
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*





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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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*

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.

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.

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.

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.

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**.

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**.

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 0th octet of the structure in which it is placed regardless of the size of the structure.

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

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

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

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

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

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

```
1  #ifndef _BASETYPES_H
2  #define _BASETYPES_H
3  #include "stdint.h"
```

NULL definition

```
4  #ifndef          NULL
5  #define          NULL      (0)
6  #endif
7  typedef uint8_t      UINT8;
8  typedef uint8_t      BYTE;
9  typedef int8_t       INT8;
10 typedef int          BOOL;
11 typedef uint16_t     UINT16;
12 typedef int16_t      INT16;
13 typedef uint32_t     UINT32;
14 typedef int32_t      INT32;
15 typedef uint64_t     UINT64;
16 typedef int64_t      INT64;
17 typedef struct {
18     UINT16          size;
19     BYTE            buffer[1];
20 } TPM2B;
21 #endif
```

6.3 bits.h

```
1  #ifndef    _BITS_H
2  #define    _BITS_H
3  #define    CLEAR_BIT(bit, vector)    BitClear((bit), (BYTE *)&(vector), sizeof(vector))
4  #define    SET_BIT(bit, vector)      BitSet((bit), (BYTE *)&(vector), sizeof(vector))
5                                     \
6  #define    TEST_BIT(bit, vector)     BitIsSet((bit), (BYTE *)&(vector), sizeof(vector))
7  #endif
```

6.4 bool.h

```

1  #ifndef _BOOL_H
2  #define _BOOL_H
3  #if defined(TRUE)
4  #undef TRUE
5  #endif
6  #if defined FALSE
7  #undef FALSE
8  #endif
9  typedef int BOOL;
10 #define FALSE ((BOOL)0)
11 #define TRUE ((BOOL)1)
12 #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().

```

1  #ifndef _CAPABILITIES_H
2  #define _CAPABILITIES_H
3  #define MAX_CAP_DATA (MAX_CAP_BUFFER - sizeof(TPM_CAP) - sizeof(UINT32))
4  #define MAX_CAP_ALGS (ALG_LAST_VALUE - ALG_FIRST_VALUE + 1)
5  #define MAX_CAP_HANDLES (MAX_CAP_DATA / sizeof(TPM_HANDLE))
6  #define MAX_CAP_CC ((TPM_CC_LAST - TPM_CC_FIRST) + 1)
7  #define MAX_TPM_PROPERTIES (MAX_CAP_DATA / sizeof(TPMS_TAGGED_PROPERTY))
8  #define MAX_PCR_PROPERTIES (MAX_CAP_DATA / sizeof(TPMS_TAGGED_PCR_SELECT))
9  #define MAX_ECC_CURVES (MAX_CAP_DATA / sizeof(TPM_ECC_CURVE))
10 #endif

```

6.6 TPMB.h

This file contains extra TPM2B structures

```

1  #ifndef _TPMB_H
2  #define _TPMB_H
3  #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.

```

4  #define TPM2B_TYPE(name, bytes) \
5      typedef union { \
6          struct { \
7              UINT16 size; \
8              BYTE buffer[(bytes)]; \
9          } t; \
10     TPM2B b; \
11 } TPM2B_##name

```

Macro to instance and initialize a TPM2B value

```

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

```

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

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

```

1  #ifndef      GLOBAL_H
2  #define      GLOBAL_H
3  // #define SELF_TEST
4  #include     "TpmBuildSwitches.h"
5  #include     "Tpm.h"
6  #include     "TPMB.h"
7  #include     "CryptoEngine.h"
8  #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.

```

9  #ifndef UNREFERENCED_PARAMETER
10 # define UNREFERENCED_PARAMETER(a) (a)
11 #endif
12 #include "bits.h"

```

6.8.3.2 Crypto Self-Test Values

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

```

13 #ifndef SELF_TEST
14 extern ALGORITHM_VECTOR g_implementedAlgorithms;
15 extern ALGORITHM_VECTOR g_toTest;
16 #else
17 LIB_IMPORT extern ALGORITHM_VECTOR g_implementedAlgorithms;
18 LIB_IMPORT extern ALGORITHM_VECTOR g_toTest;
19 #endif

```

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

```

20 #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.

```

21 #define TEST_HASH(alg) \
22     if( TEST_BIT(alg, g_toTest) \
23         && (alg != ALG_NULL_VALUE)) \
24         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

```

25 typedef BYTE HASH_STATE_TYPE;
26 #define HASH_STATE_EMPTY ((HASH_STATE_TYPE) 0)

```

```

27 #define HASH_STATE_HASH      ((HASH_STATE_TYPE) 1)
28 #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.

```

29 typedef struct
30 {
31     CPRI_HASH_STATE      state;           // hash state
32     HASH_STATE_TYPE      type;           // type of the context
33 } 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.

```

34 typedef struct
35 {
36     HASH_STATE            hashState;       // the hash state
37     TPM2B_HASH_BLOCK      hmacKey;        // the HMAC key
38 } HMAC_STATE;

```

6.8.3.4 Other Types

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

```

39 typedef BYTE    AUTH_VALUE[sizeof(TPMU_HA)];

```

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

```

40 typedef BYTE    TIME_INFO[sizeof(TPMS_TIME_INFO)];

```

A NAME is a BYTE array that can contain a TPMU_NAME

```

41 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.

```

42 typedef struct
43 {
44     unsigned      publicOnly    : 1;    //0) SET if only the public portion of
45                                     // an object is loaded
46     unsigned      epsHierarchy : 1;    //1) SET if the object belongs to EPS
47                                     // Hierarchy

```

```

48     unsigned          ppsHierarchy : 1; //2) SET if the object belongs to PPS
49                               // Hierarchy
50     unsigned          spsHierarchy : 1; //3) SET if 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;

```

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.

```

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

```

6.8.4.5 ANY_OBJECT

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

```

114  typedef union
115  {
116      OBJECT            entity;
117      HASH_OBJECT       hash;
118  } ANY_OBJECT;

```

6.8.5 AUTH_DUP Types

These values are used in the authorization processing.

```

119  typedef UINT32        AUTH_ROLE;
120  #define AUTH_NONE      ((AUTH_ROLE) (0))
121  #define AUTH_USER      ((AUTH_ROLE) (1))
122  #define AUTH_ADMIN     ((AUTH_ROLE) (2))
123  #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.

```

124  typedef struct

```

```

125 {
126     unsigned    isPolicy : 1;        //1) SET if the session may only
127                                     // be used for policy
128     unsigned    isAudit : 1;        //2) SET if the session is used
129                                     // for audit
130     unsigned    isBound : 1;        //3) SET if the session is bound to
131                                     // with an entity.
132                                     // This attribute will be CLEAR if
133                                     // either isPolicy or isAudit is SET.
134     unsigned    iscpHashDefined : 1; //4) SET if the cpHash has been defined
135                                     // This attribute is not SET unless
136                                     // 'isPolicy' is SET.
137     unsigned    isAuthValueNeeded : 1;
138                                     //5) SET if the authValue is required
139                                     // for computing the session HMAC.
140                                     // This attribute is not SET unless
141                                     // isPolicy is SET.
142     unsigned    isPasswordNeeded : 1;
143                                     //6) SET if a password authValue is
144                                     // required for authorization
145                                     // This attribute is not SET unless
146                                     // isPolicy is SET.
147     unsigned    isPPRequired : 1;   //7) SET if physical presence is
148                                     // required to be asserted when the
149                                     // authorization is checked.
150                                     // This attribute is not SET unless
151                                     // isPolicy is SET.
152     unsigned    isTrialPolicy : 1;  //8) SET if the policy session is
153                                     // created for trial of the policy's
154                                     // policyHash generation.
155                                     // This attribute is not SET unless
156                                     // isPolicy is SET.
157     unsigned    isDaBound : 1;      //9) SET if the bind entity had noDA
158                                     // CLEAR. If this is SET, then an
159                                     // auth failure using this session
160                                     // will count against lockout even
161                                     // if the object being authorized is
162                                     // exempt from DA.
163     unsigned    isLockoutBound : 1; //10) SET if the session is bound to
164                                     // lockoutAuth.
165     unsigned    requestWasBound : 1; //11) SET if the session is being used
166                                     // with the bind entity. If SET
167                                     // the authValue will not be use
168                                     // in the response HMAC computation.
169     unsigned    checkNvWritten : 1; //12) SET if the TPMA_NV_WRITTEN
170                                     // attribute needs to be checked
171                                     // when the policy is used for
172                                     // authorization for NV access.
173                                     // If this is SET for any other
174                                     // type, the policy will fail.
175     unsigned    nvWrittenState : 1; //13) SET if TPMA_NV_WRITTEN is
176                                     // required to be SET.
177 } 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.

```

178 typedef struct
179 {
180     TPM_ALG_ID    authHashAlg;        // session hash algorithm
181     TPM2B_NONCE    nonceTPM;          // last TPM-generated nonce for

```

```

182                                     // this session
183
184     TPMT_SYM_DEF      symmetric;      // session symmetric algorithm (if any)
185     TPM2B_AUTH        sessionKey;     // session secret value used for
186                                     // generating HMAC and encryption keys
187
188     SESSION_ATTRIBUTES attributes;     // session attributes
189     TPM_CC             commandCode;    // command code (policy session)
190     TPMA_LOCALITY      commandLocality; // command locality (policy session)
191     UINT32             pcrCounter;     // PCR counter value when PCR is
192                                     // included (policy session)
193                                     // If no PCR is included, this
194                                     // value is 0.
195
196     UINT64             startTime;      // value of TPMS_CLOCK_INFO.clock when
197                                     // the session was started (policy
198                                     // session)
199
200     UINT64             timeOut;        // timeout relative to
201                                     // TPMS_CLOCK_INFO.clock
202                                     // There is no timeout if this value
203                                     // is 0.
204     union
205     {
206         TPM2B_NAME      boundEntity;   // value used to track the entity to
207                                     // which the session is bound
208
209         TPM2B_DIGEST     cpHash;       // the required cpHash value for the
210                                     // command being authorized
211
212     } u1;                             // 'boundEntity' and 'cpHash' may
213                                     // share the same space to save memory
214
215     union
216     {
217         TPM2B_DIGEST     auditDigest;  // audit session digest
218         TPM2B_DIGEST     policyDigest; // policyHash
219
220     } u2;                             // audit log and policyHash may
221                                     // share space to save memory
222 } 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.

```

223 typedef struct
224 {
225     #ifdef TPM_ALG_SHA1
226         BYTE      sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
227     #endif
228     #ifdef TPM_ALG_SHA256
229         BYTE      sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
230     #endif
231     #ifdef TPM_ALG_SHA384
232         BYTE      sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
233     #endif
234     #ifdef TPM_ALG_SHA512

```

```

235     BYTE                sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
236 #endif
237 #ifndef 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;

```

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 method provides an imperfect analog to an actual NV implementation.

```

265 typedef enum
266 {
267 // Entries below mirror the PERSISTENT_DATA structure. These values are written
268 // to NV as individual items.
269 // hierarchy
270     NV_DISABLE_CLEAR,
271     NV_OWNER_ALG,
272     NV_ENDORSEMENT_ALG,
273     NV_LOCKOUT_ALG,
274     NV_OWNER_POLICY,
275     NV_ENDORSEMENT_POLICY,
276     NV_LOCKOUT_POLICY,
277     NV_OWNER_AUTH,
278     NV_ENDORSEMENT_AUTH,
279     NV_LOCKOUT_AUTH,
280
281     NV_EP_SEED,
282     NV_SP_SEED,
283     NV_PP_SEED,
284
285     NV_PH_PROOF,
286     NV_SH_PROOF,
287     NV_EH_PROOF,
288
289 // Time
290     NV_TOTAL_RESET_COUNT,
291     NV_RESET_COUNT,
292
293 // PCR
294     NV_PCR_POLICIES,
295     NV_PCR_ALLOCATED,
296
297 // Physical Presence
298     NV_PP_LIST,
299
300 // Dictionary Attack
301     NV_FAILED_TRIES,
302     NV_MAX_TRIES,
303     NV_RECOVERY_TIME,
304     NV_LOCKOUT_RECOVERY,
305     NV_LOCKOUT_AUTH_ENABLED,
306
307 // Orderly State flag
308     NV_ORDERLY,
309
310 // Command Audit
311     NV_AUDIT_COMMANDS,
312     NV_AUDIT_HASH_ALG,
313     NV_AUDIT_COUNTER,
314
315 // Algorithm Set
316     NV_ALGORITHM_SET,
317
318     NV_FIRMWARE_V1,
319     NV_FIRMWARE_V2,

```

```

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;

```

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`.

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 TPM 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;
```

```

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.

```

```

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 instantiated 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 yyyyymmdd
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().

```

495 typedef struct orderly_data
496 {
497
498 //*****
499 //          TIME
500 //*****
501
502 // Clock has two parts. One is the state save part and one is the NV part. The
503 // state save version is updated on each command. When the clock rolls over, the
504 // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
505 // orderly way, then the sClock value is used to initialize the clock. If the
506 // TPM shutdown was not orderly, then the persistent value is used and the safe
507 // attribute is clear.
508
509     UINT64          clock;           // The orderly version of clock
510     TPMI_YES_NO     clockSafe;      // Indicates if the clock value is
511                                     // safe.
512 //*****
513 //          DRBG
514 //*****
515 #ifndef _DRBG_STATE_SAVE
516     // This is DRBG state data. This is saved each time the value of clock is
517     // updated.
518     DRBG_STATE      drbgState;
519 #endif
520
521 } ORDERLY_DATA;
522 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).

```

523 typedef struct state_clear_data
524 {
525 //*****
526 //          Hierarchy Control
527 //*****
528     BOOL          shEnable;        // default reset is SET
529     BOOL          ehEnable;        // default reset is SET
530     BOOL          phEnableNV;      // default reset is SET
531     TPMI_ALG_HASH platformAlg;     // default reset is TPM_ALG_NULL
532     TPM2B_DIGEST  platformPolicy;  // default reset is an Empty Buffer
533     TPM2B_AUTH    platformAuth;    // default reset is an Empty Buffer
534
535 //*****
536 //          PCR
537 //*****
538 // The set of PCR to be saved on Shutdown(STATE)
539     PCR_SAVE      pcrSave;         // default reset is 0...0
540
541 // This structure hold the authorization values for those PCR that have an
542 // update authorization.
543 // This implementation only supports a single group of PCR controlled by
544 // authorization. If more are required, then this structure would be changed to
545 // an array.
546     PCR_AUTHVALUE pcrAuthValues;
547
548 }

```

```

549 } STATE_CLEAR_DATA;
550 extern STATE_CLEAR_DATA gc;

```

6.8.12.5 State Reset Data

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

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

```

551 typedef struct state_reset_data
552 {
553 //*****
554 //          Hierarchy Control
555 //*****
556     TPM2B_AUTH          nullProof;          // The proof value associated with
557                                           // the TPM_RH_NULL hierarchy. The
558                                           // default reset value is from the RNG.
559
560     TPM2B_SEED          nullSeed;           // The seed value for the TPM_RN_NULL
561                                           // hierarchy. The default reset value
562                                           // is from the RNG.
563
564 //*****
565 //          Context
566 //*****
567 // The 'clearCount' counter is incremented each time the TPM successfully executes
568 // a TPM Resume. The counter is included in each saved context that has 'stClear'
569 // SET (including descendants of keys that have 'stClear' SET). This prevents these
570 // objects from being loaded after a TPM Resume.
571 // If 'clearCount' at its maximum value when the TPM receives a Shutdown(STATE),
572 // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
573     UINT32              clearCount;         // The default reset value is 0.
574
575     UINT64              objectContextID;    // This is the context ID for a saved
576                                           // object context. The default reset
577                                           // value is 0.
578
579     CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];
580                                           // This is the value from which the
581                                           // 'contextID' is derived. The
582                                           // default reset value is {0}.
583
584
585     CONTEXT_COUNTER     contextCounter;     // This array contains contains the
586                                           // values used to track the version
587                                           // numbers of saved contexts (see
588                                           // Session.c in for details). The
589                                           // default reset value is 0.
590
591 //*****
592 //          Command Audit
593 //*****
594 // When an audited command completes, ExecuteCommand() checks the return
595 // value. If it is TPM_RC_SUCCESS, and the command is an audited command, the
596 // TPM will extend the cpHash and rpHash for the command to this value. If this
597 // digest was the Zero Digest before the cpHash was extended, the audit counter
598 // is incremented.
599
600     TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest
601                                           // by TPM2_GetCommandAuditDigest() or a
602                                           // TPM Reset.
603
604 //*****
605 //          Boot counter

```

```

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 is 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 //          ECDA
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

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;

```

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

```

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 {

```

```

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

```

6.10 Tpm.h

Root header file for building any TPM. lib code

```

1  #ifndef _TPM_H
2  #define _TPM_H
3  #include "bool.h"
4  #include "Implementation.h"
5  #include "TPM_Types.h"
6  #include "swap.h"
7  #endif

```

6.11 swap.h

```

1  #ifndef _SWAP_H
2  #define _SWAP_H
3  #include "Implementation.h"
4  #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 a UINT

```

5  #define BYTE_ARRAY_TO_UINT8(b) (UINT8)((b)[0])
6  #define BYTE_ARRAY_TO_UINT16(b) (UINT16)((b)[0] << 8) \
7      + (b)[1])
8  #define BYTE_ARRAY_TO_UINT32(b) (UINT32)((b)[0] << 24) \
9      + ((b)[1] << 16) \
10     + ((b)[2] << 8) \
11     + (b)[3])
12 #define BYTE_ARRAY_TO_UINT64(b) (UINT64)((UINT64)(b)[0] << 56) \
13     + ((UINT64)(b)[1] << 48) \
14     + ((UINT64)(b)[2] << 40) \
15     + ((UINT64)(b)[3] << 32) \
16     + ((UINT64)(b)[4] << 24) \
17     + ((UINT64)(b)[5] << 16) \
18     + ((UINT64)(b)[6] << 8) \
19     + (UINT64)(b)[7])

```

Disaggregate a UINT into a byte array

```

20 #define UINT8_TO_BYTE_ARRAY(i, b) ((b)[0] = (BYTE)(i), i)
21 #define UINT16_TO_BYTE_ARRAY(i, b) ((b)[0] = (BYTE)((i) >> 8), \
22     (b)[1] = (BYTE)(i), \
23     (i))
24 #define UINT32_TO_BYTE_ARRAY(i, b) ((b)[0] = (BYTE)((i) >> 24), \
25     (b)[1] = (BYTE)((i) >> 16), \
26     (b)[2] = (BYTE)((i) >> 8), \
27     (b)[3] = (BYTE)(i), \
28     (i))
29 #define UINT64_TO_BYTE_ARRAY(i, b) ((b)[0] = (BYTE)((i) >> 56), \
30     (b)[1] = (BYTE)((i) >> 48), \
31     (b)[2] = (BYTE)((i) >> 40), \
32     (b)[3] = (BYTE)((i) >> 32), \
33     (b)[4] = (BYTE)((i) >> 24), \
34     (b)[5] = (BYTE)((i) >> 16), \
35     (b)[6] = (BYTE)((i) >> 8), \
36     (b)[7] = (BYTE)(i), \
37     (i))
38 #else

```

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

```

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"

```