
110 }
```

B.11.4. Symmetric Encryption

```
111 LIB_EXPORT CRYPT_RESULT
112 _cpri__SymmetricEncrypt(
113     BYTE            *dOut,          // OUT:
114     TPM_ALG_ID       algorithm,     // IN: the symmetric algorithm
115     UINT16           keySizeInBits, // IN: key size in bits
116     const BYTE      *key,           // IN: key buffer. The size of this buffer
117                                     //     in bytes is (keySizeInBits + 7) / 8
118     TPM2B_IV        *ivInOut,       // IN/OUT: IV for decryption.
119     TPM_ALG_ID       mode,          // IN: Mode to use
120     UINT32           dInSize,       // IN: data size (may need to be a
121                                     //     multiple of the blockSize)
122     const BYTE      *dIn            // IN: data buffer
123     )
124 {
125     BYTE            *pIv;
126     INT32            dSize;         // Need a signed version
127     int              i;
128     BYTE             tmp[MAX_SYM_BLOCK_SIZE];
129     BYTE            *pT;
130     keySchedule_t    keySchedule;
131     INT32            blockSize;
132     encryptCall_t    encrypt;
133     BYTE            *iv;
134
135     pAssert(dOut != NULL && key != NULL && ivInOut != NULL && dIn != NULL &&
136             dInSize <= INT32_MAX);
137     if(dInSize == 0)
138         return CRYPT_SUCCESS;
139
140     dSize = (INT32)dInSize;
141     blockSize = ivInOut->t.size;
142     iv = ivInOut->t.buffer;
143
144     // Create encrypt key schedule and set the encryption function pointer
145     SELECT(algorithm, ENCRYPT);
146     switch (mode)
147     {
148         case TPM_ALG_CTR:
149             for(; dSize > 0; dSize -= blockSize)
150             {
151                 // Encrypt the current value of the IV(counter)
152                 encrypt(iv, tmp, &keySchedule);
153
154                 //increment the counter (counter is big-endian so start at end)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
155                 for(i = blockSize-1; i >= 0; i--)
156                     if((iv[i] += 1) != 0)
157                         break;
158 
159                 // XOR the encrypted counter value with input and put into output
160                 pT = tmp;
161                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
162                     *dOut++ = *dIn++ ^ *pT++;
163             }
164             break;
165         case TPM_ALG_OFB:
166             // This is written so that dIn and dOut may be the same
167             for(; dSize > 0; dSize -= blockSize)
168             {
169                 // Encrypt the current value of the "IV"
170                 encrypt(iv, iv, &keySchedule);
171 
172                 // XOR the encrypted IV into dIn to create the cipher text (dOut)
173                 pIv = iv;
174                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
175                     *dOut++ = (*pIv++ ^ *dIn++);
176             }
177             break;
178         case TPM_ALG_CBC:
179             // For CBC the data size must be an even multiple of the
180             // cipher block size
181             if((dSize % blockSize) != 0)
182                 return CRYPT_PARAMETER;
183             // XOR the data block into the IV, encrypt the IV into the IV
184             // and then copy the IV to the output
185             for(; dSize > 0; dSize -= blockSize)
186             {
187                 pIv = iv;
188                 for(i = blockSize; i > 0; i--)
189                     *pIv++ ^= *dIn++;
190                 encrypt(iv, iv, &keySchedule);
191                 pIv = iv;
192                 for(i = blockSize; i > 0; i--)
193                     *dOut++ = *pIv++;
194             }
195             break;
196         case TPM_ALG_CFB:
197             // Encrypt the IV into the IV, XOR in the data, and copy to output
198             for(; dSize > 0; dSize -= blockSize)
199             {
200                 // Encrypt the current value of the IV
201                 encrypt(iv, iv, &keySchedule);
202                 pIv = iv;
203                 for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
204                     // XOR the data into the IV to create the cipher text
205                     // and put into the output
206                     *dOut++ = *pIv++ ^= *dIn++;
207             }
208             // If the inner loop (i loop) was smaller than blockSize, then dSize would
209             // have been smaller than blockSize and it is now negative. If it is
    negative,
210             // then it indicates how many bytes are needed to pad out the IV for
211             // the next round.
212             for(; dSize < 0; dSize++)
213                 *pIv++ = 0;
214             break;
215 
216         case TPM_ALG_ECB:
217             // For ECB the data size must be an even multiple of the
218             // cipher block size
219             if((dSize % blockSize) != 0)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
220                 return CRYPT_PARAMETER;
221             // Encrypt the inpupt block to the output block
222             for(; dSize > 0; dSize -= blockSize)
223             {
224                 encrypt(dIn, dOut, &keySchedule);
225                 dIn = &dIn[blockSize];
226                 dOut = &dOut[blockSize];
227             }
228             break;
229
230         default:
231             pAssert(0);
232     }
233     return CRYPT_SUCCESS;
234 }
```

B.11.4.1. _cpri__SymmetricDecrypt()

This function performs symmetric decryption based on the mode.

Table B.26

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	*dlnSize* is not a multiple of the block size

```
235 LIB_EXPORT CRYPT_RESULT
236 _cpri__SymmetricDecrypt(
237     BYTE            *dOut,          // OUT: the decrypted data
238     TPM_ALG_ID       algorithm,     // IN: the symmetric algorithm
239     UINT16           keySizeInBits, // IN: key size in bits
240     const BYTE      *key,           // IN: key buffer. The size of this buffer
241                                     //     in bytes is (keySizeInBits + 7) / 8
242     TPM2B_IV        *ivInOut,       // IN/OUT: IV for decryption. The size of
243                                     //     this buffer is blockSize in bytes.
244     TPM_ALG_ID       mode,          // IN: the decryption mode
245     UINT32           dInSize,       // IN: data size (may need to be a multiple of
246                                     //     the block size)
247     const BYTE      *dIn            // IN: data buffer
248 )
249 {
250     BYTE            *pIv;
251     INT32            dSize;         // Need a signed version
252     int              i;
253     BYTE             tmp[MAX_SYM_BLOCK_SIZE];
254     BYTE            *pT;
255     keySchedule_t    keySchedule;
256     INT32            blockSize;
257     BYTE            *iv;
258     encryptCall_t    encrypt;
259     encryptCall_t    decrypt;
260
261     pAssert(dOut != NULL && key != NULL && ivInOut != NULL && dIn != NULL &&
262             dInSize <= INT32_MAX);
263     if(dInSize == 0)
264         return CRYPT_SUCCESS;
265
266     dSize = (INT32)dInSize;
267     blockSize = ivInOut->t.size;
268     iv = ivInOut->t.buffer;
269     // Use the mode to select the key schedule to create.
270     switch (mode)
271     {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
272             case TPM_ALG_CBC: // decrypt = decrypt
273             case TPM_ALG_ECB:
274                 // For ECB and CBC, the data size must be an even multiple of the
275                 // cipher block size
276                 if((dSize % blockSize) != 0)
277                     return CRYPT_PARAMETER;
278                 SELECT(algorithm, DECRYPT);
279                 break;
280             // For these algorithms, encrypt and decrypt are the same
281             case TPM_ALG_CFB:
282             case TPM_ALG_CTR:
283             case TPM_ALG_OFB:
284                 SELECT(algorithm, ENCRYPT);
285                 break;
286         }
287         // Now do the mode-dependent decryption
288         switch (mode)
289         {
290             case TPM_ALG_CBC:
291                 // Copy the input data to a temp buffer, decrypt the buffer into the
    output;
292                 // XOR in the IV, and copy the temp buffer to the IV and repeat.
293                 for(; dSize > 0; dSize -= blockSize)
294                 {
295                     pT = tmp;
296                     for(i = blockSize; i > 0; i--)
297                         *pT++ = *dIn++;
298                     decrypt(tmp, dOut, &keySchedule);
299                     pIv = iv;
300                     pT = tmp;
301                     for(i = blockSize; i> 0; i--)
302                     {
303                         *dOut++ ^= *pIv;
304                         *pIv++ = *pT++;
305                     }
306                 }
307                 break;
308
309             case TPM_ALG_CFB:
310                 for(; dSize > 0; dSize -= blockSize)
311                 {
312                     // Encrypt the IV into the temp buffer
313                     encrypt(iv, tmp, &keySchedule);
314                     pT = tmp;
315                     pIv = iv;
316                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
317                         // Copy the current cipher text to IV, XOR
318                         // with the temp buffer and put into the output
319                         *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
320                 }
321                 // If the inner loop (i loop) was smaller than blockSize, then dSize
322                 // would have been smaller than blockSize and it is now negative
323                 // If it is negative, then it indicates how may fill bytes
324                 // are needed to pad out the IV for the next round.
325                 for(; dSize < 0; dSize++)
326                     *pIv++ = 0;
327
328                 break;
329             case TPM_ALG_CTR:
330                 for(; dSize > 0; dSize -= blockSize)
331                 {
332                     // Encrypt the current value of the IV(counter)
333                     encrypt(iv, tmp, &keySchedule);
334
335                     //increment the counter (counter is big-endian so start at end)
336                     for(i = blockSize-1; i >= 0; i--)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
337                     if((iv[i] += 1) != 0)
338                         break;
339
340                 // XOR the encrypted counter value with input and put into output
341                 pT = tmp;
342                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
343                     *dOut++ = *dIn++ ^ *pT++;
344             }
345             break;
346
347         case TPM_ALG_ECB:
348             for(; dSize > 0; dSize -= blockSize)
349             {
350                 decrypt(dIn, dOut, &keySchedule);
351                 dIn = &dIn[blockSize];
352                 dOut = &dOut[blockSize];
353             }
354             break;
355         case TPM_ALG_OFB:
356             // This is written so that dIn and dOut may be the same
357             for(; dSize > 0; dSize -= blockSize)
358             {
359                 // Encrypt the current value of the "IV"
360                 encrypt(iv, iv, &keySchedule);
361
362                 // XOR the encrypted IV into dIn to create the cipher text (dOut)
363                 pIv = iv;
364                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
365                     *dOut++ = (*pIv++ ^ *dIn++);
366             }
367             break;
368     }
369     return CRYPT_SUCCESS;
370 }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.12 RSA Files

B.12.1. CpriRSA.c

B.12.1.1. Introduction

This file contains implementation of crypto primitives for RSA. This is a simulator of a crypto engine. Vendors may replace the implementation in this file with their own library functions.

Integer format: the big integers passed in/out to the function interfaces in this library adopt the same format used in ISO/IEC 11889-1 that states:

“An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer."

The interface uses TPM2B as a big number format for numeric values passed to/from CryptUtil().

B.12.1.2. Includes

```
1   #include "OsslCryptoEngine.h"
```

B.12.1.3. Local Functions

B.12.1.3.1. RsaPrivateExponent()

This function computes the private exponent *de* = 1 mod (*p*-1)*(*q*-1) The inputs are the public modulus and one of the primes.

The results are returned in the key->private structure. The size of that structure is expanded to hold the private exponent. If the computed value is smaller than the public modulus, the private exponent is de-normalized.

Table B.27

Return Value	Meaning
CRYPT_SUCCESS	private exponent computed
CRYPT_PARAMETER	prime is not half the size of the modulus, or the modulus is not evenly divisible by the prime, or no private exponent could be computed from the input parameters

```
2   static CRYPT_RESULT
3   RsaPrivateExponent(
4       RSA_KEY         *key            // IN: the key to augment with the private
5                                       //     exponent
6       )
7   {
8       BN_CTX          *context;
9       BIGNUM          *bnD;
10      BIGNUM          *bnN;
11      BIGNUM          *bnP;
12      BIGNUM          *bnE;
13      BIGNUM          *bnPhi;
14      BIGNUM          *bnQ;
15      BIGNUM          *bnQr;
16      UINT32           fill;
17
18      CRYPT_RESULT     retVal = CRYPT_SUCCESS;     // Assume success
19
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
20      pAssert(key != NULL && key->privateKey != NULL && key->publicKey != NULL);
21
22      context = BN_CTX_new();
23      if(context == NULL)
24          FAIL(FATAL_ERROR_ALLOCATION);
25      BN_CTX_start(context);
26      bnE = BN_CTX_get(context);
27      bnD = BN_CTX_get(context);
28      bnN = BN_CTX_get(context);
29      bnP = BN_CTX_get(context);
30      bnPhi = BN_CTX_get(context);
31      bnQ = BN_CTX_get(context);
32      bnQr = BN_CTX_get(context);
33
34      if(bnQr == NULL)
35          FAIL(FATAL_ERROR_ALLOCATION);
36
37      // Assume the size of the public key value is within range
38      pAssert(key->publicKey->size <= MAX_RSA_KEY_BYTES);
39
40      if(   BN_bin2bn(key->publicKey->buffer, key->publicKey->size, bnN) == NULL
41         || BN_bin2bn(key->privateKey->buffer, key->privateKey->size, bnP) == NULL)
42
43          FAIL(FATAL_ERROR_INTERNAL);
44
45      // If P size is not 1/2 of n size, then this is not a valid value for this
46      // implementation. This will also catch the case were P is input as zero.
47      // This generates a return rather than an assert because the key being loaded
48      // might be SW generated and wrong.
49      if(BN_num_bits(bnP) < BN_num_bits(bnN)/2)
50      {
51          retVal = CRYPT_PARAMETER;
52          goto Cleanup;
53      }
54      // Get q = n/p;
55      if (BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
56          FAIL(FATAL_ERROR_INTERNAL);
57
58      // If there is a remainder, then this is not a valid n
59      if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
60      {
61          retVal = CRYPT_PARAMETER;       // problem may be recoverable
62          goto Cleanup;
63      }
64      // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
65      if(   BN_copy(bnPhi, bnN) == NULL
66         || !BN_sub(bnPhi, bnPhi, bnP)
67         || !BN_sub(bnPhi, bnPhi, bnQ)
68         || !BN_add_word(bnPhi, 1))
69          FAIL(FATAL_ERROR_INTERNAL);
70
71      // Compute the multiplicative inverse
72      BN_set_word(bnE, key->exponent);
73      if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
74      {
75          // Going to assume that the error is caused by a bad
76          // set of parameters. Specifically, an exponent that is
77          // not compatible with the primes. In an implementation that
78          // has better visibility to the error codes, this might be
79          // refined so that failures in the library would return
80          // a more informative value.  Should not assume here that
81          // the error codes will remain unchanged.
82
83          retVal = CRYPT_PARAMETER;
84          goto Cleanup;
85      }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
86
87      fill = key->publicKey->size - BN_num_bytes(bnD);
88      BN_bn2bin(bnD, &key->privateKey->buffer[fill]);
89      memset(key->privateKey->buffer, 0, fill);
90
91      // Change the size of the private key so that it is known to contain
92      // a private exponent rather than a prime.
93      key->privateKey->size = key->publicKey->size;
94
95  Cleanup:
96      BN_CTX_end(context);
97      BN_CTX_free(context);
98      return retVal;
99  }
```

B.12.1.3.2. _cpri__TestKeyRSA()

This function computes the private exponent *de* = 1 mod *(p*-1)*(q*-1) The inputs are the public modulus and one of the primes or two primes.

If both primes are provided, the public modulus is computed. If only one prime is provided, the second prime is computed. In either case, a private exponent is produced and placed in *d*.

If no modular inverse exists, then CRYPT_PARAMETER is returned.

Table B.28

Return Value	Meaning
CRYPT_SUCCESS	private exponent (d) was generated
CRYPT_PARAMETER	one or more parameters are invalid

```
100 LIB_EXPORT CRYPT_RESULT
101 _cpri__TestKeyRSA(
102     TPM2B           *d,             // OUT: the address to receive the private
103                                     //      exponent
104     UINT32           exponent,      // IN: the public modulus
105     TPM2B           *publicKey,     // IN/OUT: an input if only one prime is
106                                     //      provided. an output if both primes are
107                                     //      provided
108     TPM2B           *prime1,        // IN: a first prime
109     TPM2B           *prime2         // IN: an optional second prime
110     )
111 {
112     BN_CTX          *context;
113     BIGNUM          *bnD;
114     BIGNUM          *bnN;
115     BIGNUM          *bnP;
116     BIGNUM          *bnE;
117     BIGNUM          *bnPhi;
118     BIGNUM          *bnQ;
119     BIGNUM          *bnQr;
120     UINT32          fill;
121
122     CRYPT_RESULT    retVal = CRYPT_SUCCESS;     // Assume success
123
124     pAssert(publicKey != NULL && prime1 != NULL);
125     // Make sure that the sizes are within range
126     pAssert(   prime1->size <= MAX_RSA_KEY_BYTES/2
127             && publicKey->size <= MAX_RSA_KEY_BYTES);
128     pAssert( prime2 == NULL || prime2->size < MAX_RSA_KEY_BYTES/2);
129
130     if(publicKey->size/2 != prime1->size)
131         return CRYPT_PARAMETER;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
132
133         context = BN_CTX_new();
134         if(context == NULL)
135             FAIL(FATAL_ERROR_ALLOCATION);
136         BN_CTX_start(context);
137         bnE = BN_CTX_get(context);      // public exponent (e)
138         bnD = BN_CTX_get(context);      // private exponent (d)
139         bnN = BN_CTX_get(context);      // public modulus (n)
140         bnP = BN_CTX_get(context);      // prime1 (p)
141         bnPhi = BN_CTX_get(context);    // (p-1)(q-1)
142         bnQ = BN_CTX_get(context);      // prime2 (q)
143         bnQr = BN_CTX_get(context);     // n mod p
144
145         if(bnQr == NULL)
146             FAIL(FATAL_ERROR_ALLOCATION);
147
148         if(BN_bin2bn(prime1->buffer, prime1->size, bnP) == NULL)
149             FAIL(FATAL_ERROR_INTERNAL);
150
151         // If prime2 is provided, then compute n
152         if(prime2 != NULL)
153         {
154             // Two primes provided so use them to compute n
155             if(BN_bin2bn(prime2->buffer, prime2->size, bnQ) == NULL)
156                 FAIL(FATAL_ERROR_INTERNAL);
157
158             // Make sure that the sizes of the primes are compatible
159             if(BN_num_bits(bnQ) != BN_num_bits(bnP))
160             {
161                 retVal = CRYPT_PARAMETER;
162                 goto Cleanup;
163             }
164             // Multiply the primes to get the public modulus
165
166             if(BN_mul(bnN, bnP, bnQ, context) != 1)
167                 FAIL(FATAL_ERROR_INTERNAL);
168
169             // if the space provided for the public modulus is large enough,
170             // save the created value
171             if(BN_num_bits(bnN) != (publicKey->size * 8))
172             {
173                 retVal = CRYPT_PARAMETER;
174                 goto Cleanup;
175             }
176             BN_bn2bin(bnN, publicKey->buffer);
177         }
178         else
179         {
180             // One prime provided so find the second prime by division
181             BN_bin2bn(publicKey->buffer, publicKey->size, bnN);
182
183             // Get q = n/p;
184             if(BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
185                 FAIL(FATAL_ERROR_INTERNAL);
186
187             // If there is a remainder, then this is not a valid n
188             if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
189             {
190                 retVal = CRYPT_PARAMETER;      // problem may be recoverable
191                 goto Cleanup;
192             }
193         }
194         // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
195         BN_copy(bnPhi, bnN);
196         BN_sub(bnPhi, bnPhi, bnP);
197         BN_sub(bnPhi, bnPhi, bnQ);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
198         BN_add_word(bnPhi, 1);
199         // Compute the multiplicative inverse
200         BN_set_word(bnE, exponent);
201         if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
202         {
203             // Going to assume that the error is caused by a bad set of parameters.
204             // Specifically, an exponent that is not compatible with the primes.
205             // In an implementation that has better visibility to the error codes,
206             // this might be refined so that failures in the library would return
207             // a more informative value.
208             // Do not assume that the error codes will remain unchanged.
209             retVal = CRYPT_PARAMETER;
210             goto Cleanup;
211         }
212         // Return the private exponent.
213         // Make sure it is normalized to have the correct size.
214         d->size = publicKey->size;
215         fill = d->size - BN_num_bytes(bnD);
216         BN_bn2bin(bnD, &d->buffer[fill]);
217         memset(d->buffer, 0, fill);
218     Cleanup:
219         BN_CTX_end(context);
220         BN_CTX_free(context);
221         return retVal;
222     }
```

B.12.1.3.3. 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 B.29

Return Value	Meaning
CRYPT_SUCCESS	encryption complete
CRYPT_PARAMETER	number to exponentiate is larger than the modulus

```
223     static CRYPT_RESULT
224     RSAEP (
225         UINT32          dInOutSize,     // OUT size of the encrypted block
226         BYTE            *dInOut,        // OUT: the encrypted data
227         RSA_KEY         *key            // IN: the key to use
228         )
229     {
230         UINT32          e;
231         BYTE            exponent[4];
232         CRYPT_RESULT retVal;
233
234         e = key->exponent;
235         if(e == 0)
236             e = RSA_DEFAULT_PUBLIC_EXPONENT;
237         UINT32_TO_BYTE_ARRAY(e, exponent);
238
239         //!!! Can put check for test of RSA here
240
241         retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut, 4, exponent,
242                                key->publicKey->size, key->publicKey->buffer);
243
244         // Exponentiation result is stored in-place, thus no space shortage is possible.
245         pAssert(retVal != CRYPT_UNDERFLOW);
246
247         return retVal;
248     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.12.1.3.4. 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 B.30

Return Value	Meaning
CRYPT_SUCCESS	decryption succeeded
CRYPT_PARAMETER	the value to decrypt is larger than the modulus

```
249  static CRYPT_RESULT
250  RSADP (
251      UINT32           dInOutSize,     // IN/OUT: size of decrypted data
252      BYTE            *dInOut,         // IN/OUT: the decrypted data
253      RSA_KEY         *key             // IN: the key
254      )
255  {
256      CRYPT_RESULT retVal;
257
258      //!!! Can put check for RSA tested here
259
260      // Make sure that the pointers are provided and that the private key is present
261      // If the private key is present it is assumed to have been created by
262      // so is presumed good _cpri__PrivateExponent
263      pAssert(key != NULL && dInOut != NULL &&
264              key->publicKey->size == key->publicKey->size);
265
266      // make sure that the value to be decrypted is smaller than the modulus
267      // note: this check is redundant as is also performed by _math__ModExp()
268      // which is optimized for use in RSA operations
269      if(_math__uComp(key->publicKey->size, key->publicKey->buffer,
270                      dInOutSize, dInOut) <= 0)
271          return CRYPT_PARAMETER;
272
273      // _math__ModExp can return CRYPT_PARAMTER or CRYPT_UNDERFLOW but actual
274      // underflow is not possible because everything is in the same buffer.
275      retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut,
276                             key->privateKey->size, key->privateKey->buffer,
277                             key->publicKey->size, key->publicKey->buffer);
278
279      // Exponentiation result is stored in-place, thus no space shortage is possible.
280      pAssert(retVal != CRYPT_UNDERFLOW);
281
282      return retVal;
283  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

B.12.1.3.5. OaepEncode()

This function performs OAEP padding. The size of the buffer to receive the OAEP padded data must equal the size of the modulus.

© ISO/IEC 2015 – All rights reserved

Table B.31

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	*hashAlg* is not valid
CRYPT_FAIL	message size is too large

```
284  static CRYPT_RESULT
285  OaepEncode(
286      UINT32       paddedSize,    // IN: pad value size
287      BYTE        *padded,        // OUT: the pad data
288      TPM_ALG_ID   hashAlg,       // IN: algorithm to use for padding
289      const char  *label,         // IN: null-terminated string (may be NULL)
290      UINT32       messageSize,   // IN: the message size
291      BYTE        *message        // IN: the message being padded
292  #ifdef  TEST_RSA                //
293      ,  BYTE        *testSeed    // IN: optional seed used for testing
294  #endif  // TEST_RSA             //
295  )
296  {
297      UINT32       padLen;
298      UINT32       dbSize;
299      UINT32       i;
300      BYTE         mySeed[MAX_DIGEST_SIZE];
301      BYTE        *seed = mySeed;
302      INT32        hLen = _cpri__GetDigestSize(hashAlg);
303      BYTE         mask[MAX_RSA_KEY_BYTES];
304      BYTE        *pp;
305      BYTE        *pm;
306      UINT32       lSize = 0;
307      CRYPT_RESULT retVal = CRYPT_SUCCESS;
308
309
310      pAssert(padded != NULL && message != NULL);
311
312      // A value of zero is not allowed because the KDF can't produce a result
313      // if the digest size is zero.
314      if(hLen <= 0)
315          return CRYPT_PARAMETER;
316
317      // If a label is provided, get the length of the string, including the
318      // terminator
319      if(label != NULL)
320          lSize = (UINT32)strlen(label) + 1;
321
322      // Basic size check
323      // messageSize <= k  2hLen  2
324      if(messageSize > paddedSize - 2 * hLen - 2)
325          return CRYPT_FAIL;
326
327      // Hash L even if it is null
328      // Offset into padded leaving room for masked seed and byte of zero
329      pp = &padded[hLen + 1];
330      retVal = _cpri__HashBlock(hashAlg, lSize, (BYTE *)label, hLen, pp);
331
332      // concatenate PS of k  mLen  2hLen  2
333      padLen = paddedSize - messageSize - (2 * hLen) - 2;
334      memset(&pp[hLen], 0, padLen);
335      pp[hLen+padLen] = 0x01;
336      padLen += 1;
337      memcpy(&pp[hLen+padLen], message, messageSize);
338
339      // The total size of db = hLen + pad + mSize;
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
340         dbSize = hLen+padLen+messageSize;
341
342         // If testing, then use the provided seed. Otherwise, use values
343         // from the RNG
344     #ifdef  TEST_RSA
345         if(testSeed != NULL)
346             seed = testSeed;
347         else
348     #endif  // TEST_RSA
349             _cpri__GenerateRandom(hLen, mySeed);
350
351         // mask = MGF1 (seed, nSize  hLen  1)
352         if((retVal = _cpri__MGF1(dbSize, mask,  hashAlg, hLen, seed)) < 0)
353             return retVal; // Don't expect an error because hash size is not zero
354                            // was detected in the call to _cpri__HashBlock() above.
355
356         // Create the masked db
357         pm = mask;
358         for(i = dbSize; i > 0; i--)
359             *pp++ ^= *pm++;
360         pp = &padded[hLen + 1];
361
362         // Run the masked data through MGF1
363         if((retVal = _cpri__MGF1(hLen, &padded[1],  hashAlg, dbSize, pp)) < 0)
364             return retVal; // Don't expect zero here as the only case for zero
365                            // was detected in the call to _cpri__HashBlock() above.
366
367         // Now XOR the seed to create masked seed
368         pp = &padded[1];
369         pm = seed;
370         for(i = hLen; i > 0; i--)
371             *pp++ ^= *pm++;
372
373         // Set the first byte to zero
374         *padded = 0x00;
375         return CRYPT_SUCCESS;
376     }
```

B.12.1.3.6. 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 CRYPT_NO_RESULTS.

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 CRYPT_FAIL.

Table B.32

Return Value	Meaning
CRYPT_SUCCESS	decode complete
CRYPT_PARAMETER	the value to decode was larger than the modulus
CRYPT_FAIL	the padding is wrong or the buffer to receive the results is too small

```
377     static CRYPT_RESULT
378     OaepDecode(
379         UINT32          *dataOutSize,   // IN/OUT: the recovered data size
380         BYTE            *dataOut,       // OUT: the recovered data
381         TPM_ALG_ID       hashAlg,       // IN: algorithm to use for padding
382         const char      *label,         // IN: null-terminated string (may be NULL)
383         UINT32           paddedSize,    // IN: the size of the padded data
384         BYTE            *padded         // IN: the padded data
385         )
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
386 {
387     UINT32       dSizeSave;
388     UINT32       i;
389     BYTE         seedMask[MAX_DIGEST_SIZE];
390     INT32        hLen = _cpri__GetDigestSize(hashAlg);
391
392     BYTE         mask[MAX_RSA_KEY_BYTES];
393     BYTE        *pp;
394     BYTE        *pm;
395     UINT32       lSize = 0;
396     CRYPT_RESULT retVal = CRYPT_SUCCESS;
397
398     // Unknown hash
399     pAssert(hLen > 0 && dataOutSize != NULL && dataOut != NULL && padded != NULL);
400
401     // If there is a label, get its size including the terminating 0x00
402     if(label != NULL)
403         lSize = (UINT32)strlen(label) + 1;
404
405     // Set the return size to zero so that it doesn't have to be done on each
406     // failure
407     dSizeSave = *dataOutSize;
408     *dataOutSize = 0;
409
410     // Strange size (anything smaller can't be an OAEP padded block)
411     // Also check for no leading 0
412     if(paddedSize < (<K>unsigned)((2 * hLen) + 2) || *padded != 0)
413         return CRYPT_FAIL;
414
415     // Use the hash size to determine what to put through MGF1 in order
416     // to recover the seedMask
417     if((retVal = _cpri__MGF1(hLen, seedMask,  hashAlg,
418                              paddedSize-hLen-1, &padded[hLen+1])) < 0)
419         return retVal;
420
421     // Recover the seed into seedMask
422     pp = &padded[1];
423     pm = seedMask;
424     for(i = hLen; i > 0; i--)
425         *pm++ ^= *pp++;
426
427     // Use the seed to generate the data mask
428     if((retVal = _cpri__MGF1(paddedSize-hLen-1, mask,  hashAlg,
429                              hLen, seedMask)) < 0)
430         return retVal;
431
432     // Use the mask generated from seed to recover the padded data
433     pp = &padded[hLen+1];
434     pm = mask;
435     for(i = paddedSize-hLen-1; i > 0; i--)
436         *pm++ ^= *pp++;
437
438     // Make sure that the recovered data has the hash of the label
439     // Put trial value in the seed mask
440     if((retVal=_cpri__HashBlock(hashAlg, lSize,(BYTE *)label, hLen, seedMask)) < 0)
441         return retVal;
442
443     if(memcmp(seedMask, mask, hLen) != 0)
444         return CRYPT_FAIL;
445
446
447     // find the start of the data
448     pm = &mask[hLen];
449     for(i = paddedSize-(2*hLen)-1; i > 0; i--)
450     {
451         if(*pm++ != 0)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
452                 break;
453         }
454         if(i == 0)
455             return CRYPT_PARAMETER;
456 
457         // pm should be pointing at the first part of the data
458         // and i is one greater than the number of bytes to move
459         i--;
460         if(i > dSizeSave)
461         {
462             // Restore dSize
463             *dataOutSize = dSizeSave;
464             return CRYPT_FAIL;
465         }
466         memcpy(dataOut, pm, i);
467         *dataOutSize = i;
468         return CRYPT_SUCCESS;
469     }
```

B.12.1.3.7. PKSC1v1_5Encode()

This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2. 1.

Table B.33

Return Value	Meaning
CRYPT_SUCCESS	data encoded
CRYPT_PARAMETER	message size is too large

```
470     static CRYPT_RESULT
471     RSAES_PKSC1v1_5Encode(
472         UINT32           paddedSize,    // IN: pad value size
473         BYTE            *padded,        // OUT: the pad data
474         UINT32           messageSize,   // IN: the message size
475         BYTE            *message        // IN: the message being padded
476         )
477     {
478         UINT32      ps = paddedSize - messageSize - 3;
479         if(messageSize > paddedSize - 11)
480             return CRYPT_PARAMETER;
481 
482         // move the message to the end of the buffer
483         memcpy(&padded[paddedSize - messageSize], message, messageSize);
484 
485         // Set the first byte to 0x00 and the second to 0x02
486         *padded = 0;
487         padded[1] = 2;
488 
489         // Fill with random bytes
490         _cpri__GenerateRandom(ps, &padded[2]);
491 
492         // Set the delimiter for the random field to 0
493         padded[2+ps] = 0;
494 
495         // Now, the only messy part. Make sure that all the ps bytes are non-zero
496         // In this implementation, use the value of the current index
497         for(ps++; ps > 1; ps--)
498         {
499             if(padded[ps] == 0)
500                 padded[ps] = 0x55;    // In the < 0.5% of the cases that the random
501                                       // value is 0, just pick a value to put into
502                                       // the spot.
503         }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
504         return CRYPT_SUCCESS;
505     }
```

B.12.1.3.8. RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2. 1.

Table B.34

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decoding error or results would no fit into provided buffer

```
506     static CRYPT_RESULT
507     RSAES_Decode(
508         UINT32          *messageSize,   // IN/OUT: recovered message size
509         BYTE            *message,       // OUT: the recovered message
510         UINT32           codedSize,     // IN: the encoded message size
511         BYTE            *coded          // IN: the encoded message
512         )
513     {
514         BOOL        fail = FALSE;
515         UINT32      ps;
516
517         fail = (codedSize < 11);
518         fail |= (coded[0] != 0x00) || (coded[1] != 0x02);
519         for(ps = 2; ps < codedSize; ps++)
520         {
521             if(coded[ps] == 0)
522                 break;
523         }
524         ps++;
525
526         // Make sure that ps has not gone over the end and that there are at least 8
527         // bytes of pad data.
528         fail |= ((ps >= codedSize) || ((ps-2) < 8));
529         if((*messageSize < codedSize - ps) || fail)
530             return CRYPT_FAIL;
531
532         *messageSize = codedSize - ps;
533         memcpy(message, &coded[ps], codedSize - ps);
534         return CRYPT_SUCCESS;
535     }
```

B.12.1.3.9. 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.

Table B.35

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	hashAlg is not a supported hash algorithm

```
536     static CRYPT_RESULT
537     PssEncode   (
538         UINT32       eOutSize,      // IN: size of the encode data buffer
539         BYTE        *eOut,          // OUT: encoded data buffer
540         TPM_ALG_ID   hashAlg,       // IN: hash algorithm to use for the encoding
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
541         UINT32       hashInSize,     // IN: size of digest to encode
542         BYTE        *hashIn          // IN: the digest
543     #ifdef TEST_RSA                  //
544         ,  BYTE        *saltIn       // IN: optional parameter for testing
545     #endif // TEST_RSA               //
546     )
547     {
548         INT32               hLen = _cpri__GetDigestSize(hashAlg);
549         BYTE                salt[MAX_RSA_KEY_BYTES - 1];
550         UINT16              saltSize;
551         BYTE               *ps = salt;
552         CRYPT_RESULT        retVal;
553         UINT16              mLen;
554         CPRI_HASH_STATE     hashState;
555
556         // These are fatal errors indicating bad TPM firmware
557         pAssert(eOut != NULL && hLen > 0 && hashIn != NULL );
558
559         // Get the size of the mask
560         mLen = (UINT16)(eOutSize - hLen - 1);
561
562         // Use the maximum salt size
563         saltSize = mLen - 1;
564
565     //using eOut for scratch space
566         // Set the first 8 bytes to zero
567         memset(eOut, 0, 8);
568
569
570         // Get set the salt
571     #ifdef  TEST_RSA
572         if(saltIn != NULL)
573         {
574             saltSize = hLen;
575             memcpy(salt, saltIn, hLen);
576         }
577         else
578     #endif  // TEST_RSA
579             _cpri__GenerateRandom(saltSize, salt);
580
581         // Create the hash of the pad || input hash || salt
582         _cpri__StartHash(hashAlg, FALSE, &hashState);
583         _cpri__UpdateHash(&hashState, 8, eOut);
584         _cpri__UpdateHash(&hashState, hashInSize, hashIn);
585         _cpri__UpdateHash(&hashState, saltSize, salt);
586         _cpri__CompleteHash(&hashState, hLen, &eOut[eOutSize - hLen - 1]);
587
588         // Create a mask
589         if((retVal = _cpri__MGF1(mLen, eOut, hashAlg, hLen, &eOut[mLen])) < 0)
590         {
591             // Currently _cpri__MGF1 is not expected to return a CRYPT_RESULT error.
592             pAssert(0);
593         }
594         // Since this implementation uses key sizes that are all even multiples of
595         // 8, just need to make sure that the most significant bit is CLEAR
596         eOut[0] &= 0x7f;
597
598         // Before we mess up the eOut value, set the last byte to 0xbc
599         eOut[eOutSize - 1] = 0xbc;
600
601         // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
602         eOut = &eOut[mLen - saltSize - 1];
603         *eOut++ ^= 0x01;
604
605         // XOR the salt data into the buffer
606         for(; saltSize > 0; saltSize--)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
607             *eOut++ ^= *ps++;
608
609         // and we are done
610         return CRYPT_SUCCESS;
611     }
```

B.12.1.3.10. PssDecode()

This function checks that the PSS encoded block was built from the provided digest. If the check is successful, CRYPT_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 B.36

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_SCHEME	*hashAlg* is not a supported hash algorithm
CRYPT_FAIL	decode operation failed

```
612     static CRYPT_RESULT
613     PssDecode(
614         TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use for the encoding
615         UINT32           dInSize,       // IN: size of the digest to compare
616         BYTE            *dIn,           // In: the digest to compare
617         UINT32           eInSize,       // IN: size of the encoded data
618         BYTE            *eIn,           // IN: the encoded data
619         UINT32           saltSize       // IN: the expected size of the salt
620         )
621     {
622         INT32            hLen = _cpri__GetDigestSize(hashAlg);
623         BYTE             mask[MAX_RSA_KEY_BYTES];
624         BYTE            *pm = mask;
625         BYTE             pad[8] = {0};
626         UINT32           i;
627         UINT32           mLen;
628         BOOL             fail = FALSE;
629         CRYPT_RESULT     retVal;
630         CPRI_HASH_STATE  hashState;
631
632         // These errors are indicative of failures due to programmer error
633         pAssert(dIn != NULL && eIn != NULL);
634
635         // check the hash scheme
636         if(hLen == 0)
637             return CRYPT_SCHEME;
638
639         // most significant bit must be zero
640         fail = ((eIn[0] & 0x80) != 0);
641
642         // last byte must be 0xbc
643         fail |= (eIn[eInSize - 1] != 0xbc);
644
645         // Use the hLen bytes at the end of the buffer to generate a mask
646         // Doesn't start at the end which is a flag byte
647         mLen = eInSize - hLen - 1;
648         if((retVal = _cpri__MGF1(mLen, mask, hashAlg, hLen, &eIn[mLen])) < 0)
649             return retVal;
650         if(retVal == 0)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
651             return CRYPT_FAIL;
652
653         // Clear the MSO of the mask to make it consistent with the encoding.
654         mask[0] &= 0x7F;
655
656         // XOR the data into the mask to recover the salt. This sequence
657         // advances eIn so that it will end up pointing to the seed data
658         // which is the hash of the signature data
659         for(i = mLen; i > 0; i--)
660             *pm++ ^= *eIn++;
661
662         // Find the first byte of 0x01 after a string of all 0x00
663         for(pm = mask, i = mLen; i > 0; i--)
664         {
665             if(*pm == 0x01)
666                 break;
667             else
668                 fail |= (*pm++ != 0);
669         }
670         fail |= (i == 0);
671
672         // if we have failed, will continue using the entire mask as the salt value so
673         // that the timing attacks will not disclose anything (I don't think that this
674         // is a problem for TPM applications but, usually, we don't fail so this
675         // doesn't cost anything).
676         if(fail)
677         {
678             i = mLen;
679             pm = mask;
680         }
681         else
682         {
683             pm++;
684             i--;
685         }
686         // If the salt size was provided, then the recovered size must match
687         fail |= (saltSize != 0 && i != saltSize);
688
689         // i contains the salt size and pm points to the salt. Going to use the input
690         // hash and the seed to recreate the hash in the lower portion of eIn.
691         _cpri__StartHash(hashAlg, FALSE, &hashState);
692
693         // add the pad of 8 zeros
694         _cpri__UpdateHash(&hashState, 8, pad);
695
696         // add the provided digest value
697         _cpri__UpdateHash(&hashState, dInSize, dIn);
698
699         // and the salt
700         _cpri__UpdateHash(&hashState, i, pm);
701
702         // get the result
703         retVal = _cpri__CompleteHash(&hashState, MAX_DIGEST_SIZE, mask);
704
705         // retVal will be the size of the digest or zero. If not equal to the indicated
706         // digest size, then the signature doesn't match
707         fail |= (retVal != hLen);
708         fail |= (memcmp(mask, eIn, hLen) != 0);
709         if(fail)
710             return CRYPT_FAIL;
711         else
712             return CRYPT_SUCCESS;
713     }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.12.1.3.11. PKSC1v1_5SignEncode()

Encode a message using PKCS1v1(). 5 method.

Table B.37

Return Value	Meaning
CRYPT_SUCCESS	encode complete
CRYPT_SCHEME	*hashAlg* is not a supported hash algorithm
CRYPT_PARAMETER	*eOutSize* is not large enough or *hInSize* does not match the digest size of *hashAlg*

```
714  static CRYPT_RESULT
715  RSASSA_Encode(
716      UINT32           eOutSize,      // IN: the size of the resulting block
717      BYTE            *eOut,          // OUT: the encoded block
718      TPM_ALG_ID       hashAlg,       // IN: hash algorithm for PKSC1v1_5
719      UINT32           hInSize,       // IN: size of hash to be signed
720      BYTE            *hIn            // IN: hash buffer
721      )
722  {
723      BYTE            *der;
724      INT32            derSize = _cpri__GetHashDER(hashAlg, &der);
725      INT32            fillSize;
726
727      pAssert(eOut != NULL && hIn != NULL);
728
729      // Can't use this scheme if the algorithm doesn't have a DER string defined.
730      if(derSize == 0 )
731          return CRYPT_SCHEME;
732
733      // If the digest size of 'hashAl' doesn't match the input digest size, then
734      // the DER will misidentify the digest so return an error
735      if((unsigned)_cpri__GetDigestSize(hashAlg) != hInSize)
736          return CRYPT_PARAMETER;
737
738      fillSize = eOutSize - derSize - hInSize - 3;
739
740      // Make sure that this combination will fit in the provided space
741      if(fillSize < 8)
742          return CRYPT_PARAMETER;
743      // Start filling
744      *eOut++ = 0; // initial byte of zero
745      *eOut++ = 1; // byte of 0x01
746      for(; fillSize > 0; fillSize--)
747          *eOut++ = 0xff; // bunch of 0xff
748      *eOut++ = 0; // another 0
749      for(; derSize > 0; derSize--)
750          *eOut++ = *der++;   // copy the DER
751      for(; hInSize > 0; hInSize--)
752          *eOut++ = *hIn++;   // copy the hash
753      return CRYPT_SUCCESS;
754  }
```

B.12.1.3.12. RSASSA_Decode()

This function performs the RSASSA decoding of a signature.

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

Table B.38

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decode unsuccessful
CRYPT_SCHEME	*haslAlg* is not supported

```
755  static CRYPT_RESULT
756  RSASSA_Decode(
757      TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use for the encoding
758      UINT32           hInSize,       // IN: size of the digest to compare
759      BYTE            *hIn,           // In: the digest to compare
760      UINT32           eInSize,       // IN: size of the encoded data
761      BYTE            *eIn            // IN: the encoded data
762      )
763  {
764      BOOL             fail = FALSE;
765      BYTE            *der;
766      INT32            derSize = _cpri__GetHashDER(hashAlg, &der);
767      INT32            hashSize = _cpri__GetDigestSize(hashAlg);
768      INT32            fillSize;
769
770      pAssert(hIn != NULL && eIn != NULL);
771
772      // Can't use this scheme if the algorithm doesn't have a DER string
773      // defined or if the provided hash isn't the right size
774      if(derSize == 0 || (unsigned)hashSize != hInSize)
775          return CRYPT_SCHEME;
776
777      // Make sure that this combination will fit in the provided space
778      // Since no data movement takes place, can just walk though this
779      // and accept nearly random values. This can only be called from
780      // _cpri__ValidateSignature() so eInSize is known to be in range.
781      fillSize = eInSize - derSize - hashSize - 3;
782
783      // Start checking
784      fail |= (*eIn++ != 0); // initial byte of zero
785      fail |= (*eIn++ != 1); // byte of 0x01
786      for(; fillSize > 0; fillSize--)
787          fail |= (*eIn++ != 0xff); // bunch of 0xff
788      fail |= (*eIn++ != 0); // another 0
789      for(; derSize > 0; derSize--)
790          fail |= (*eIn++ != *der++); // match the DER
791      for(; hInSize > 0; hInSize--)
792          fail |= (*eIn++ != *hIn++); // match the hash
793      if(fail)
794          return CRYPT_FAIL;
795      return CRYPT_SUCCESS;
796  }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

B.12.1.4. Externally Accessible Functions

B.12.1.4.1. _cpri__RsaStartup()

Function that is called to initialize the hash service. In this implementation, this function does nothing but it is called by the CryptUtilStartup() function and must be present.

```
797  LIB_EXPORT BOOL
798  _cpri__RsaStartup(
799      void
800      )
```

© ISO/IEC 2015 – All rights reserved

```
{
    return TRUE;
}
```

B.12.1.4.2. _cpri__EncryptRSA()

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 treaded 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 B.39

Return Value	Meaning
CRYPT_SUCCESS	encryption complete
CRYPT_PARAMETER	*cOutSize* is too small (must be the size of the modulus)
CRYPT_SCHEME	*padType* is not a supported scheme

```
LIB_EXPORT CRYPT_RESULT
_cpri__EncryptRSA(
    UINT32          *cOutSize,      // OUT: the size of the encrypted data
    BYTE            *cOut,          // OUT: the encrypted data
    RSA_KEY         *key,           // IN: the key to use for encryption
    TPM_ALG_ID       padType,       // IN: the type of padding
    UINT32           dInSize,       // IN: the amount of data to encrypt
    BYTE            *dIn,           // IN: the data to encrypt
    TPM_ALG_ID       hashAlg,       // IN: in case this is needed
    const char      *label          // IN: in case it is needed
    )
{
    CRYPT_RESULT    retVal = CRYPT_SUCCESS;

    pAssert(cOutSize != NULL);

    // All encryption schemes return the same size of data
    if(*cOutSize < key->publicKey->size)
        return CRYPT_PARAMETER;
    *cOutSize = key->publicKey->size;

    switch (padType)
    {
    case TPM_ALG_NULL:  // 'raw' encryption
        {
            // dIn can have more bytes than cOut as long as the extra bytes
            // are zero
            for(; dInSize > *cOutSize; dInSize--)
            {
                if(*dIn++ != 0)
                    return CRYPT_PARAMETER;

            }
            // If dIn is smaller than cOut, fill cOut with zeros
            if(dInSize < *cOutSize)
                memset(cOut, 0, *cOutSize - dInSize);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved

```
840
841             // Copy the rest of the value
842             memcpy(&cOut[*cOutSize-dInSize], dIn, dInSize);
843             // If the size of dIn is the same as cOut dIn could be larger than
844             // the modulus. If it is, then RSAEP() will catch it.
845         }
846         break;
847     case TPM_ALG_RSAES:
848         retVal = RSAES_PKSC1v1_5Encode(*cOutSize, cOut, dInSize, dIn);
849         break;
850     case TPM_ALG_OAEP:
851         retVal = OaepEncode(*cOutSize, cOut, hashAlg, label, dInSize, dIn
852 #ifdef  TEST_RSA
853                             ,NULL
854 #endif
855                            );
856         break;
857     default:
858         return CRYPT_SCHEME;
859     }
860     // All the schemes that do padding will come here for the encryption step
861     // Check that the Encoding worked
862     if(retVal != CRYPT_SUCCESS)
863         return retVal;
864
865     // Padding OK so do the encryption
866     return RSAEP(*cOutSize, cOut, key);
867 }
```

B.12.1.4.3. _cpri__DecryptRSA()

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 B.40

Return Value	Meaning
CRYPT_SUCCESS	successful completion
CRYPT_PARAMETER	*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
CRYPT_FAIL	*dOutSize* is not large enough for the result
CRYPT_SCHEME	*padType* is not supported

```
868 LIB_EXPORT CRYPT_RESULT
869 _cpri__DecryptRSA(
870     UINT32          *dOutSize,      // OUT: the size of the decrypted data
871     BYTE            *dOut,          // OUT: the decrypted data
872     RSA_KEY         *key,           // IN: the key to use for decryption
873     TPM_ALG_ID       padType,       // IN: the type of padding
874     UINT32           cInSize,       // IN: the amount of data to decrypt
875     BYTE            *cIn,           // IN: the data to decrypt
876     TPM_ALG_ID       hashAlg,       // IN: in case this is needed for the scheme
877     const char      *label          // IN: in case it is needed for the scheme
878     )
879 {
880     CRYPT_RESULT    retVal;
881
882     // Make sure that the necessary parameters are provided
883     pAssert(cIn != NULL && dOut != NULL && dOutSize != NULL && key != NULL);
884
885     // Size is checked to make sure that the decryption works properly
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
886         if(cInSize != key->publicKey->size)
887             return CRYPT_PARAMETER;
888 
889         // For others that do padding, do the decryption in place and then
890         // go handle the decoding.
891         if((retVal = RSADP(cInSize, cIn, key)) != CRYPT_SUCCESS)
892             return retVal;      // Decryption failed
893 
894         // Remove padding
895         switch (padType)
896         {
897         case TPM_ALG_NULL:
898             if(*dOutSize < key->publicKey->size)
899                 return CRYPT_FAIL;
900             *dOutSize = key->publicKey->size;
901             memcpy(dOut, cIn, *dOutSize);
902             return CRYPT_SUCCESS;
903         case TPM_ALG_RSAES:
904             return RSAES_Decode(dOutSize, dOut, cInSize, cIn);
905             break;
906         case TPM_ALG_OAEP:
907             return OaepDecode(dOutSize, dOut, hashAlg, label, cInSize, cIn);
908             break;
909         default:
910             return CRYPT_SCHEME;
911             break;
912         }
913     }
```

B.12.1.4.4. _cpri__SignRSA()

This function is used to generate an RSA signature of the type indicated in *scheme*.

Table B.41

Return Value	Meaning
CRYPT_SUCCESS	sign operation completed normally
CRYPT_SCHEME	*scheme* or *hashAlg* are not supported
CRYPT_PARAMETER	*hInSize* does not match *hashAlg* (for RSASSA)

```
914     LIB_EXPORT CRYPT_RESULT
915     _cpri__SignRSA(
916         UINT32          *sigOutSize,    // OUT: size of signature
917         BYTE            *sigOut,        // OUT: signature
918         RSA_KEY         *key,           // IN: key to use
919         TPM_ALG_ID       scheme,        // IN: the scheme to use
920         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for PKSC1v1_5
921         UINT32           hInSize,       // IN: size of digest to be signed
922         BYTE            *hIn            // IN: digest buffer
923         )
924     {
925         CRYPT_RESULT    retVal;
926 
927         // Parameter checks
928         pAssert(sigOutSize != NULL && sigOut != NULL && key != NULL && hIn != NULL);
929 
930 
931         // For all signatures the size is the size of the key modulus
932         *sigOutSize = key->publicKey->size;
933         switch (scheme)
934         {
935         case TPM_ALG_NULL:
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
936             *sigOutSize = 0;
937             return CRYPT_SUCCESS;
938         case TPM_ALG_RSAPSS:
939             // PssEncode can return CRYPT_PARAMETER
940             retVal = PssEncode(*sigOutSize, sigOut, hashAlg, hInSize, hIn
941     #ifdef  TEST_RSA
942                                , NULL
943     #endif
944                               );
945             break;
946         case TPM_ALG_RSASSA:
947             // RSASSA_Encode can return CRYPT_PARAMETER or CRYPT_SCHEME
948             retVal = RSASSA_Encode(*sigOutSize, sigOut, hashAlg, hInSize, hIn);
949             break;
950         default:
951             return CRYPT_SCHEME;
952         }
953         if(retVal != CRYPT_SUCCESS)
954             return retVal;
955         // Do the encryption using the private key
956         // RSADP can return CRYPT_PARAMETR
957         return RSADP(*sigOutSize,sigOut, key);
958     }
```

B.12.1.4.5. _cpri__ValidateSignatureRSA()

This function is used to validate an RSA signature. If the signature is valid CRYPT_SUCCESS is returned. If the signature is not valid, CRYPT_FAIL is returned. Other return codes indicate either parameter problems or fatal errors.

Table B.42

Return Value	Meaning
CRYPT_SUCCESS	the signature checks
CRYPT_FAIL	the signature does not check
CRYPT_SCHEME	unsupported scheme or hash algorithm

```
959     LIB_EXPORT CRYPT_RESULT
960     _cpri__ValidateSignatureRSA(
961         RSA_KEY         *key,           // IN: key to use
962         TPM_ALG_ID       scheme,        // IN: the scheme to use
963         TPM_ALG_ID       hashAlg,       // IN: hash algorithm
964         UINT32           hInSize,       // IN: size of digest to be checked
965         BYTE            *hIn,           // IN: digest buffer
966         UINT32           sigInSize,     // IN: size of signature
967         BYTE            *sigIn,         // IN: signature
968         UINT16           saltSize       // IN: salt size for PSS
969         )
970     {
971         CRYPT_RESULT     retVal;
972
973         // Fatal programming errors
974         pAssert(key != NULL && sigIn != NULL && hIn != NULL);
975
976         // Errors that might be caused by calling parameters
977         if(sigInSize != key->publicKey->size)
978             return CRYPT_FAIL;
979         // Decrypt the block
980         if((retVal = RSAEP(sigInSize, sigIn, key)) != CRYPT_SUCCESS)
981             return CRYPT_FAIL;
982         switch (scheme)
983         {
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
984         case TPM_ALG_NULL:
985             return CRYPT_SCHEME;
986             break;
987         case TPM_ALG_RSAPSS:
988             return PssDecode(hashAlg, hInSize, hIn, sigInSize, sigIn, saltSize);
989             break;
990         case TPM_ALG_RSASSA:
991             return RSASSA_Decode(hashAlg, hInSize, hIn, sigInSize, sigIn);
992             break;
993         default:
994             break;
995         }
996         return CRYPT_SCHEME;
997     }
998     #ifndef RSA_KEY_SIEVE
```

B.12.1.4.6. _cpri__GenerateKeyRSA()

Generate an RSA key from a provided seed.

Table B.43

Return Value	Meaning
CRYPT_FAIL	exponent is not prime or is less than 3; or could not find a prime using the provided parameters
CRYPT_CANCEL	operation was canceled

```
999     LIB_EXPORT CRYPT_RESULT
1000    _cpri__GenerateKeyRSA(
1001        TPM2B           *n,             // OUT: The public modulus
1002        TPM2B           *p,             // OUT: One of the prime factors of n
1003        UINT16           keySizeInBits, // IN: Size of the public modulus in bits
1004        UINT32           e,             // IN: The public exponent
1005        TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use in the key
1006                                        //     generation process
1007        TPM2B           *seed,          // IN: the seed to use
1008        const char      *label,         // IN: A label for the generation process.
1009        TPM2B           *extra,         // IN: Party 1 data for the KDF
1010        UINT32          *counter        // IN/OUT: Counter value to allow KFD iteration
1011                                        //     to be propagated across multiple routines
1012        )
1013    {
1014        UINT32           lLen;          // length of the label
1015                                        // (counting the terminating 0);
1016        UINT16           digestSize = _cpri__GetDigestSize(hashAlg);
1017
1018        TPM2B_HASH_BLOCK    oPadKey;
1019
1020        UINT32          outer;
1021        UINT32          inner;
1022        BYTE            swapped[4];
1023
1024        CRYPT_RESULT    retVal;
1025        int             i, fill;
1026        const static char     defaultLabel[] = "RSA key";
1027        BYTE            *pb;
1028
1029
1030        CPRI_HASH_STATE  h1;            // contains the hash of the
1031                                        //   HMAC key w/ iPad
1032        CPRI_HASH_STATE  h2;            // contains the hash of the
1033                                        //   HMAC key w/ oPad
1034        CPRI_HASH_STATE  h;             // the working hash context
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1035
1036        BIGNUM          *bnP;
1037        BIGNUM          *bnQ;
1038        BIGNUM          *bnT;
1039        BIGNUM          *bnE;
1040        BIGNUM          *bnN;
1041        BN_CTX          *context;
1042        UINT32           rem;
1043
1044        // Make sure that hashAlg is valid hash
1045        pAssert(digestSize != 0);
1046
1047        // if present, use externally provided counter
1048        if(counter != NULL)
1049            outer = *counter;
1050        else
1051            outer = 1;
1052
1053        // Validate exponent
1054        UINT32_TO_BYTE_ARRAY(e, swapped);
1055
1056        // Need to check that the exponent is prime and not less than 3
1057        if( e != 0 && (e < 3  || !_math__IsPrime(e)))
1058            return CRYPT_FAIL;
1059
1060        // Get structures for the big number representations
1061        context = BN_CTX_new();
1062        if(context == NULL)
1063            FAIL(FATAL_ERROR_ALLOCATION);
1064        BN_CTX_start(context);
1065        bnP = BN_CTX_get(context);
1066        bnQ = BN_CTX_get(context);
1067        bnT = BN_CTX_get(context);
1068        bnE = BN_CTX_get(context);
1069        bnN = BN_CTX_get(context);
1070        if(bnN == NULL)
1071            FAIL(FATAL_ERROR_INTERNAL);
1072
1073        // Set Q to zero. This is used as a flag. The prime is computed in P. When a
1074        // new prime is found, Q is checked to see if it is zero.  If so, P is copied
1075        // to Q and a new P is found.  When both P and Q are non-zero, the modulus and
1076        // private exponent are computed and a trial encryption/decryption is
1077        // performed.  If the encrypt/decrypt fails, assume that at least one of the
1078        // primes is composite. Since we don't know which one, set Q to zero and start
1079        // over and find a new pair of primes.
1080        BN_zero(bnQ);
1081
1082        // Need to have some label
1083        if(label == NULL)
1084            label = (const char *)&defaultLabel;
1085        // Get the label size
1086        for(lLen = 0; label[lLen++] != 0;);
1087
1088
1089        // Start the hash using the seed and get the intermediate hash value
1090        _cpri__StartHMAC(hashAlg, FALSE, &h1, seed->size, seed->buffer, &oPadKey.b);
1091        _cpri__StartHash(hashAlg, FALSE, &h2);
1092        _cpri__UpdateHash(&h2, oPadKey.b.size, oPadKey.b.buffer);
1093
1094        n->size = (keySizeInBits +7)/8;
1095        pAssert(n->size <= MAX_RSA_KEY_BYTES);
1096        p->size = n->size / 2;
1097        if(e == 0)
1098            e = RSA_DEFAULT_PUBLIC_EXPONENT;
1099
1100        BN_set_word(bnE, e);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1101
1102        // The first test will increment the counter from zero.
1103        for(outer += 1; outer != 0; outer++)
1104        {
1105            if(_plat__IsCanceled())
1106            {
1107                retVal = CRYPT_CANCEL;
1108                goto Cleanup;
1109            }
1110
1111            // Need to fill in the candidate with the hash
1112            fill = digestSize;
1113            pb = p->buffer;
1114
1115            // Reset the inner counter
1116            inner = 0;
1117            for(i = p->size; i > 0; i -= digestSize)
1118            {
1119                inner++;
1120                // Initialize the HMAC with saved state
1121                _cpri__CopyHashState(&h, &h1);
1122
1123                // Hash the inner counter (the one that changes on each HMAC iteration)
1124                UINT32_TO_BYTE_ARRAY(inner, swapped);
1125                _cpri__UpdateHash(&h, 4, swapped);
1126                _cpri__UpdateHash(&h, lLen, (BYTE *)label);
1127
1128                // Is there any party 1 data
1129                if(extra != NULL)
1130                    _cpri__UpdateHash(&h, extra->size, extra->buffer);
1131
1132                // Include the outer counter (the one that changes on each prime
1133                // prime candidate generation
1134                UINT32_TO_BYTE_ARRAY(outer, swapped);
1135                _cpri__UpdateHash(&h, 4, swapped);
1136                _cpri__UpdateHash(&h, 2, (BYTE *)&keySizeInBits);
1137                if(i < fill)
1138                    fill = i;
1139                _cpri__CompleteHash(&h, fill, pb);
1140
1141                // Restart the oPad hash
1142                _cpri__CopyHashState(&h, &h2);
1143
1144                // Add the last hashed data
1145                _cpri__UpdateHash(&h, fill, pb);
1146
1147                // gives a completed HMAC
1148                _cpri__CompleteHash(&h, fill, pb);
1149                pb += fill;
1150            }
1151            // Set the Most significant 2 bits and the low bit of the candidate
1152            p->buffer[0] |= 0xC0;
1153            p->buffer[p->size - 1] |= 1;
1154
1155            // Convert the candidate to a BN
1156            BN_bin2bn(p->buffer, p->size, bnP);
1157
1158            // If this is the second prime, make sure that it differs from the
1159            // first prime by at least 2^100
1160            if(!BN_is_zero(bnQ))
1161            {
1162                // bnQ is non-zero if we already found it
1163                if(BN_ucmp(bnP, bnQ) < 0)
1164                    BN_sub(bnT, bnQ, bnP);
1165                else
1166                    BN_sub(bnT, bnP, bnQ);
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1167            if(BN_num_bits(bnT) < 100)  <Q>// Difference has to be at least 100 bits
1168                continue;
1169        }
1170        // Make sure that the prime candidate (p) is not divisible by the exponent
1171        // and that (p-1) is not divisible by the exponent
1172        // Get the remainder after dividing by the modulus
1173        rem = BN_mod_word(bnP, e);
1174        if(rem == 0) // evenly divisible so add two keeping the number odd and
1175            // making sure that 1 != p mod e
1176            BN_add_word(bnP, 2);
1177        else if(rem == 1) // leaves a remainder of 1 so subtract two keeping the
1178            // number odd and making (e-1) = p mod e
1179            BN_sub_word(bnP, 2);
1180
1181        // Have a candidate, check for primality
1182        if((retVal = (CRYPT_RESULT)BN_is_prime_ex(bnP,
1183                    BN_prime_checks, NULL, NULL)) < 0)
1184            FAIL(FATAL_ERROR_INTERNAL);
1185
1186        if(retVal != 1)
1187            continue;
1188
1189        // Found a prime, is this the first or second.
1190        if(BN_is_zero(bnQ))
1191        {
1192            // copy p to q and compute another prime in p
1193            BN_copy(bnQ, bnP);
1194            continue;
1195        }
1196        //Form the public modulus
1197        BN_mul(bnN, bnP, bnQ, context);
1198        if(BN_num_bits(bnN) != keySizeInBits)
1199            FAIL(FATAL_ERROR_INTERNAL);
1200
1201        // Save the public modulus
1202        BnTo2B(n, bnN, n->size);  // Will pad the buffer to the correct size
1203        pAssert((n->buffer[0] & 0x80) != 0);
1204
1205        // And one prime
1206        BnTo2B(p, bnP, p->size);
1207        pAssert((p->buffer[0] & 0x80) != 0);
1208
1209        // Finish by making sure that we can form the modular inverse of PHI
1210        // with respect to the public exponent
1211        // Compute PHI = (p - 1)(q - 1) = n - p - q + 1
1212        // Make sure that we can form the modular inverse
1213        BN_sub(bnT, bnN, bnP);
1214        BN_sub(bnT, bnT, bnQ);
1215        BN_add_word(bnT, 1);
1216
1217        // find d such that (Phi * d) mod e ==1
1218        // if there isn't then we are broken because we took the step
1219        // of making sure that the prime != 1 mod e so the modular inverse
1220        // must exist
1221        if(BN_mod_inverse(bnT, bnE, bnT, context) == NULL || BN_is_zero(bnT))
1222            FAIL(FATAL_ERROR_INTERNAL);
1223
1224        // And, finally, do a trial encryption decryption
1225        {
1226            TPM2B_TYPE(RSA_KEY, MAX_RSA_KEY_BYTES);
1227            TPM2B_RSA_KEY        r;
1228            r.t.size = sizeof(n->size);
1229
1230            // If we are using a seed, then results must be reproducible on each
1231            // call. Otherwise, just get a random number
1232            if(seed == NULL)
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
1233                _cpri__GenerateRandom(n->size, r.t.buffer);
1234            else
1235            {
1236                // this this version does not have a deterministic RNG, XOR the
1237                // public key and private exponent to get a deterministic value
1238                // for testing.
1239                int         i;
1240
1241                // Generate a random-ish number starting with the public modulus
1242                // XORed with the MSO of the seed
1243                for(i = 0; i < n->size; i++)
1244                    r.t.buffer[i] = n->buffer[i] ^ seed->buffer[0];
1245            }
1246            // Make sure that the number is smaller than the public modulus
1247            r.t.buffer[0] &= 0x7F;
1248                  // Convert
1249            if(   BN_bin2bn(r.t.buffer, r.t.size, bnP) == NULL
1250                  // Encrypt with the public exponent
1251               || BN_mod_exp(bnQ, bnP, bnE, bnN, context) != 1
1252                  // Decrypt with the private exponent
1253               || BN_mod_exp(bnQ, bnQ, bnT, bnN, context) != 1)
1254                FAIL(FATAL_ERROR_INTERNAL);
1255            // If the starting and ending values are not the same, start over )-;
1256            if(BN_ucmp(bnP, bnQ) != 0)
1257            {
1258                BN_zero(bnQ);
1259                continue;
1260            }
1261        }
1262        retVal = CRYPT_SUCCESS;
1263        goto Cleanup;
1264    }
1265    retVal = CRYPT_FAIL;
1266
1267
1268 Cleanup:
1269    // Close out the hash sessions
1270    _cpri__CompleteHash(&h2, 0, NULL);
1271    _cpri__CompleteHash(&h1, 0, NULL);
1272
1273    // Free up allocated BN values
1274    BN_CTX_end(context);
1275    BN_CTX_free(context);
1276    if(counter != NULL)
1277        *counter = outer;
1278    return retVal;
1279 }
1280 #endif      // RSA_KEY_SIEVE
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

B.12.2. Alternative RSA Key Generation

B.12.2.1. Introduction

The files in Annex B.12.2 implement an alternative RSA key generation method that is about an order of magnitude faster than the regular method in B.12.1 and is provided simply to speed testing of the test functions. The method implemented in Annex B.12.2 uses a sieve rather than choosing prime candidates at random and testing for primeness. In this alternative, the sieve filed starting address is chosen at random and a sieve operation is performed on the field using small prime values. After sieving, the bits representing values that are not divisible by the small primes tested, will be checked in a pseudo-random order until a prime is found.

 © ISO/IEC 2015 – All rights reserved

The size of the sieve field is tunable as is the value indicating the number of primes that should be checked. As the size of the prime increases, the density of primes is reduced so the size of the sieve field should be increased to improve the probability that the field will contain at least one prime. In addition, as the sieve field increases the number of small primes that should be checked increases. Eliminating a number from consideration by using division is considerably faster than eliminating the number with a Miller-Rabin test.

B.12.2.2. RSAKeySieve.h

This header file is used to for parameterization of the Sieve and RNG used by the RSA module

```
#ifndef     RSA_H
#define     RSA_H
```

This value is used to set the size of the table that is searched by the prime iterator. This is used during the generation of different primes. The smaller tables are used when generating smaller primes.

```
extern const UINT16   primeTableBytes;
```

The following define determines how large the prime number difference table will be defined. The value of 13 will allocate the maximum size table which allows generation of the first 6542 primes which is all the primes less than 2^16.

```
#define PRIME_DIFF_TABLE_512_BYTE_PAGES     13
```

This set of macros used the value above to set the table size.

```
#ifndef PRIME_DIFF_TABLE_512_BYTE_PAGES
#   define PRIME_DIFF_TABLE_512_BYTE_PAGES   4
#endif
#ifdef PRIME_DIFF_TABLE_512_BYTE_PAGES
#   if PRIME_DIFF_TABLE_512_BYTE_PAGES > 12
#       define PRIME_DIFF_TABLE_BYTES 6542
#   else
#       if  PRIME_DIFF_TABLE_512_BYTE_PAGES <= 0
#           define PRIME_DIFF_TABLE_BYTES 512
#       else
#           define PRIME_DIFF_TABLE_BYTES (PRIME_DIFF_TABLE_512_BYTE_PAGES * 512)
#       endif
#   endif
#endif
extern const BYTE primeDiffTable [PRIME_DIFF_TABLE_BYTES];
```

This determines the number of bits in the sieve field This must be a power of two.

```
#define FIELD_POWER     14  // This is the only value in this group that should be
                            // changed
#define FIELD_BITS      (1 << FIELD_POWER)
#define MAX_FIELD_SIZE      ((FIELD_BITS / 8) + 1)
```

This is the pre-sieved table. It already has the bits for multiples of 3, 5, and 7 cleared.

```
#define SEED_VALUES_SIZE          105
const extern BYTE                 seedValues[SEED_VALUES_SIZE];
```

This allows determination of the number of bits that are set in a byte without having to count them individually.

```
const extern BYTE                 bitsInByte[256];
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

This is the iterator structure for accessing the compressed prime number table. The expectation is that values will need to be accesses sequentially. This tries to save some data access.

```
27  typedef struct {
28      UINT32      lastPrime;
29      UINT32      index;
30      UINT32      final;
31  } PRIME_ITERATOR;
32  #ifdef  RSA_INSTRUMENT
33  #   define INSTRUMENT_SET(a, b) ((a) = (b))
34  #   define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
35  #   define INSTRUMENT_INC(a)    (a) = (a) + 1
36  extern UINT32  failedAtIteration[10];
37  extern UINT32  MillerRabinTrials;
38  extern UINT32  totalFieldsSieved;
39  extern UINT32  emptyFieldsSieved;
40  extern UINT32  noPrimeFields;
41  extern UINT32  primesChecked;
42  extern UINT16   lastSievePrime;
43  #else
44  #   define INSTRUMENT_SET(a, b)
45  #   define INSTRUMENT_ADD(a, b)
46  #   define INSTRUMENT_INC(a)
47  #endif
48  #ifdef RSA_DEBUG
49  extern UINT16   defaultFieldSize;
50  #define NUM_PRIMES          2047
51  extern const __int16        primes[NUM_PRIMES];
52  #else
53  #define defaultFieldSize    MAX_FIELD_SIZE
54  #endif
55  #endif
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

B.12.2.3. RSAKeySieve.c

B.12.2.3.1. Includes and Defines

```
#include    "OsslCryptoEngine.h"
```

This file produces no code unless the compile switch is set to cause it to generate code.

```
#ifdef      RSA_KEY_SIEVE               //%
#include    "RsaKeySieve.h"
```

This next line will show up in the header file for this code. It will make the local functions public when debugging.

```
//%#ifdef   RSA_DEBUG
```

B.12.2.3.2. Bit Manipulation Functions

Introduction

These functions operate on a bit array. A bit array is an array of bytes with the 0th byte being the byte with the lowest memory address. Within the byte, bit 0 is the least significant bit.

ClearBit()

This function will CLEAR a bit in a bit array.

```
void
ClearBit(
    unsigned char   *a,             // IN: A pointer to an array of bytes
    int              i              // IN: the number of the bit to CLEAR
    )
{
    a[i >> 3] &= 0xff ^ (1 << (i & 7));
}
```

SetBit()

Function to SET a bit in a bit array.

```
void
SetBit(
    unsigned char   *a,             // IN: A pointer to an array of bytes
    int              i              // IN: the number of the bit to SET
    )
{
    a[i >> 3] |= (1 << (i & 7));
}
```

IsBitSet()

Function to test if a bit in a bit array is SET.

Table B.44

Return Value	Meaning
0	bit is CLEAR
1	bit is SET

```
UINT32
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

```
IsBitSet(
    unsigned char   *a,             // IN: A pointer to an array of bytes
    int              i              // IN: the number of the bit to test
    )
{
    return ((a[i >> 3] & (1 << (i & 7))) != 0);
}
```

BitsInArry()

This function counts the number of bits set in an array of bytes.

```
int
BitsInArray(
    unsigned char   *a,             // IN: A pointer to an array of bytes
    int              i              // IN: the number of bytes to sum
    )
{
    int     j = 0;
    for(; i ; i--)
        j += bitsInByte[*a++];
    return j;
}
```

FindNthSetBit()

This function finds the nth SET bit in a bit array. The caller should check that the offset of the returned value is not out of range. If called when the array does not have n bits set, it will return a fatal error

```
UINT32
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
    )
{
    UINT32       i;
    const BYTE  *pA = a;
    UINT32       retValue;
    BYTE         sel;

    (aSize);

    //find the bit
    for(i = 0; i < n; i += bitsInByte[*pA++]);

    // The chosen bit is in the byte that was just accessed
    // Compute the offset to the start of that byte
    pA--;
    retValue = (UINT32)(pA - a) * 8;

    // Subtract the bits in the last byte added.
    i -= bitsInByte[*pA];

    // Now process the byte, one bit at a time.
    for(sel = *pA; sel != 0 ; sel = sel >> 1)
    {
        if(sel & 1)
        {
            i += 1;
            if(i == n)
                return retValue;
        }
        retValue += 1;
    }
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
76      FAIL(FATAL_ERROR_INTERNAL);
77  }
```

B.12.2.3.3. Miscellaneous Functions

RandomForRsa()

This function uses a special form of KDFa() to produces a pseudo random sequence. It's input is a structure that contains pointers to a pre-computed set of hash contexts that are set up for the HMAC computations using the seed.

This function will test that ktx. outer will not wrap to zero if incremented. If so, the function returns FALSE. Otherwise, the ktx. outer is incremented before each number is generated.

```
78  void
79  RandomForRsa(
80      KDFa_CONTEXT    *ktx,           // IN: a context for the KDF
81      const char      *label,         // IN: a use qualifying label
82      TPM2B           *p              // OUT: the pseudo random result
83      )
84  {
85      INT16                   i;
86      UINT32                  inner;
87      BYTE                    swapped[4];
88      UINT16                  fill;
89      BYTE                    *pb;
90      UINT16                  lLen = 0;
91      UINT16                  digestSize = _cpri__GetDigestSize(ktx->hashAlg);
92      CPRI_HASH_STATE         h;      // the working hash context
93
94      if(label != NULL)
95          for(lLen = 0; label[lLen++];);
96      fill = digestSize;
97      pb = p->buffer;
98      inner = 0;
99      *(ktx->outer) += 1;
100     for(i = p->size; i > 0; i -= digestSize)
101     {
102         inner++;
103
104         // Initialize the HMAC with saved state
105         _cpri__CopyHashState(&h, &(ktx->iPadCtx));
106
107         // Hash the inner counter (the one that changes on each HMAC iteration)
108         UINT32_TO_BYTE_ARRAY(inner, swapped);
109         _cpri__UpdateHash(&h, 4, swapped);
110         if(lLen != 0)
111             _cpri__UpdateHash(&h, lLen, (BYTE *)label);
112
113         // Is there any party 1 data
114         if(ktx->extra != NULL)
115             _cpri__UpdateHash(&h, ktx->extra->size, ktx->extra->buffer);
116
117         // Include the outer counter (the one that changes on each prime
118         // prime candidate generation
119         UINT32_TO_BYTE_ARRAY(*(ktx->outer), swapped);
120         _cpri__UpdateHash(&h, 4, swapped);
121         _cpri__UpdateHash(&h, 2, (BYTE *)&ktx->keySizeInBits);
122         if(i < fill)
123             fill = i;
124         _cpri__CompleteHash(&h, fill, pb);
125
126         // Restart the oPad hash
127         _cpri__CopyHashState(&h, &(ktx->oPadCtx));
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

© ISO/IEC 2015 – All rights reserved

```
128
129         // Add the last hashed data
130         _cpri__UpdateHash(&h, fill, pb);
131
132         // gives a completed HMAC
133         _cpri__CompleteHash(&h, fill, pb);
134         pb += fill;
135     }
136     return;
137 }
```

MillerRabinRounds()

Function returns the number of MillerRabin() rounds necessary to give an error probability equal to the security strength of the prime. These values are from FIPS 186-3.

```
138 UINT32
139 MillerRabinRounds(
140     UINT32          bits           // IN: Number of bits in the RSA prime
141     )
142 {
143     if(bits < 511) <K>return 8;    // don't really expect this
144     if(bits < 1536) <K>return 5;   // for 512 and 1K primes
145     return 4;                   // for 3K public modulus and greater
146 }
```

MillerRabin()

This function performs a Miller-Rabin test from FIPS 186-3. It does *iterations* trials on the number. I all likelihood, if the number is not prime, the first test fails.

If a KDFa(), PRNG context is provide (ktx), then it is used to provide the random values. Otherwise, the random numbers are retrieved from the random number generator.

Table B.45

Return Value	Meaning
TRUE	probably prime
FALSE	composite

```
147 BOOL
148 MillerRabin(
149     BIGNUM          *bnW,
150     int             iterations,
151     KDFa_CONTEXT    *ktx,
152     BN_CTX          *context
153     )
154 {
155     BIGNUM      *bnWm1;
156     BIGNUM      *bnM;
157     BIGNUM      *bnB;
158     BIGNUM      *bnZ;
159     BOOL         ret = FALSE;   // Assumed composite for easy exit
160     TPM2B_TYPE(MAX_PRIME, MAX_RSA_KEY_BYTES/2);
161     TPM2B_MAX_PRIME   b;
162     int          a;
163     int          j;
164     int          wLen;
165     int          i;
166
167     pAssert(BN_is_bit_set(bnW, 0));
168     INSTRUMENT_INC(MillerRabinTrials);  // Instrumentation
169
```

IECNORM.COM : Click to view the full PDF of ISO/IEC 11889-4:2015

 © ISO/IEC 2015 – All rights reserved