

# Trusted Platform Module Library

## Part 4: Supporting Routines

Family "02"

Level 00 Revision 00.96

March 15, 2013

Published

Contact: [admin@trustedcomputinggroup.org](mailto:admin@trustedcomputinggroup.org)

**Published**

Copyright © TCG 2006-2013

## Licenses and Notices

### 1. Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the "Source Code") a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

### 2. Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

### 3. Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration ([admin@trustedcomputinggroup.org](mailto:admin@trustedcomputinggroup.org)) for information on specification licensing rights available through TCG membership agreements.
- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.
- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

## CONTENTS

1	Scope .....	1
2	Terms and definitions .....	1
3	Symbols and abbreviated terms .....	1
4	Automation .....	1
4.1	Configuration Parser .....	1
4.2	Structure Parser .....	1
4.2.1	Introduction .....	1
4.2.2	Unmarshaling Code Prototype .....	2
4.2.2.1	Simple Types and Structures .....	2
4.2.2.2	Union Types .....	3
4.2.2.3	Null Types .....	3
4.2.2.4	Arrays.....	3
4.2.3	Marshaling Code Function Prototypes .....	3
4.2.3.1	Simple Types and Structures .....	3
4.2.3.2	Union Types .....	4
4.2.3.3	Arrays.....	4
4.3	Command Parser .....	4
4.4	Portability.....	5
5	Header Files .....	6
5.1	Introduction .....	6
5.2	bool.h.....	6
5.3	Capabilities.h .....	6
5.4	TPMB.h.....	6
5.5	TpmError.h.....	6
5.6	Global.h .....	7
5.7	swap.h .....	7
5.8	InternalRoutines.h .....	25
5.9	VendorString.h .....	26
6	Main .....	29
6.1	CommandDispatcher().....	29
6.2	ExecCommand.c .....	29
6.3	ParseHandleBuffer().....	29
6.4	SessionProcess.c.....	35
7	Command Support Functions.....	35
7.1	Introduction .....	69
7.2	Attestation Command Support .....	69
7.3	Context Management Command Support.....	69
7.4	Policy Command Support .....	72
7.5	NV Command Support.....	73
7.6	Object Command Support .....	75
8	Subsystem.....	102
8.1	CommandAudit.c .....	102
8.2	DA.c.....	102
8.3	Hierarchy.c.....	106
8.4	NV.c.....	109
8.5	Object.c.....	112
8.6	PCR.c .....	144
8.7	PP.c .....	159
8.8	Session.c .....	180

8.9	Time.c .....	182
9	Support .....	199
9.1	AlgorithmCap.c .....	204
9.2	Bits.c .....	204
9.3	CommandCodeAttributes_fp.h .....	206
9.4	Commands.c .....	208
9.5	DRTM.c .....	208
9.6	Entity.c .....	209
9.7	Global.c .....	210
9.8	Handle.c .....	216
9.9	Locality.c .....	218
9.10	Manufacture.c .....	219
9.11	Marshal.c .....	220
9.11.1	Introduction .....	222
9.11.2	Unmarshal and Marshal a Value .....	222
9.11.3	Unmarshal and Marshal a Union .....	223
9.11.4	Unmarshal and Marshal a Structure .....	225
9.11.5	Unmarshal and Marshal an Array .....	227
9.11.6	TPM2B Handling .....	229
9.12	MemoryLib.c .....	230
9.13	Power.c .....	230
9.14	PropertyCap.c .....	234
	Cryptographic Functions .....	235
9.15	Introduction .....	244
9.16	CryptUtil.c .....	244
9.17	Ticket.c .....	244
	Annex A (informative) Implementation Dependent .....	310
A.1	Introduction .....	311
A.2	Implementation.h .....	312
	Annex B (informative) Cryptographic Library Interface .....	320
B.1	Introduction .....	321
B.2	CryptoEngine.h .....	322
9.18	CryptPri.h .....	324
B.3	CpriData.c .....	328
B.4	MathFunctions.c .....	328
B.5	OpenSSLCryptPri.c .....	329
B.6	CpriRNG.c .....	339
B.7	CpriHash.c .....	342
B.8	CpriSym.c .....	345
B.9	RSA Files .....	360
B.9.1	CpriRSA.c .....	377
B.9.2	Alternative RSA Key Generation .....	401
B.9.2.1	Introduction .....	401
B.9.2.2	RSASieve.h .....	401
B.9.2.3	RSASieve.c .....	401
B.9.2.4	RSADData.c .....	402
B.10	Elliptic Curve Files .....	419
B.10.1	CpriDataEcc.h .....	431
B.10.2	CpriDataEcc.c .....	431
B.10.3	CpriECC.c .....	431
	Annex C (informative) Simulation Environment .....	472

C.1	Introduction .....	473
C.2	Cancel.c .....	474
C.3	Clock.c .....	474
C.4	Entropy.c .....	475
C.5	LocalityPlat.c .....	477
C.6	NVMem.c .....	478
C.7	PowerPlat.c .....	484
C.8	Platform.h .....	484
C.9	PlatformData.h .....	491
C.10	PlatformData.c .....	491
C.11	PPPlat.c .....	492
C.12	TpmFail.c .....	493
Annex D (informative) Remote Procedure Interface .....		495
D.1	Introduction .....	496
D.2	TpmTcpProtocol.h .....	497
D.3	TcpServer.c .....	497
D.4	TPMCmdp.c .....	499
D.5	TPMCmds.c .....	507



## Trusted Platform Module Library

### Part 4: Supporting Routines

#### 1 Scope

This document contains C code that describes the algorithms and methods used by the command code in part 3. The code in this document augments Parts 2 and 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 document may be replaced by code that provides similar results when interfacing to the action code in part 3. The behavior of code in this document that is not included in an annex is *normative*, as observed at the interfaces with part 3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of the specification from the provided code.

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

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

#### 2 Terms and definitions

For the purposes of this document, the terms and definitions given in part 1 of this specification apply.

#### 3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in part 1 apply.

#### 4 Automation

##### 4.1 Configuration Parser

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

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

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

##### 4.2 Structure Parser

###### 4.2.1 Introduction

The program that processes the tables in part 2 (other than the table in the annexes) is called "The part 2 Structure Parser."

NOTE                      A Perl script was used to parse the tables in part 2 to produce the header files and unmarshaling code in for the reference implementation.

The part 2 Structure Parser takes as input the files produced by the part 2 Configuration Parser and the same part 2 specification that was used as input to the part 2 Configuration Parser. The part 2 Structure Parser will generate all of the C structure constant definitions that are required by the TPM interface. Additionally, the parser will generate unmarshaling code for all structures passed to the TPM, and marshaling code for structures passed from the TPM.

The unmarshaling code produced by the TCG provided 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.

An implementer may substitute their own marshaling and unmarshaling code in place of the code produced by the TCG-provide tool. However, it is required that equivalent errors in the input data produce the equivalent response codes.

The sections below describe the function prototypes for the marshaling and unmarshaling code that is automatically generated by the TCG-provided part 2 Structure Parser. These prototypes are described here as the unmarshaling and marshaling of various types occurs in places other than when the command is being parsed or the response is being built. The prototypes and the description of the interface are intended to aid in the comprehension of the code that uses these auto-generated routines.

## 4.2.2 Unmarshaling Code Prototype

### 4.2.2.1 Simple Types and Structures

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

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

Where:

<b>TYPE</b>	name of the data type or structure
<b>*target</b>	location in the TPM memory into which the data from <b>**buffer</b> is placed
<b>**buffer</b>	location in input buffer containing the most significant octet (MSO) of <b>*target</b>
<b>*size</b>	number of octets remaining in <b>**buffer</b>

When the data is successfully unmarshaled, the called routine will return TPM\_RC\_SUCCESS. Otherwise, it will return a Format-One response code (see part 2).

If the data is successfully unmarshaled, **\*buffer** is advanced point to the first octet of the next parameter in the input buffer and **size** is reduced by the number of octets removed from the buffer.

When the data type is a simple type, the parser will generate code that will unmarshal the underlying type and then perform checks on the type as indicated by the type definition.

When the data type is a structure, the parser will generate code that unmarshals each of the structure elements in turn and performs any additional parameter checks as indicated by the data type.



### 4.2.2.2 Union Types

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

The function prototype for a union has the form:

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

where:

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

### 4.2.2.3 Null Types

In some cases, the structure definition allows an optional “null” value. The “null” value allows the use of the same C type for the entity even though it does not always have the same members.

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

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

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

### 4.2.2.4 Arrays

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

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

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

## 4.2.3 Marshaling Code Function Prototypes

### 4.2.3.1 Simple Types and Structures

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

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

Where:

<b>TYPE</b>	name of the data type or structure
<b>*source</b>	location in the TPM memory containing the value that is to be marshaled in to the designated buffer
<b>**buffer</b>	location in the output buffer where the first octet of the <b>TYPE</b> is to be placed
<b>*size</b>	number of octets remaining in <b>**buffer</b> . If <b>size</b> 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.

#### 4.2.3.2 Union Types

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

The function prototype for a union has the form:

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

The parameters have a similar meaning as those in 4.2.2.2 but the data movement is from **source** to **buffer**.

#### 4.2.3.3 Arrays

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

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

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

### 4.3 Command Parser

The program that processes the tables in part 3 is called "The part 3 Command Parser."

The part 3 Command Parser takes as input a part 3 of the TPM specification and some configuration files produced by the part 2 Configuration Parser. This parser uses the contents of the command and response tables in part 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 4.

#### 4.4 Portability

Where reasonable, the code is written to be portable. There are a few known cases where the code is not portable. Specifically, the handling of bit fields will not always be portable. The bit fields are marshaled and unmarshaled as a simple element of the underlying type. For example, a `TPMA_SESSION` is defined as a bit field in an octet (`BYTE`). When sent on the interface a `TPMA_SESSION` will occupy one octet. When unmarshaled, it is unmarshaled as a `UINT8`. The ramifications of this are that a `TPMA_SESSION` will occupy the 0<sup>th</sup> 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<sup>th</sup> octet always contains the 0<sup>th</sup> bit of the structure no matter how large the structure. However, for a big endian machine, the 0<sup>th</sup> 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<sup>th</sup> octet of the `TPMA_SESSION`. Since the 0<sup>th</sup> octet is most significant octet, this has the effect of shifting all the session attribute bits left by 24 places.

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

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

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

## 5 Header Files

### 5.1 Introduction

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

### 5.2 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

```

### 5.3 Capabilities.h

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

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

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

```

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

```

### 5.4 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

```

## 5.5 TpmError.h

```

1 #ifndef _TPM_ERROR_H
2 #define _TPM_ERROR_H
3 #define FATAL_ERROR_ALLOCATION (1)
4 #define FATAL_ERROR_DIVIDE_ZERO (2)
5 #define FATAL_ERROR_INTERNAL (3)
6 #define FATAL_ERROR_PARAMETER (4)

```

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

```

7 int _plat_TpmFail(const char *function, int line, int code);
8 #define pAssert(a) (!!(a) || _plat_TpmFail(__FUNCTION__, \
9     __LINE__, \
10    FATAL_ERROR_PARAMETER))
11 #define FAIL(a) (_plat_TpmFail(__FUNCTION__, __LINE__, a))
12 #endif

```

## 5.6 Global.h

### 5.6.1 Description

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

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

## 5.6.2 Includes

```

1  #ifndef      GLOBAL_H
2  #define      GLOBAL_H
3  #include     "Tpm.h"
4  #include     "TPMB.h"
5  #include     "CryptPri.h"

```

## 5.6.3 Defines

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

```

6  typedef BYTE    HASH_STATE_TYPE;
7  #define HASH_STATE_EMPTY    ((HASH_STATE_TYPE) 0)
8  #define HASH_STATE_HASH     ((HASH_STATE_TYPE) 1)
9  #define HASH_STATE_HMAC     ((HASH_STATE_TYPE) 2)

```

## 5.6.4 Hash State Structures

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.

```

10 typedef struct
11 {
12     HASH_STATE_TYPE    type;           // type of the context
13     CPRI_HASH_STATE    state;         // hash state
14 } 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.

```

15 typedef struct
16 {
17     HASH_STATE    hashState;         // the hash state
18     TPM2B_HASH_BLOCK    hmacKey;    // the HMAC key
19 } HMAC_STATE;

```

## 5.6.5 Loaded Object Structures

### 5.6.5.1 Description

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

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

### 5.6.5.2 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.

```

20 typedef struct
21 {

```

```

22     unsigned      publicOnly   : 1;    //0) SET if only the public portion of
23                                     // an object is loaded
24     unsigned      epsHierarchy : 1;    //1) SET if the object belongs to EPS
25                                     // Hierarchy
26     unsigned      ppsHierarchy : 1;    //2) SET if the object belongs to PPS
27                                     // Hierarchy
28     unsigned      spsHierarchy : 1;    //3) SET if the object belongs to SPS
29                                     // Hierarchy
30     unsigned      evict        : 1;    //4) SET if the object is a platform or
31                                     // owner evict object. Platform-
32                                     // evict object belongs to PPS
33                                     // hierarchy, owner-evict object
34                                     // belongs to SPS or EPS hierarchy.
35                                     // This bit is also used to mark a
36                                     // completed sequence object so it
37                                     // will be flush when the
38                                     // SequenceComplete command succeeds.
39     unsigned      primary      : 1;    //5) SET for a primary object
40     unsigned      temporary    : 1;    //6) SET for a temporary object
41     unsigned      stClear      : 1;    //7) SET for an stClear object
42     unsigned      hmacSeq      : 1;    //8) SET for an HMAC sequence object
43     unsigned      hashSeq      : 1;    //9) SET for a hash sequence object
44     unsigned      eventSeq     : 1;    //10) SET for an event sequence object
45     unsigned      ticketSafe   : 1;    //11) SET if a ticket is safe to create
46                                     // for hash sequence object
47     unsigned      firstBlock   : 1;    //12) SET if the first block of hash
48                                     // data has been received. It
49                                     // works with ticketSafe bit
50     unsigned      isParent     : 1;    //13) SET if the key has the proper
51                                     // attributes to be a parent key
52     unsigned      privateExp   : 1;    //14) SET when the private exponent
53                                     // of an RSA key has been validated.
54     unsigned      reserved     : 1;    //15) reserved bits. unused.
55 } OBJECT_ATTRIBUTES;

```

### 5.6.5.3 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.

```

56 typedef struct
57 {
58     // The attributes field is required to be first followed by the publicArea.
59     // This allows the overlay of the object structure and a sequence structure
60     OBJECT_ATTRIBUTES  attributes;    // object attributes
61     TPMT_PUBLIC        publicArea;    // public area of an object
62     TPMT_SENSITIVE     sensitive;     // sensitive area of an object
63
64 #ifndef TPM_ALG_RSA
65     TPM2B_PUBLIC_KEY_RSA privateExponent; // Additional field for the private
66                                     // exponent of an RSA key.
67 #endif
68     TPM2B_NAME          qualifiedName; // object qualified name
69     TPMT_DH_OBJECT      evictHandle;   // if the object is an evict object,
70                                     // the original handle is kept here.
71                                     // The 'working' handle will be the
72                                     // handle of an object slot.
73
74     TPM2B_NAME          name;         // Name of the object name. Kept here
75                                     // to avoid repeatedly computing it.
76 } OBJECT;

```

### 5.6.5.4 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.

```

77 typedef struct
78 {
79     OBJECT_ATTRIBUTES    attributes;           // The attributes of the HASH object
80     TPMI_ALG_PUBLIC      type;                // algorithm
81     TPMI_ALG_HASH        nameAlg;            // name algorithm
82     TPMA_OBJECT          objectAttributes;    // object attributes
83
84     // The data below is unique to a sequence object
85     TPM2B_AUTH           auth;                // auth for use of sequence
86     union
87     {
88         HASH_STATE       hashState[HASH_COUNT];
89         HMAC_STATE       hmacState;
90         state;
91     } HASH_OBJECT;

```

### 5.6.5.5 ANY\_OBJECT

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

```

92 typedef union
93 {
94     OBJECT            entity;
95     HASH_OBJECT       hash;
96 } ANY_OBJECT;

```

### 5.6.6 AUTH\_DUP Types

These values are used in the authorization processing.

```

97 typedef UINT32        AUTH_ROLE;
98 #define AUTH_NONE      ((AUTH_ROLE) (0))
99 #define AUTH_USER      ((AUTH_ROLE) (1))
100 #define AUTH_ADMIN     ((AUTH_ROLE) (2))
101 #define AUTH_DUP       ((AUTH_ROLE) (3))

```

### 5.6.7 Active Session Context

#### 5.6.7.1 Description

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

#### 5.6.7.2 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.

```

102 typedef struct

```



```

103 {
104     unsigned    isPolicy : 1;        //1) SET if the session may only
105                                     // be used for policy
106     unsigned    isAudit : 1;        //2) SET if the session is used
107                                     // for audit
108     unsigned    isBound : 1;        //3) SET if the session is bound to
109                                     // with an entity.
110                                     // This attribute will be CLEAR if
111                                     // either isPolicy or isAudit is SET.
112     unsigned    iscpHashDefined : 1; //4) SET if the cpHash has been defined
113                                     // This attribute is not SET unless
114                                     // 'isPolicy' is SET.
115     unsigned    isAuthValueNeeded : 1;
116                                     //5) SET if the authValue is required
117                                     // for computing the session HMAC.
118                                     // This attribute is not SET unless
119                                     // isPolicy is SET.
120     unsigned    isPasswordNeeded : 1;
121                                     //6) SET if a password authValue is
122                                     // required for authorization
123                                     // This attribute is not SET unless
124                                     // isPolicy is SET.
125     unsigned    isPPRequired : 1;   //7) SET if physical presence is
126                                     // required to be asserted when the
127                                     // authorization is checked.
128                                     // This attribute is not SET unless
129                                     // isPolicy is SET.
130     unsigned    isTrialPolicy : 1;  //8) SET if the policy session is
131                                     // created for trial of the policy's
132                                     // policyHash generation.
133                                     // This attribute is not SET unless
134                                     // isPolicy is SET.
135     unsigned    isDaBound : 1;      //9) SET if the bind entity had noDA
136                                     // CLEAR. If this is SET, then an
137                                     // auth failure using this session
138                                     // will count against lockout even
139                                     // if the object being authorized is
140                                     // exempt from DA.
141     unsigned    isLockoutBound : 1; //10) SET if the session is bound to
142                                     // lockoutAuth.
143 } SESSION_ATTRIBUTES;

```

### 5.6.7.3 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.

```

144 typedef struct
145 {
146     TPM_ALG_ID    authHashAlg;        // session hash algorithm
147     TPM2B_NONCE   nonceTPM;          // last TPM-generated nonce for
148                                     // this session
149
150     TPMT_SYM_DEF  symmetric;         // session symmetric algorithm (if any)
151     TPM2B_AUTH    sessionKey;        // session secret value used for
152                                     // generating HMAC and encryption keys
153
154     SESSION_ATTRIBUTES attributes;    // session attributes
155     TPM_CC        commandCode;        // command code (policy session)
156     TPMA_LOCALITY commandLocality;    // command locality (policy session)
157     UINT32        pcrCounter;         // PCR counter value when PCR is
158                                     // included (policy session)
159                                     // If no PCR is included, this

```

```

160                                     // value is 0.
161
162     UINT64        startTime;          // value of TPMS_CLOCK_INFO.clock when
163                                     // the session was started (policy
164                                     // session)
165
166     UINT64        timeOut;            // timeout relative to
167                                     // TPMS_CLOCK_INFO.clock
168                                     // There is no timeout if this value
169                                     // is 0.
170     union
171     {
172         TPM2B_NAME    boundEntity;    // value used to track the entity to
173                                     // which the session is bound
174
175         TPM2B_DIGEST  cpHash;         // the required cpHash value for the
176                                     // command being authorized
177
178     } u1;                             // 'boundEntity' and 'cpHash' may
179                                     // share the same space to save memory
180
181     union
182     {
183         TPM2B_DIGEST  auditDigest;    // audit session digest
184         TPM2B_DIGEST  policyDigest;   // policyHash
185
186     } u2;                             // audit log and policyHash may
187                                     // share space to save memory
188 } SESSION;

```

## 5.6.8 PCR

### 5.6.8.1 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.

```

189 typedef struct
190 {
191     #ifdef TPM_ALG_SHA1
192         BYTE        sha1[ NUM_STATIC_PCR ][ SHA1_DIGEST_SIZE ];
193     #endif
194     #ifdef TPM_ALG_SHA256
195         BYTE        sha256[ NUM_STATIC_PCR ][ SHA256_DIGEST_SIZE ];
196     #endif
197     #ifdef TPM_ALG_SHA384
198         BYTE        sha384[ NUM_STATIC_PCR ][ SHA384_DIGEST_SIZE ];
199     #endif
200     #ifdef TPM_ALG_SHA512
201         BYTE        sha512[ NUM_STATIC_PCR ][ SHA512_DIGEST_SIZE ];
202     #endif
203     #ifdef TPM_ALG_SM3_256
204         BYTE        sm3_256[ NUM_STATIC_PCR ][ SM3_256_DIGEST_SIZE ];
205     #endif
206
207     // This counter increments whenever the PCR are updated.
208     // NOTE: A platform-specific specification may designate
209     //       certain PCR changes as not causing this counter
210     //       to increment.
211     UINT32        pcrCounter;
212 }

```

```
213 } PCR_SAVE;
```

### 5.6.8.2 PCR\_POLICY

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

```
214 typedef struct
215 {
216     TPMI_ALG_HASH      hashAlg[NUM_POLICY_PCR_GROUP];
217     TPM2B_DIGEST      a;
218     TPM2B_DIGEST      policy[NUM_POLICY_PCR_GROUP];
219 } PCR_POLICY;
```

### 5.6.8.3 PCR\_AUTHVALUE

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

```
220 typedef struct
221 {
222     TPM2B_DIGEST      auth[NUM_AUTHVALUE_PCR_GROUP];
223 } PCR_AUTHVALUE;
```

## 5.6.9 Startup

### 5.6.9.1 SHUTDOWN\_NONE

Part 2 defines the two shutdown/startup types that may be used in TPM2\_Shutdown() and TPM2\_Startup(). This additional define is used by the TPM to indicate that no shutdown was received.

NOTE: This is a reserved value.

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

### 5.6.9.2 STARTUP\_TYPE

This enumeration is the possible startup types. The type is determined by the combination of TPM2\_ShutDown() and TPM2\_Startup().

```
225 typedef enum
226 {
227     SU_RESET,
228     SU_RESTART,
229     SU_RESUME
230 } STARTUP_TYPE;
```

## 5.6.10 NV

### 5.6.10.1 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.

```
231 typedef enum
232 {
```

```

233 // Entries below mirror the PERSISTENT_DATA structure. These values are written
234 // to NV as individual items.
235 // hierarchy
236 NV_DISABLE_CLEAR,
237 NV_OWNER_ALG,
238 NV_ENDORSEMENT_ALG,
239 NV_OWNER_POLICY,
240 NV_ENDORSEMENT_POLICY,
241 NV_OWNER_AUTH,
242 NV_ENDORSEMENT_AUTH,
243 NV_LOCKOUT_AUTH,
244
245 NV_EP_SEED,
246 NV_SP_SEED,
247 NV_PP_SEED,
248
249 NV_PH_PROOF,
250 NV_SH_PROOF,
251 NV_EH_PROOF,
252
253 // Time
254 NV_TOTAL_RESET_COUNT,
255 NV_RESET_COUNT,
256
257 // PCR
258 NV_PCR_POLICIES,
259 NV_PCR_ALLOCATED,
260
261 // Physical Presence
262 NV_PP_LIST,
263
264 // Dictionary Attack
265 NV_FAILED_TRIES,
266 NV_MAX_TRIES,
267 NV_RECOVERY_TIME,
268 NV_LOCKOUT_RECOVERY,
269 NV_LOCKOUT_AUTH_ENABLED,
270
271 // Orderly State flag
272 NV_ORDERLY,
273
274 // Command Audit
275 NV_AUDIT_COMMANDS,
276 NV_AUDIT_HASH_ALG,
277 NV_AUDIT_COUNTER,
278
279 // Algorithm Set
280 NV_ALGORITHM_SET,
281
282 NV_FIRMWARE_V1,
283 NV_FIRMWARE_V2,
284
285 // The entries above are in PERSISTENT_DATA. The entries below represent
286 // structures that are read and written as a unit.
287
288 // ORDERLY_DATA data structure written on each orderly shutdown
289 NV_CLOCK,
290
291 // STATE_CLEAR_DATA structure written on each Shutdown(STATE)
292 NV_STATE_CLEAR,
293
294 // STATE_RESET_DATA structure written on each Shutdown(STATE)
295 NV_STATE_RESET,
296
297 NV_RESERVE_LAST // end of NV reserved data list
298 } NV_RESERVE;

```

### 5.6.10.2 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.

```

299 typedef struct
300 {
301     TPMS_NV_PUBLIC      publicArea;
302     TPM2B_AUTH          authValue;
303 } NV_INDEX;

```

### 5.6.11 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.

```

304 #ifndef TPM_ALG_ECC
305 #define COMMIT_INDEX_MASK ((UINT16)((sizeof(gr.commitArray)*8)-1))
306 #endif

```

### 5.6.12 RAM Global Values

#### 5.6.12.1 Description

The values in this section are only extant in RAM. They are defined here and instanced in Global.c.

#### 5.6.12.2 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.

```

307 extern const UINT16    g_rcIndex[15];

```

#### 5.6.12.3 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.

```

308 extern TPM_HANDLE      g_exclusiveAuditSession;

```

#### 5.6.12.4 g\_time

This value is the count of milliseconds since the TPM was powered up. This value is initialized at *\_TPM\_Init()*.

```

309 extern UINT64          g_time;

```

#### 5.6.12.5 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.

```

310 extern BOOL            g_phEnable;

```

### 5.6.12.6 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).

```
311 extern BOOL g_pcrReConfig;
```

### 5.6.12.7 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().

```
312 extern TPMI_DH_OBJECT g_DRTMHandle;
```

### 5.6.12.8 g\_DrtmPreStartup

This value indicates that an H-CRTM occurred after \_TPM\_Init() but before TPM2\_Startup()

```
313 extern BOOL g_DrtmPreStartup;
```

### 5.6.12.9 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.

```
314 extern BOOL g_updateNV;
```

### 5.6.12.10 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.

```
315 extern BOOL g_clearOrderly;
```

### 5.6.12.11 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.

```
316 extern TPM_SU g_prevOrderlyState;
```

## 5.6.13 Persistent Global Values

### 5.6.13.1 Description

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

### 5.6.13.2 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()).

```

317 typedef struct
318 {
319 //*****
320 //          Hierarchy
321 //*****
322 // The values in this section are related to the hierarchies.
323
324     BOOL                disableClear;        // TRUE if TPM2_Clear() using
325                                                // lockoutAuth is disabled
326
327     // Hierarchy authPolicies
328     TPML_ALG_HASH       ownerAlg;
329     TPML_ALG_HASH       endorsementAlg;
330     TPML2B_DIGEST       ownerPolicy;
331     TPML2B_DIGEST       endorsementPolicy;
332
333     // Hierarchy authValues
334     TPM2B_AUTH          ownerAuth;
335     TPM2B_AUTH          endorsementAuth;
336     TPM2B_AUTH          lockoutAuth;
337
338     // Primary Seeds
339     TPM2B_SEED          EPSeed;
340     TPM2B_SEED          SPSeed;
341     TPM2B_SEED          PPSeed;
342     // Note there is a nullSeed in the state_reset memory.
343
344     // Hierarchy proofs
345     TPM2B_AUTH          phProof;
346     TPM2B_AUTH          shProof;
347     TPM2B_AUTH          ehProof;
348     // Note there is a nullProof in the state_reset memory.
349
350 //*****
351 //          Reset Events
352 //*****
353 // A count that increments at each TPM reset and never get reset during the life
354 // time of TPM. The value of this counter is initialized to 1 during TPM
355 // manufacture process.
356     UINT64              totalResetCount;
357
358 // This counter increments on each TPM Reset. The counter is reset by
359 // TPM2_Clear().
360     UINT32              resetCount;
361
362
363 //*****
364 //          PCR
365 //*****
366 // This structure hold the policies for those PCR that have an update policy.
367 // This implementation only supports a single group of PCR controlled by
368 // policy. If more are required, then this structure would be changed to
369 // an array.
370     PCR_POLICY          pcrPolicies;
371
372 // This structure indicates the allocation of PCR. The structure contains a
373 // list of PCR allocations for each implemented algorithm. If no PCR are
374 // allocated for an algorithm, a list entry still exists but the bit map
375 // will contain no SET bits.
376     TPML_PCR_SELECTION pcrAllocated;

```

```

377
378 //*****
379 //           Physical Presence
380 //*****
381 // The PP_LIST type contains a bit map of the commands that require physical
382 // to be asserted when the authorization is evaluated. Physical presence will be
383 // checked if the corresponding bit in the array is SET and if the authorization
384 // handle is TPM_RH_PLATFORM.
385 //
386 // These bits may be changed with TPM2_PP_Commands().
387     BYTE                ppList[((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7)/8];
388
389 //*****
390 //           Dictionary attack values
391 //*****
392 // These values are used for dictionary attack tracking and control.
393     UINT32              failedTries;           // the current count of unexpired
394                                           // authorization failures
395
396     UINT32              maxTries;             // number of unexpired authorization
397                                           // failures before the TPM is in
398                                           // lockout
399
400     UINT32              recoveryTime;         // time between authorization failures
401                                           // before failedTries is decremented
402
403     UINT32              lockoutRecovery;      // time that must expire between
404                                           // authorization failures associated
405                                           // with lockoutAuth
406
407     BOOL                lockOutAuthEnabled;  // TRUE if use of lockoutAuth is
408                                           // allowed
409
410 //*****
411 //           Orderly State
412 //*****
413 // The orderly state for current cycle
414     TPM_SU              orderlyState;
415
416 //*****
417 //           Command audit values.
418 //*****
419     BYTE                auditComands[((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8];
420     TPMI_ALG_HASH       auditHashAlg;
421     UINT64              auditCounter;
422
423 //*****
424 //           Algorithm selection
425 //*****
426 //
427 // The 'algorithmSet' value indicates the collection of algorithms that are
428 // currently in used on the TPM. The interpretation of value is vendor dependent.
429     UINT32              algorithmSet;
430
431 //*****
432 //           Firmware version
433 //*****
434 // The firmwareV1 and firmwareV2 values are instantiated in TimeStamp.c. This is
435 // a scheme used in development to allow determination of the linker build time
436 // of the TPM. An actual implementation would implement these values in a way that
437 // is consistent with vendor needs. The values are maintained in RAM for simplified
438 // access with a master version in NV. These values are modified in a
439 // vendor-specific way.
440
441 // g_firmwareV1 contains the more significant 32-bits of the vendor version number.
442 // In the reference implementation, if this value is printed as a hex

```



```

443 // value, it will have the format of yyyyymmdd
444     UINT32             firmwareV1;
445
446 // g_firmwareV1 contains the less significant 32-bits of the vendor version number.
447 // In the reference implementation, if this value is printed as a hex
448 // value, it will have the format of 00 hh mm ss
449     UINT32             firmwareV2;
450
451 } PERSISTENT_DATA;
452 extern PERSISTENT_DATA gp;

```

### 5.6.13.3 ORDERLY\_DATA

The data in this structure is saved to NV on each TPM2\_Shutdown().

```

453 typedef struct orderly_data
454 {
455 //*****
456 //           TIME
457 //*****
458 // Clock has two parts. One is the state save part and one is the NV part. The
459 // state save version is updated on each command. When the clock rolls over, the
460 // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
461 // orderly way, then the sClock value is used to initialize the clock. If the
462 // TPM shutdown was not orderly, then the persistent value is used and the safe
463 // attribute is clear.
464
465     UINT64             clock;           // The orderly version of clock
466     TPMI_YES_NO       clockSafe;      // Indicates if the clock value is
467                                     // safe.
468 } ORDERLY_DATA;
469 extern ORDERLY_DATA go;

```

### 5.6.13.4 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).

```

472 typedef struct state_clear_data
473 {
474 //*****
475 //           Hierarchy Control
476 //*****
477     BOOL              shEnable;       // default reset is SET
478     BOOL              ehEnable;       // default reset is SET
479     TPMI_ALG_HASH     platformAlg;    // default reset is TPM_ALG_NULL
480     TPM2B_DIGEST      platformPolicy; // default reset is an Empty Buffer
481     TPM2B_AUTH        platformAuth;   // default reset is an Empty Buffer
482
483 //*****
484 //           PCR
485 //*****
486 // The set of PCR to be saved on Shutdown(STATE)
487     PCR_SAVE         pcrSave;         // default reset is 0...0
488
489 // This structure hold the authorization values for those PCR that have an
490 // update authorization.

```

```

491 // This implementation only supports a single group of PCR controlled by
492 // authorization. If more are required, then this structure would be changed to
493 // an array.
494     PCR_AUTHVALUE          pcrAuthValues;
495
496 } STATE_CLEAR_DATA;
497 extern STATE_CLEAR_DATA gc;

```

### 5.6.13.5 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.

```

498 typedef struct state_reset_data
499 {
500 //*****
501 //          Hierarchy Control
502 //*****
503     TPM2B_AUTH          nullProof;          // The proof value associated with
504                                     // the TPM_RH_NULL hierarchy. The
505                                     // default reset value is from the RNG.
506
507     TPM2B_SEED          nullSeed;          // The seed value for the TPM_RN_NULL
508                                     // hierarchy. The default reset value
509                                     // is from the RNG.
510
511 //*****
512 //          Context
513 //*****
514 // The 'clearCount' counter is incremented each time the TPM successfully executes
515 // a TPM Resume. The counter is included in each saved context that has 'stClear'
516 // SET (including descendants of keys that have 'stClear' SET). This prevents these
517 // objects from being loaded after a TPM Resume.
518 // If 'clearCount' at its maximum value when the TPM receives a Shutdown(STATE),
519 // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
520     UINT32              clearCount;        // The default reset value is 0.
521
522     UINT64              objectContextID;   // This is the context ID for a saved
523                                     // object context. The default reset
524                                     // value is 0.
525
526     CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];
527                                     // This is the value from which the
528                                     // 'contextID' is derived. The
529                                     // default reset value is {0}.
530
531
532     CONTEXT_COUNTER     contextCounter;    // This array contains contains the
533                                     // values used to track the version
534                                     // numbers of saved contexts (see
535                                     // Session.c in for details). The
536                                     // default reset value is 0.
537
538 //*****
539 //          Command Audit
540 //*****
541 // When an audited command completes, ExecuteCommand() checks the return
542 // value. If it is TPM_RC_SUCCESS, and the command is an audited command, the
543 // TPM will extend the cpHash and rpHash for the command to this value. If this
544 // digest was the Zero Digest before the cpHash was extended, the audit counter
545 // is incremented.
546
547     TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest

```

```

548                                     // by TPM2_GetCommandAuditDigest() or a
549                                     // TPM Reset.
550
551 //*****
552 //           Boot counter
553 //*****
554
555     UINT32          restartCount;      // This counter counts TPM Restarts.
556                                     // The default reset value is 0.
557
558 //*****
559 //           PCR
560 //*****
561 // This counter increments whenever the PCR are updated. This counter is preserved
562 // across TPM Resume even though the PCR are not preserved. This is because
563 // sessions remain active across TPM Restart and the count value in the session
564 // is compared to this counter so this counter must have values that are unique
565 // as long as the sessions are active.
566 // NOTE: A platform-specific specification may designate that certain PCR changes
567 // do not increment this counter to increment.
568     UINT32          pcrCounter;        // The default reset value is 0.
569
570 #ifndef TPM_ALG_ECC
571
572 //*****
573 //           ECDA
574 //*****
575     UINT64          commitCounter;     // This counter increments each time
576                                     // TPM2_Commit() returns
577                                     // TPM_RC_SUCCESS. The default reset
578                                     // value is 0.
579
580
581     TPM2B_NONCE     commitNonce;      // This random value is used to compute
582                                     // the commit values. The default reset
583                                     // value is from the RNG.
584
585 // This implementation relies on the number of bits in g_commitArray being a
586 // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
587     BYTE            commitArray[16];  // The default reset value is {0}.
588
589 #endif //TPM_ALG_ECC
590
591 } STATE_RESET_DATA;
592 extern STATE_RESET_DATA gr;

```

### 5.6.14 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.

```

593 #define RcSafeAddToResult(r, v) \
594     ((r) + ((r) & RC_FMT1) ? (v) : 0)

```

### 5.6.15 Private data

```

595 #if defined SESSION_PROCESS_C || defined GLOBAL_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

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

Array of authorization session attributes

```
597 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

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

Array of nonces provided by the caller for the corresponding sessions

```
599 TPM2B_NONCE                s_nonceCaller[MAX_SESSION_NUM];
```

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

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

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter

```
602 extern UINT32              s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
603 extern UINT32              s_decryptSessionIndex;
```

Index of a session used for audit

```
604 extern UINT32              s_auditSessionIndex;
```

The *cpHash* for an audit session

```
605 extern TPM2B_DIGEST        s_cpHashForAudit;
```

The *cpHash* for command audit

```
606 #ifndef TPM_CC_GetCommandAuditDigest
607 extern TPM2B_DIGEST        s_cpHashForCommandAudit;
608 #endif
```

Number of authorization sessions present in the command

```
609 extern UINT32              s_sessionNum;
```

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

```
610 extern BOOL                 s_DAPendingOnNV;
611 #endif // SESSION_PROCESS_C
612 #if defined DA_C || defined GLOBAL_C
```

From DA.c

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

```
613 extern UINT64      s_selfHealTimer;
```

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

```
614 UINT64      s_lockoutTimer;
615 #endif // DA_C
616 #if defined NV_C || defined GLOBAL_C
```

From NV.c

List of pre-defined address of reserved data

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

List of pre-defined reserved data size in byte

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

Size of data in RAM index buffer

```
619 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

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

Address of size of RAM index space in NV

```
621 extern UINT32      s_ramIndexSizeAddr;
```

Address of NV copy of RAM index space

```
622 extern UINT32      s_ramIndexAddr;
```

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

```
623 extern UINT32      s_maxCountAddr;
```

Beginning of NV dynamic area; starts right after the *s\_maxCountAddr* and *s\_evictHandleMapAddr* variables

```
624 extern UINT32      s_evictNvStart;
```

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

```
625 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

```
626 extern TPM_RC      s_NvIsAvailable;
627 #endif
628 #if defined OBJECT_C || defined GLOBAL_C
```

From Object.c

This type is the container for an object.

```
629 typedef struct
630 {
631     BOOL        occupied;
632     ANY_OBJECT  object;
633 } OBJECT_SLOT;
```

This is the memory that holds the loaded objects.

```
634 extern OBJECT_SLOT    s_objects[MAX_LOADED_OBJECTS];
635 #endif // OBJECT_C
636 #if defined PCR_C || defined GLOBAL_C
```

From PCR.c

```
637 typedef struct
638 {
639     #ifdef TPM_ALG_SHA1
640         // SHA1 PCR
641         BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
642     #endif
643     #ifdef TPM_ALG_SHA256
644         // SHA256 PCR
645         BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
646     #endif
647     #ifdef TPM_ALG_SHA384
648         // SHA384 PCR
649         BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
650     #endif
651     #ifdef TPM_ALG_SHA512
652         // SHA512 PCR
653         BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
654     #endif
655     #ifdef TPM_ALG_SM3_256
656         // SHA256 PCR
657         BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
658     #endif
659 } PCR;
660 typedef struct
661 {
662     unsigned int    stateSave : 1;           // if the PCR value should be
663                                           // saved in state save
664     unsigned int    resetLocality : 5;      // The locality that the PCR
665                                           // can be reset
666     unsigned int    extendLocality : 5;     // The locality that the PCR
667                                           // can be extend
668 } PCR_Attributes;
669 extern PCR          s_pcrs[IMPLEMENTATION_PCR];
670 #endif // PCR_C
671 #if defined SESSION_C || defined GLOBAL_C
```

From Session.c

Container for HMAC or policy session tracking information

```
672 typedef struct
673 {
674     BOOL        occupied;
675     SESSION     session;           // session structure
676 } SESSION_SLOT;
677 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`.

```
678 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)

```
679 extern int            s_freeSessionSlots;
680 #endif // SESSION_C
681 #if defined MANUFACTURE_C || defined GLOBAL_C
```

From Manufacture.c

```
682 extern BOOL           s_manufactured;
683 #endif // MANUFACTURE_C
684 #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`.

```
685 extern BOOL           s_initialized;
686 #endif // POWER_C
687 #endif // GLOBAL_H
```

## 5.7 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 an 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

```

## 5.8 InternalRoutines.h

```

1 #ifndef INTERNAL_ROUTINES_H
2 #define INTERNAL_ROUTINES_H

```

Error Reporting

```

3 #include "TpmError.h"

```

NULL definition

```

4 #ifndef NULL
5 #define NULL (0)
6 #endif

```

UNUSED\_PARAMETER

```

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

```

Internal data definition

```

10 #include "Global.h"
11 #include "VendorString.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"

```

Internal support functions

```

22 #include "CommandCodeAttributes_fp.h"
23 #include "MemoryLib_fp.h"
24 #include "marshal_fp.h"
25 #include "Time_fp.h"
26 #include "Locality_fp.h"
27 #include "PP_fp.h"
28 #include "CommandAudit_fp.h"
29 #include "Manufacture_fp.h"
30 #include "Power_fp.h"
31 #include "Handle_fp.h"
32 #include "Commands_fp.h"
33 #include "AlgorithmCap_fp.h"
34 #include "PropertyCap_fp.h"
35 #include "Bits_fp.h"

```

Internal crypto functions

```

36 #include "Ticket_fp.h"
37 #include "CryptUtil_fp.h"
38 #endif

```

## 5.9 VendorString.h

```

1 #ifndef _VENDOR_STRING_H
2 #define _VENDOR_STRING_H

```

Define up to 4-byte values for MANUFACTURER. This value defines the response for TPM\_PT\_MANUFACTURER in TPM2\_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here.

```

3 #define MANUFACTURER "MSFT"

```

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

```

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

```

Define up to 4, 4-byte values. The values must each be 4 bytes long and the last value used may contain trailing zeros. These values define the response for TPM\_PT\_VENDOR\_STRING\_(1-4) in TPM2\_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here. The vendor strings 2-4 may also be defined as appropriately.

```
9  #define      VENDOR_STRING_1      "Micr"
10 #define      VENDOR_STRING_2      "osof"
11 #define      VENDOR_STRING_3      "t Co"
12 #define      VENDOR_STRING_4      "rp."
```

The following #if macro may be deleted after a proper VENDOR\_STRING\_1 is provided.

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

the more significant 32-bits of a vendor-specific value indicating the version of the firmware The following line should be un-commented and a vendor specific firmware V1 should be provided here. The FIRMWARE\_V2 may also be defined as appropriately.

```
18 #define      FIRMWARE_V1          (0x20130118)
```

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

```
19 #define      FIRMWARE_V2          (0x00093437)
```

The following #if macro may be deleted after a proper FIRMWARE\_V1 is provided.

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

## 6 Main

### 6.1 CommandDispatcher()

In the reference implementation, the command dispatch code is automatically generated by a program that uses part 3 as input. The function prototype header file (CommandDispatcher\_fp.h) is shown here.

CommandDispatcher() performs the following operations:

- Unmarshals command parameters from input buffer.
- Invokes the function that performs the command actions.
- Marshals the returned handles, if any.
- Marshals the returned parameters, if any, into the output buffer putting in the *parameterSize* field if authorization sessions are present.

NOTE A machine readable version of CommandDispatcher.c and CommandDispatcher\_fp.h is available from the TCG.

**[[CommandDispatcher\_fp\_h]]**

### 6.2 ExecCommand.c

#### 6.2.1 Introduction

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

#### 6.2.2 Includes

```
1 #include "InternalRoutines.h"
2 #include "HandleProcess_fp.h"
3 #include "SessionProcess_fp.h"
4 #include "CommandDispatcher_fp.h"
```

#### 6.2.3 ExecuteCommand()

The function performs the following steps.

- a) Parses the command header from input buffer.
- b) Calls *ParseHandleBuffer()* to parse the handle area of the command.
- c) Validates that each of the handles references a loaded entity.
- d) Calls *ParseSessionBuffer()* () to:
  - 1) unmarshal and parse the session area;
  - 2) check the authorizations; and
  - 3) when necessary, decrypt a parameter.
- e) Calls *CommandDispatcher()* to:
  - 1) unmarshal the command parameters from the command buffer;
  - 2) call the routine that performs the command actions; and
  - 3) marshal the responses into the response buffer.

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

```

5 void ExecuteCommand(
6     unsigned int     requestSize,        // IN: command buffer size
7     unsigned char    *request,          // IN: command buffer
8     unsigned int     *responseSize,     // OUT: response buffer size
9     unsigned char    **response         // OUT: response buffer
10 )
11 {
12     // Command local variables
13     TPM_ST           tag;                // these first three variables are the
14     UINT32           commandSize;
15     TPM_CC           commandCode = 0;
16
17     BYTE             *parmBufferStart;   // pointer to the first byte of an
18                                         // optional parameter buffer
19
20     UINT32           parmBufferSize = 0; // number of bytes in parameter area
21
22     UINT32           handleNum = 0;      // number of handles unmarshaled into
23                                         // the handles array
24
25     TPM_HANDLE       handles[MAX_HANDLE_NUM]; // array to hold handles in the
26                                         // command. Only handles in the handle
27                                         // area are stored here, not handles
28                                         // passed as parameters.
29
30     // Response local variables
31     TPM_RC           result;             // return code for the command
32
33     TPM_ST           resTag;             // tag for the response
34
35     UINT32           resHandleSize = 0;  // size of the handle area in the
36                                         // response. This is needed so that the
37                                         // handle area can be skipped when
38                                         // generating the rpHash.
39
40     UINT32           resParmSize = 0;    // the size of the response parameters
41                                         // These values go in the rpHash.
42
43     UINT32           resAuthSize = 0;    // size of authorization area in the
44                                         // response
45
46     INT32            size;               // remaining data to be unmarshaled
47                                         // or remaining space in the marshaling
48                                         // buffer
49
50     BYTE             *buffer;           // pointer into the buffer being used
51                                         // for marshaling or unmarshaling
52
53     UINT32           i;                 // local temp
54
55     // Assume that everything is going to work.
56     result = TPM_RC_SUCCESS;
57
58     // Set flags for NV access state. This should happen before any other
59     // operation that may require a NV write.

```

```

60     g_updateNV = FALSE;
61     g_clearOrderly = FALSE;
62
63     // Query platform to get the NV state. The result state is saved internally
64     // and will be reported by NvIsAvailable(). The reference code requires that
65     // accessibility of NV does not change during the execution of a command.
66     // Specifically, if NV is available when the command execution starts and then
67     // is not available later when it is necessary to write to NV, then the TPM
68     // will go into failure mode.
69     NvCheckState();
70
71     // Due to the limitations of the simulation, TPM clock must be explicitly
72     // synchronized with the system clock whenever a command is received.
73     // This function call is not necessary in a hardware TPM. However, taking
74     // a snapshot of the hardware timer at the beginning of the command allows
75     // the time value to be consistent for the duration of the command execution.
76     TimeUpdateToCurrent();
77
78     // Any command through this function will unceremoniously end the
79     // _TPM_Hash_Data/_TPM_Hash_End sequence.
80     if(g_DRTMHandle != TPM_RH_UNASSIGNED)
81         ObjectTerminateEvent();
82
83     // Get command buffer size and command buffer.
84     size = requestSize;
85     buffer = request;
86
87     // Parse command header: tag, commandSize and commandCode.
88     // First parse the tag. The unmarshaling routine will validate
89     // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
90     result = TPMI_ST_COMMAND_TAG_Unmarshal(&tag, &buffer, &size);
91     if(result != TPM_RC_SUCCESS)
92         goto Cleanup;
93
94     // Unmarshal the commandSize indicator.
95     result = UINT32_Unmarshal(&commandSize, &buffer, &size);
96     if(result != TPM_RC_SUCCESS)
97         goto Cleanup;
98
99     // On a TPM that receives bytes on a port, the number of bytes that were
100    // received on that port is requestSize it must be identical to commandSize.
101    // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
102    // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
103    // as the input processing (the function that receives the command bytes and
104    // places them in the input buffer) would likely have the input truncated when
105    // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
106    if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
107    {
108        result = TPM_RC_COMMAND_SIZE;
109        goto Cleanup;
110    }
111
112    // Unmarshal the command code.
113    result = TPM_CC_Unmarshal(&commandCode, &buffer, &size);
114    if(result != TPM_RC_SUCCESS)
115        goto Cleanup;
116
117    // Check to see if the command is implemented.
118    if(!CommandIsImplemented(commandCode))
119    {
120        result = TPM_RC_COMMAND_CODE;
121        goto Cleanup;
122    }
123
124    #if FIELD_UPGRADE_IMPLEMENTED == YES
125    // If the TPM is in FUM, then the only allowed command is

```

```

126     // TPM_CC_FieldUpgradeData.
127     if(IsFieldUpgradeMode() && (commandCode != TPM_CC_FieldUpgradeData))
128     {
129         result = TPM_RC_UPGRADE;
130         goto Cleanup;
131     }
132     else
133 #endif
134     // Excepting FUM, the TPM only accepts TPM2_Startup() after
135     // _TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
136     // is no longer allowed.
137     if(( !TPMIsStarted() && commandCode != TPM_CC_Startup)
138         || (TPMIsStarted() && commandCode == TPM_CC_Startup))
139     {
140         result = TPM_RC_INITIALIZE;
141         goto Cleanup;
142     }
143
144     // Start regular command process.
145     // Parse Handle buffer.
146     result = ParseHandleBuffer(commandCode, &buffer, &size, handles, &handleNum);
147     if(result != TPM_RC_SUCCESS)
148         goto Cleanup;
149
150     // Number of handles retrieved from handle area should be less than
151     // MAX_HANDLE_NUM.
152     pAssert(handleNum <= MAX_HANDLE_NUM);
153
154     // All handles in the handle area are required to reference TPM-resident
155     // entities.
156     for(i = 0; i < handleNum; i++)
157     {
158         result = EntityGetLoadStatus(&handles[i]);
159         if(result != TPM_RC_SUCCESS)
160         {
161             if(result == TPM_RC_REFERENCE_H0)
162                 result = result + i;
163             else
164                 result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
165             goto Cleanup;
166         }
167     }
168
169     // Authorization session handling for the command.
170     if(tag == TPM_ST_SESSIONS)
171     {
172         BYTE          *sessionBufferStart; // address of the session area first byte
173                                     // in the input buffer
174
175         UINT32        authorizationSize; // number of bytes in the session area
176
177         // Find out session buffer size.
178         result = UINT32_Unmarshal(&authorizationSize, &buffer, &size);
179         if(result != TPM_RC_SUCCESS)
180             goto Cleanup;
181
182         // Perform sanity check on the unmarshaled value. If it is smaller than
183         // the smallest possible session or larger than the remaining size of
184         // the command, then it is an error. NOTE: This check could pass but the
185         // session size could still be wrong. That will be determined after the
186         // sessions are unmarshaled.
187         if( authorizationSize < 9
188            || authorizationSize > (UINT32) size)
189         {
190             result = TPM_RC_SIZE;
191             goto Cleanup;

```

```

192     }
193
194     // The sessions, if any, follows authorizationSize.
195     sessionBufferStart = buffer;
196
197     // The parameters follow the session area.
198     parmBufferStart = sessionBufferStart + authorizationSize;
199
200     // Any data left over after removing the authorization sessions is
201     // parameter data. If the command does not have parameters, then an
202     // error will be returned if the remaining size is not zero. This is
203     // checked later.
204     parmBufferSize = size - authorizationSize;
205
206     // The actions of ParseSessionBuffer() are described in the introduction.
207     result = ParseSessionBuffer(commandCode,
208                                handleNum,
209                                handles,
210                                sessionBufferStart,
211                                authorizationSize,
212                                parmBufferStart,
213                                parmBufferSize);
214     if(result != TPM_RC_SUCCESS)
215         goto Cleanup;
216 }
217 else
218 {
219     // Whatever remains in the input buffer is used for the parameters of the
220     // command.
221     parmBufferStart = buffer;
222     parmBufferSize = size;
223
224     // The command has no authorization sessions.
225     // If the command requires authorizations, then CheckAuthNoSession() will
226     // return an error.
227     result = CheckAuthNoSession(commandCode, handleNum, handles,
228                                parmBufferStart, parmBufferSize);
229     if(result != TPM_RC_SUCCESS)
230         goto Cleanup;
231 }
232
233 // CommandDispatcher returns a response handle buffer and a response parameter
234 // buffer if it succeeds. It will also set the parameterSize field in the
235 // buffer if the tag is TPM_RC_SESSIONS.
236 result = CommandDispatcher(tag,
237                            commandCode,
238                            (INT32 *) &parmBufferSize,
239                            parmBufferStart,
240                            handles,
241                            &resHandleSize,
242                            &resParmSize);
243 if(result != TPM_RC_SUCCESS)
244     goto Cleanup;
245
246 // Build the session area at the end of the parameter area.
247 BuildResponseSession(tag,
248                     commandCode,
249                     resHandleSize,
250                     resParmSize,
251                     &resAuthSize);
252
253 Cleanup:
254 // This implementation loads an "evict" object to a transient object slot in
255 // RAM whenever an "evict" object handle is used in a command so that the
256 // access to any object is the same. These temporary objects need to be
257 // cleared from RAM whether the command succeeds or fails.

```

```

258     ObjectCleanupEvict();
259
260     // The response will contain at least a response header.
261     *responseSize = sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC);
262
263     // If the command completed successfully, then build the rest of the response.
264     if(result == TPM_RC_SUCCESS)
265     {
266         // Outgoing tag will be the same as the incoming tag.
267         resTag = tag;
268         // The overall response will include the handles, parameters,
269         // and authorizations.
270         *responseSize += resHandleSize + resParmSize + resAuthSize;
271
272         // Adding parameter size field.
273         if(tag == TPM_ST_SESSIONS)
274             *responseSize += sizeof(UINT32);
275
276         if( g_clearOrderly == TRUE
277            && gp.orderlyState != SHUTDOWN_NONE)
278         {
279             gp.orderlyState = SHUTDOWN_NONE;
280             NvWriteReserved(NV_ORDERLY, &gp.orderlyState);
281             g_updateNV = TRUE;
282         }
283     }
284     else
285     {
286         // The command failed.
287         resTag = TPM_ST_NO_SESSIONS;
288     }
289     // Try to commit all the writes to NV if any NV write happened during this
290     // command execution. This check should be made for both succeeded and failed
291     // commands, because a failed one may trigger a NV write in DA logic as well.
292     // This is the only place in the command execution path that may call the NV
293     // commit. If the NV commit fails, the TPM should be put in failure mode.
294     if(g_updateNV)
295     {
296         if(!NvCommit())
297             FAIL(FATAL_ERROR_INTERNAL);
298     }
299
300     // Marshal the response header.
301     buffer = MemoryGetResponseBuffer(commandCode);
302     TPM_ST_Marshal(&resTag, &buffer, NULL);
303     UINT32_Marshal((UINT32 *)responseSize, &buffer, NULL);
304     pAssert(*responseSize <= MAX_RESPONSE_SIZE);
305     TPM_RC_Marshal(&result, &buffer, NULL);
306
307     *response = MemoryGetResponseBuffer(commandCode);
308
309     // Clear unused bit in response buffer.
310     MemorySet(*response + *responseSize, 0, MAX_RESPONSE_SIZE - *responseSize);
311
312     return;
313 }

```

### 6.3 ParseHandleBuffer()

In the reference implementation, the routine for unmarshaling the command handles is automatically generated from part 3 command tables. The prototype header file (HandleProcess\_fp.h) is shown here.

[\[\[HandleProcess\\_fp\\_h\]\]](#)



## 6.4 SessionProcess.c

### 6.4.1 Introduction

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

### 6.4.2 Includes and Data Definitions

```

1  #define SESSION_PROCESS_C
2  #include "InternalRoutines.h"
3  #include "SessionProcess_fp.h"
4  #include "Platform.h"

```

### 6.4.3 Authorization Support Functions

#### 6.4.3.1 IsDAExempted()

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

- a) a primary seed handle,
- b) an object with *noDA* bit SET,
- c) an NV Index with TPMA\_NV\_NO\_DA bit SET, or
- d) a PCR handle.

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

```

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

```

```

32         if(nvIndex.publicArea.attributes.TPMA_NV_NO_DA == SET)
33             return TRUE;
34         break;
35     }
36     case TPM_HT_PCR:
37         // PCRs are always exempted from DA.
38         return TRUE;
39         break;
40     default:
41         break;
42 }
43
44 return FALSE;
45 }

```

#### 6.4.3.2 IncrementLockout()

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

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

```

46 static TPM_RC
47 IncrementLockout(
48     UINT32      sessionIndex
49 )
50 {
51     TPM_HANDLE   handle = s_associatedHandles[sessionIndex];
52     TPM_HANDLE   sessionHandle = s_sessionHandles[sessionIndex];
53     TPM_RC       result;
54     SESSION      *session = NULL;
55
56
57     // Don't increment lockout unless the handle associated with the session
58     // is DA protected or the session is bound to a DA protected entity.
59     if(sessionHandle == TPM_RS_PW)
60     {
61         if(IsDAExempted(handle))
62             return TPM_RC_BAD_AUTH;
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            )
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
84     if(handle == TPM_RH_LOCKOUT)
85     {
86         pAssert(gp.lockOutAuthEnabled);
87         gp.lockOutAuthEnabled = FALSE;
88         // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
89         // the lockout auth will be reset at startup.
90         if(gp.lockoutRecovery != 0)
91         {
92             result = NvIsAvailable();
93             if(result != TPM_RC_SUCCESS)
94             {
95                 // No NV access for now. Put the TPM in pending mode.
96                 s_DAPendingOnNV = TRUE;
97             }
98             else
99             {
100                // Update NV.
101                NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
102                g_updateNV = TRUE;
103            }
104        }
105    }
106    else
107    {
108        if(gp.recoveryTime != 0)
109        {
110            gp.failedTries++;
111            result = NvIsAvailable();
112            if(result != TPM_RC_SUCCESS)
113            {
114                // No NV access for now. Put the TPM in pending mode.
115                s_DAPendingOnNV = TRUE;
116            }
117            else
118            {
119                // Record changes to NV.
120                NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
121                g_updateNV = TRUE;
122            }
123        }
124    }
125
126    // Register a DA failure and reset the timers.
127    DARegisterFailure(handle);
128
129    return TPM_RC_AUTH_FAIL;
130 }

```

#### 6.4.3.3 IsSessionBindEntity()

This function indicates if the entity associated with the handle is the entity, to which this session is bound. The binding would occur by making the **bind** parameter in TPM2\_StartAuthSession() not equal to TPM\_RH\_NULL. The binding only occurs if the session is an HMAC session. The bind value is a combination of the Name and the *authValue* of the entity.

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

```

131 static BOOL
132 IsSessionBindEntity(

```

```

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

```

#### 6.4.3.4 IsWriteOperation()

This function indicates if a command is a write operation for an NV Index. It is only used in the context of NV commands. For other commands, the return value of this function has no meaning. The reason for checking on NV Index writes is that an NV Index has separate read and write authorizations.

Return Value	Meaning
TRUE	the command is an NV write operation
FALSE	the command is not an NV write operation

```

149     static BOOL
150     IsWriteOperation(
151         TPM_CC command_code
152     )
153     {
154         switch(command_code)
155         {
156             case TPM_CC_NV_Write:
157             case TPM_CC_NV_Increment:
158             case TPM_CC_NV_SetBits:
159             case TPM_CC_NV_Extend:
160                 return TRUE;
161             default:
162                 return FALSE;
163         }
164     }

```

#### 6.4.3.5 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:

- e) the command requires the DUP role,
- f) the command requires the ADMIN role and the authorized entity is an object and its *adminWithPolicy* bit is SET, or
- g) the command requires the ADMIN role and the authorized entity is a permanent handle.
- h) The authorized entity is a PCR belongs to a policy group, and has its policy initialized

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

```

165  static BOOL
166  IsPolicySessionRequired(
167      TPM_CC      commandCode,      // IN: command code
168      UINT32      sessionIndex     // IN: session index
169  )
170  {
171      AUTH_ROLE   role = CommandAuthRole(commandCode, sessionIndex);
172      TPM_HT      type = HandleGetType(s_associatedHandles[sessionIndex]);
173
174      if(role == AUTH_DUP)
175          return TRUE;
176
177      if(role == AUTH_ADMIN)
178      {
179          if(type == TPM_HT_TRANSIENT)
180          {
181              OBJECT      *object = ObjectGet(s_associatedHandles[sessionIndex]);
182
183              if(object->publicArea.objectAttributes.adminWithPolicy == CLEAR)
184                  return FALSE;
185          }
186          return TRUE;
187      }
188
189      if(type == TPM_HT_PCR)
190      {
191          if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
192          {
193              TPM2B_DIGEST      policy;
194              TPML_ALG_HASH     policyAlg;
195              policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
196                                          &policy);
197
198              if(policyAlg != TPM_ALG_NULL)
199                  return TRUE;
200          }
201          return FALSE;
202      }

```

#### 6.4.3.6 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.

Return Value	Meaning
TRUE	<i>authValue</i> is available
FALSE	<i>authValue</i> is not available

```

203  static BOOL
204  IsAuthValueAvailable(
205      TPM_HANDLE    handle,          // IN: handle of entity

```

```

206     TPM_CC      commandCode,      // IN: commandCode
207     UINT32      sessionIndex      // IN: session index
208 )
209 {
210     // If a policy session is required, the entity can not be authorized by
211     // authValue. However, at this point, the policy session requirement should
212     // already have been checked.
213     pAssert(!IsPolicySessionRequired(commandCode, sessionIndex));
214
215     switch(HandleGetType(handle))
216     {
217     case TPM_HT_PERMANENT:
218         switch(handle)
219         {
220             // At this point hierarchy availability has already been
221             // checked so primary seed handles are always available here
222             case TPM_RH_OWNER:
223             case TPM_RH_ENDORSEMENT:
224             case TPM_RH_PLATFORM:
225                 return TRUE;
226                 break;
227             case TPM_RH_LOCKOUT:
228                 // At the point when authValue availability is checked, control
229                 // path has already passed the DA check so LockOut auth is
230                 // always available here
231                 return TRUE;
232                 break;
233             case TPM_RH_NULL:
234                 // NullAuth is always available.
235                 return TRUE;
236                 break;
237             default:
238                 // Otherwise authValue is not available.
239                 return FALSE;
240                 break;
241         }
242         break;
243     case TPM_HT_TRANSIENT:
244         // A persistent object has already been loaded and the internal
245         // handle changed.
246         {
247             OBJECT      *object;
248             object = ObjectGet(handle);
249
250             // authValue is always available for a sequence object.
251             if(ObjectIsSequence(object))
252                 return TRUE;
253
254             // authValue is available for an object if it has its sensitive
255             // portion loaded and
256             // 1. userWithAuth bit is SET, or
257             // 2. ADMIN role is required
258             if( object->attributes.publicOnly == CLEAR
259                 && (object->publicArea.objectAttributes.userWithAuth == SET
260                     || (CommandAuthRole(commandCode, sessionIndex) == AUTH_ADMIN
261                         && object->publicArea.objectAttributes.adminWithPolicy
262                             == CLEAR)))
263                 return TRUE;
264             else
265                 return FALSE;
266         }
267         break;
268     case TPM_HT_NV_INDEX:
269         // NV Index.
270         {
271             NV_INDEX      nvIndex;

```

```

272     NvGetIndexInfo(handle, &nvIndex);
273     if(IsWriteOperation(commandCode))
274     {
275         if (nvIndex.publicArea.attributes.TPMA_NV_AUTHWRITE == SET)
276             return TRUE;
277         else
278             return FALSE;
279     }
280     else
281     {
282         if (nvIndex.publicArea.attributes.TPMA_NV_AUTHREAD == SET)
283             return TRUE;
284         else
285             return FALSE;
286     }
287 }
288 break;
289 case TPM_HT_PCR:
290     // PCR handle.
291     // authValue is always allowed for PCR
292     return TRUE;
293 default:
294     // Otherwise, authValue is not available
295     return FALSE;
296     break;
297 }
298 }

```

#### 6.4.3.7 IsAuthPolicyAvailable()

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

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

Return Value	Meaning
TRUE	<i>authPolicy</i> is available
FALSE	<i>authPolicy</i> is not available

```

299 static BOOL
300 IsAuthPolicyAvailable(
301     TPM_HANDLE    handle,           // IN: handle of entity
302     TPM_CC        commandCode,     // IN: commandCode
303     UINT32        sessionIndex     // IN: session index
304 )
305 {
306     switch(HandleGetType(handle))
307     {
308     case TPM_HT_PERMANENT:
309         switch(handle)
310         {
311             // At this point hierarchy availability has already been checked.
312             case TPM_RH_OWNER:
313                 if (gp.ownerPolicy.t.size != 0)
314                     return TRUE;
315                 else
316                     return FALSE;
317             case TPM_RH_ENDORSEMENT:
318                 if (gp.endorsementPolicy.t.size != 0)
319                     return TRUE;
320                 else
321

```

```

322         return FALSE;
323
324     case TPM_RH_PLATFORM:
325         if (gc.platformPolicy.t.size != 0)
326             return TRUE;
327         else
328             return FALSE;
329     default:
330         // Otherwise, authPolicy is not available.
331         return FALSE;
332         break;
333     }
334     break;
335 case TPM_HT_TRANSIENT:
336 {
337     // Object handle.
338     // An evict object would already have been loaded and given a
339     // transient object handle by this point.
340     OBJECT *object = ObjectGet(handle);
341     // Policy authorization is not available for an object with only
342     // public portion loaded.
343     if(object->attributes.publicOnly == SET)
344         return FALSE;
345     // Policy authorization is always available for an object but
346     // is never available for a sequence.
347     if(ObjectIsSequence(object))
348         return FALSE;
349     else
350         return TRUE;
351     break;
352 }
353 case TPM_HT_NV_INDEX:
354     // An NV Index.
355 {
356     NV_INDEX     nvIndex;
357     NvGetIndexInfo(handle, &nvIndex);
358     // If the policy size is not zero, check if policy can be used.
359     if(nvIndex.publicArea.authPolicy.t.size != 0)
360     {
361         // If policy session is required for this handle, always
362         // uses policy regardless of the attributes bit setting
363         if(IsPolicySessionRequired(commandCode, sessionIndex))
364             return TRUE;
365         // Otherwise, the presence of the policy depends on the NV
366         // attributes.
367         if(IsWriteOperation(commandCode))
368         {
369             if (nvIndex.publicArea.attributes.TPMA_NV_POLICYWRITE == SET)
370                 return TRUE;
371             else
372                 return FALSE;
373         }
374         else
375         {
376             if (nvIndex.publicArea.attributes.TPMA_NV_POLICYREAD ==SET)
377                 return TRUE;
378             else
379                 return FALSE;
380         }
381     }
382     return FALSE;
383 }
384 break;
385 case TPM_HT_PCR:
386     // PCR handle.
387     if(PCRPolicyIsAvailable(handle))

```



```

388         return TRUE;
389     else
390         return FALSE;
391     break;
392 default:
393     // Otherwise, authPolicy is not available.
394     return FALSE;
395     break;
396 }
397
398 }

```

## 6.4.4 Session Parsing Functions

### 6.4.4.1 ComputeCpHash()

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

```

399 static void
400 ComputeCpHash(
401     TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
402     TPM_CC           commandCode,       // IN: command code
403     UINT32           handleNum,         // IN: number of handles
404     TPM_HANDLE       handles[],        // IN: array of handles
405     UINT32           parmBufferSize,   // IN: size of input parameter area
406     BYTE             *parmBuffer,      // IN: input parameter area
407     TPM2B_DIGEST    *cpHash,          // OUT: cpHash
408     TPM2B_DIGEST    *nameHash         // OUT: name hash of command
409 )
410 {
411     UINT32           i;
412     HASH_STATE       hashState;
413     TPM2B_NAME       name;
414
415     // cpHash = hash(commandCode [ || authName1
416     //                [ || authName2
417     //                [ || authName 3 ]]]
418     //                [ || parameters])
419     // A cpHash can contain just a commandCode only if the lone session is
420     // an audit session.
421
422     // Start cpHash.
423     cpHash->t.size = CryptStartHash(hashAlg, &hashState);
424
425     // Add commandCode.
426     CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
427
428     // Add authNames for each of the handles.
429     for(i = 0; i < handleNum; i++)
430     {
431         name.t.size = EntityGetName(handles[i], name.t.name);
432         CryptUpdateDigest2B(&hashState, &name.b);
433     }
434
435     // Add the parameters.
436     CryptUpdateDigest(&hashState, parmBufferSize, parmBuffer);
437
438     // Complete the hash.
439     CryptCompleteHash2B(&hashState, &cpHash->b);
440
441     // If the nameHash is needed, compute it here.
442     if(nameHash != NULL)
443     {

```

```

444     // Start name hash. hashState may be reused.
445     nameHash->t.size = CryptStartHash(hashAlg, &hashState);
446
447     // Adding names.
448     for(i = 0; i < handleNum; i++)
449     {
450         name.t.size = EntityGetName(handles[i], name.t.name);
451         CryptUpdateDigest2B(&hashState, &name.b);
452     }
453     // Complete hash.
454     CryptCompleteHash2B(&hashState, &nameHash->b);
455 }
456 return;
457 }

```

#### 6.4.4.2 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[]*.

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

```

458 static TPM_RC
459 CheckPWAuthSession(
460     UINT32      sessionIndex      // IN: index of session to be processed
461 )
462 {
463     TPM2B_AUTH  authValue;
464     TPM_HANDLE  associatedHandle = s_associatedHandles[sessionIndex];
465
466     // Strip trailing zeros from the password.
467     MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
468
469     // Get the auth value and size.
470     authValue.t.size = EntityGetAuthValue(associatedHandle, authValue.t.buffer);
471
472     // Success if the digests are identical.
473     if(Memory2BEqual(&s_inputAuthValues[sessionIndex].b, &authValue.b))
474     {
475         return TPM_RC_SUCCESS;
476     }
477     else // if the digests are not identical
478     {
479         // Invoke DA protection if applicable.
480         return IncrementLockout(sessionIndex);
481     }
482 }

```

#### 6.4.4.3 ComputeCommandHMAC()

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

```

483 static void
484 ComputeCommandHMAC(
485     UINT32      sessionIndex,      // IN: index of session to be processed
486     TPM2B_DIGEST *cpHash,         // IN: cpHash
487     TPM2B_DIGEST *hmac            // OUT: authorization HMAC
488 )

```

```

489 {
490     TPM2B_TYPE(KEY, (sizeof(TPMT_HA) * 2));
491     TPM2B_KEY    key;
492     BYTE        marshalBuffer[sizeof(TPMA_SESSION)];
493     BYTE        *buffer;
494     UINT32      marshalSize;
495     HMAC_STATE  hmacState;
496     TPM2B_NONCE *nonceDecrypt;
497     TPM2B_NONCE *nonceEncrypt;
498     SESSION     *session;
499
500     nonceDecrypt = NULL;
501     nonceEncrypt = NULL;
502
503     // Determine if extra nonceTPM values are going to be required.
504     // If this is the first session (sessionIndex = 0) and it is an authorization
505     // session that uses an HMAC, then check if additional session nonces are to be
506     // included.
507     if( sessionIndex == 0
508         && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
509     {
510         // If there is a decrypt session and if this is not the decrypt session,
511         // then an extra nonce may be needed.
512         if( s_decryptSessionIndex != UNDEFINED_INDEX
513            && s_decryptSessionIndex != sessionIndex)
514         {
515             // Will add the nonce for the decrypt session.
516             session = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
517             nonceDecrypt = &session->nonceTPM;
518         }
519         // Now repeat for the encrypt session.
520         if( s_encryptSessionIndex != UNDEFINED_INDEX
521            && s_encryptSessionIndex != sessionIndex
522            && s_encryptSessionIndex != s_decryptSessionIndex)
523         {
524             // Have to have the nonce for the encrypt session.
525             session = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
526             nonceEncrypt = &session->nonceTPM;
527         }
528     }
529
530     // Continue with the HMAC processing.
531     session = SessionGet(s_sessionHandles[sessionIndex]);
532
533     // Generate HMAC key.
534     MemoryCopy2B(&key.b, &session->sessionKey.b);
535
536     // Check if the session has an associated handle and if the associated entity
537     // is the one to which the session is bound. If not, add the authValue of
538     // this entity to the HMAC key.
539     // If the session is bound to the object or the session is a policy session
540     // with no authValue required, do not include the authValue in the HMAC key.
541     // Note: For a policy session, its isBound attribute is CLEARED.
542     if( s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED
543        && !( HandleGetType(s_sessionHandles[sessionIndex])
544              == TPM_HT_POLICY_SESSION
545              && session->attributes.isAuthValueNeeded == CLEAR)
546        && !IsSessionBindEntity(s_associatedHandles[sessionIndex], session)
547        )
548     {
549         key.t.size = key.t.size
550             + EntityGetAuthValue(s_associatedHandles[sessionIndex],
551                                 &key.b.buffer[key.b.size]);
552     }
553
554     // if the HMAC key size for a policy session is 0, a NULL string HMAC is

```

```

555     // allowed.
556     if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
557         && key.t.size == 0
558         && s_inputAuthValues[sessionIndex].t.size == 0)
559     {
560         hmac->t.size = 0;
561         return;
562     }
563
564     // Start HMAC
565     hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
566
567     // Add cpHash
568     CryptUpdateDigest2B(&hmacState, &cpHash->b);
569
570     // Add nonceCaller
571     CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
572
573     // Add nonceTPM
574     CryptUpdateDigest2B(&hmacState, &session->nonceTPM.b);
575
576     // If needed, add nonceTPM for decrypt session
577     if(nonceDecrypt != NULL)
578         CryptUpdateDigest2B(&hmacState, &nonceDecrypt->b);
579
580     // If needed, add nonceTPM for encrypt session
581     if(nonceEncrypt != NULL)
582         CryptUpdateDigest2B(&hmacState, &nonceEncrypt->b);
583
584     // Add sessionAttributes
585     buffer = marshalBuffer;
586     marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
587                                         &buffer, NULL);
588     CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
589
590     // Complete the HMAC computation
591     CryptCompleteHMAC2B(&hmacState, &hmac->b);
592
593     return;
594 }

```

#### 6.4.4.4 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.

Error Returns	Meaning
TPM_RC_AUTH_FAIL	auth failure caused <i>failureCount</i> increment
TPM_RC_BAD_AUTH	auth failure did not cause <i>failureCount</i> increment

```

595     static TPM_RC
596     CheckSessionHMAC(
597         UINT32             sessionIndex,    // IN: index of session to be processed
598         TPM2B_DIGEST      *cpHash,        // IN: cpHash of the command
599     )
600     {
601         TPM2B_DIGEST      hmac;           // authHMAC for comparing
602

```

```

603     // Compute authHMAC
604     ComputeCommandHMAC(sessionIndex, cpHash, &hmac);
605
606     // Compare the input HMAC with the authHMAC computed above.
607     if(!Memory2BEqual(&s_inputAuthValues[sessionIndex].b, &hmac.b))
608     {
609         // If an HMAC session has a failure, invoke the anti-hammering
610         // if it applies to the authorized entity or the session.
611         // Otherwise, just indicate that the authorization is bad.
612         return IncrementLockout(sessionIndex);
613     }
614     return TPM_RC_SUCCESS;
615 }

```

#### 6.4.4.5 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:

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

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

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

```

616 static TPM_RC
617 CheckPolicyAuthSession(
618     UINT32      sessionIndex, // IN: index of session to be processed
619     TPM_CC      commandCode,  // IN: command code
620     TPM2B_DIGEST *cpHash,     // IN: cpHash using the algorithm of
621                               // this session
622     TPM2B_DIGEST *nameHash    // IN: nameHash using the session algorithm
623 )
624 {
625     TPM_RC      result = TPM_RC_SUCCESS;
626     SESSION     *session;
627     TPM2B_DIGEST authPolicy;
628     TPML_ALG_HASH policyAlg;
629     UINT8       locality;
630
631     // Initialize pointer to the auth session.
632     session = SessionGet(s_sessionHandles[sessionIndex]);

```

```

633
634 // See if the PCR counter for the session is still valid.
635 if( !SessionPCRValueIsCurrent(s_sessionHandles[sessionIndex]) )
636     return TPM_RC_PCR_CHANGED;
637
638 // Get authPolicy.
639 policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
640                               &authPolicy);
641
642 // Compare policy hash algorithm.
643 if(policyAlg != session->authHashAlg)
644     return TPM_RC_POLICY_FAIL;
645
646 // Compare timeout.
647 if(session->timeOut != 0)
648 {
649     // Cannot compare time if clock stop advancing. An TPM_RC_NV_UNAVAILABLE
650     // or TPM_RC_NV_RATE error may be returned here.
651     result = NvIsAvailable();
652     if(result != TPM_RC_SUCCESS)
653         return result;
654
655     if(session->timeOut < go.clock)
656         return TPM_RC_EXPIRED;
657 }
658
659 // If command code is provided it must match
660 if(session->commandCode != 0)
661 {
662     if(session->commandCode != commandCode)
663         return TPM_RC_POLICY_CC;
664 }
665 else
666 {
667     // If command requires a DUP or ADMIN authorization, the session must have
668     // command code set.
669     AUTH_ROLE role = CommandAuthRole(commandCode, sessionIndex);
670     if(role == AUTH_ADMIN || role == AUTH_DUP)
671         return TPM_RC_POLICY_FAIL;
672 }
673 // Check command locality.
674 {
675     BYTE sessionLocality[sizeof(TPMA_LOCALITY)];
676     BYTE *buffer = sessionLocality;
677
678     // Get existing locality setting in canonical form
679     TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
680
681     // See if the locality has been set
682     if(sessionLocality[0] != 0)
683     {
684         // If so, get the current locality
685         locality = _plat__LocalityGet();
686         if (locality < 5)
687         {
688             if(*(UINT8*)&session->commandLocality != 1 << locality)
689                 return TPM_RC_LOCALITY;
690         }
691         else if (locality > 31)
692         {
693             if(*(UINT8*)&session->commandLocality != locality)
694                 return TPM_RC_LOCALITY;
695         }
696         else
697         {
698             pAssert(FALSE);

```

```

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 // Compare authPolicy.
727 if(!Memory2BEqual(&session->u2.policyDigest.b, &authPolicy.b))
728     return TPM_RC_POLICY_FAIL;
729
730 return TPM_RC_SUCCESS;
731 }

```

#### 6.4.4.6 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.

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

```

732 static TPM_RC
733 RetrieveSessionData (
734     TPM_CC      commandCode,      // IN: command code
735     UINT32      *sessionCount,    // OUT: number of sessions found
736     BYTE        *sessionBuffer,   // IN: pointer to the session buffer
737     INT32      bufferSize        // IN: size of the session buffer
738 )
739 {
740     int         sessionIndex;
741     int         i;
742     TPM_RC      result;
743     SESSION    *session;
744     TPM_HT      sessionType;
745
746     s_decryptSessionIndex = UNDEFINED_INDEX;
747     s_encryptSessionIndex = UNDEFINED_INDEX;
748     s_auditSessionIndex = UNDEFINED_INDEX;
749 }

```

```

750     for(sessionIndex = 0; bufferSize > 0; sessionIndex++)
751     {
752         // If maximum allowed number of sessions has been parsed, exit the loop.
753         if(sessionIndex == MAX_SESSION_NUM)
754             break;
755
756         // make sure that the associated handle for each session starts out
757         // unassigned
758         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
759
760         // First parameter: Session handle.
761         result = TPMSI_SH_AUTH_SESSION_Unmarshal(&s_sessionHandles[sessionIndex],
762                                                  &sessionBuffer, &bufferSize, TRUE);
763         if(result != TPM_RC_SUCCESS)
764             return result + TPM_RC_S + g_rcIndex[sessionIndex];
765
766         // Second parameter: Nonce.
767         result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
768                                       &sessionBuffer, &bufferSize);
769         if(result != TPM_RC_SUCCESS)
770             return result + TPM_RC_S + g_rcIndex[sessionIndex];
771
772         // Third parameter: sessionAttributes.
773         result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
774                                       &sessionBuffer, &bufferSize);
775         if(result != TPM_RC_SUCCESS)
776             return result + TPM_RC_S + g_rcIndex[sessionIndex];
777
778         // Fourth parameter: authValue (PW or HMAC).
779         result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
780                                     &sessionBuffer, &bufferSize);
781         if(result != TPM_RC_SUCCESS)
782             return result + TPM_RC_S + g_rcIndex[sessionIndex];
783
784         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
785         {
786             // A PWAP session needs additional processing.
787             // Can't have any attributes set other than continueSession bit
788             if(
789                 s_attributes[sessionIndex].encrypt
790                 || s_attributes[sessionIndex].decrypt
791                 || s_attributes[sessionIndex].audit
792                 || s_attributes[sessionIndex].auditExclusive
793                 || s_attributes[sessionIndex].auditReset
794             )
795                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
796
797             // The nonce size must be zero.
798             if(s_nonceCaller[sessionIndex].t.size != 0)
799                 return TPM_RC_NONCE + TPM_RC_S + g_rcIndex[sessionIndex];
800
801             continue;
802         }
803         // For not password sessions...
804
805         // Find out if the session is loaded.
806         if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
807             return TPM_RC_REFERENCE_S0 + sessionIndex;
808
809         sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
810         session = SessionGet(s_sessionHandles[sessionIndex]);
811         // Check if the session is an HMAC/policy session.
812         if(
813             ( session->attributes.isPolicy == SET
814               && sessionType == TPM_HT_HMAC_SESSION
815             )
816             || (
817                 session->attributes.isPolicy == CLEAR
818                 && sessionType == TPM_HT_POLICY_SESSION
819             )
820         )

```



```

816     )
817   )
818     return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
819
820   // Check that this handle has not previously been used.
821   for(i = 0; i < sessionIndex; i++)
822   {
823     if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
824       return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
825   }
826
827   // If the session is used for parameter encryption or audit as well, set
828   // the corresponding indices.
829
830   // First process decrypt.
831   if(s_attributes[sessionIndex].decrypt)
832   {
833     // Check if the commandCode allows command parameter encryption.
834     if(!CommandIsDecryptAllowed(commandCode))
835       return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
836
837     // Encrypt attribute can only appear in one session
838     if(s_decryptSessionIndex != UNDEFINED_INDEX)
839       return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
840
841     // All checks passed, so set the index for the session used to decrypt
842     // a command parameter.
843     s_decryptSessionIndex = sessionIndex;
844   }
845
846   // Now process encrypt.
847   if(s_attributes[s_sessionNum].encrypt)
848   {
849     // Check if the commandCode allows response parameter encryption.
850     if(!CommandIsEncryptAllowed(commandCode))
851       return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
852
853     // Encrypt attribute can only appear in one session.
854     if(s_encryptSessionIndex != UNDEFINED_INDEX)
855       return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
856
857     // All checks passed, so set the index for the session used to encrypt
858     // a response parameter.
859     s_encryptSessionIndex = sessionIndex;
860   }
861
862   // At last process audit.
863   if(s_attributes[sessionIndex].audit)
864   {
865     // Audit attribute can only appear in one session.
866     if(s_auditSessionIndex != UNDEFINED_INDEX)
867       return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
868
869     // An audit session can not be policy session.
870     if( HandleGetType(s_sessionHandles[sessionIndex])
871        == TPM_HT_POLICY_SESSION)
872       return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
873
874     // If this is a reset of the audit session, or the first use
875     // of the session as an audit session, it doesn't matter what
876     // the exclusive state is. The session will become exclusive.
877     if( s_attributes[sessionIndex].auditReset == CLEAR
878        && session->attributes.isAudit == SET)
879     {
880       // Not first use or reset. If auditExclusive is SET, then this
881       // session must be the current exclusive session.

```

```

882         if( s_attributes[sessionIndex].auditExclusive == SET
883             && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
884             return TPM_RC_EXCLUSIVE;
885     }
886
887     s_auditSessionIndex = sessionIndex;
888 }
889
890 // Initialize associated handle as undefined. This will be changed when
891 // the handles are processed.
892 s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
893
894 }
895
896 // At this point either all session data has been processed or sessions limit
897 // has been reached. In either case, the remaining size should be zero
898 if(bufferSize != 0)
899     return TPM_RC_SIZE + TPM_RC_S + g_rcIndex[sessionIndex+1];
900
901 // Set the number of sessions found.
902 *sessionCount = sessionIndex;
903 return TPM_RC_SUCCESS;
904 }

```

#### 6.4.4.7 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*

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

```

905 static TPM_RC
906 CheckLockedOut(
907     BOOL lockoutAuthCheck // IN: TRUE if checking is for lockoutAuth
908 )
909 {
910     TPM_RC result;
911
912     // If NV is unavailable, and current cycle state recorded in NV is not
913     // SHUTDOWN_NONE, refuse to check any authorization because we would
914     // not be able to handle a DA failure.
915     result = NvIsAvailable();
916     if(result != TPM_RC_SUCCESS && gp.orderlyState != SHUTDOWN_NONE)
917         return result;
918
919     // Check if DA info needs to be updated in NV.
920     if(s_DAPendingOnNV)
921     {
922         // If NV is accessible, ...
923         if(result == TPM_RC_SUCCESS)
924         {
925             // ... write the pending DA data and proceed.
926             NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED,
927                             &gp.lockOutAuthEnabled);
928             NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
929             g_updateNV = TRUE;

```

```

930         s_DAPendingOnNV = FALSE;
931     }
932     else
933     {
934         // Otherwise no authorization can be checked.
935         return result;
936     }
937 }
938
939 // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
940 // is disabled...
941 if(lockoutAuthCheck)
942 {
943     if(gp.lockOutAuthEnabled == FALSE)
944         return TPM_RC_LOCKOUT;
945 }
946 else
947 {
948     // ... or if the number of failed tries has been maxed out.
949     if(gp.failedTries >= gp.maxTries)
950         return TPM_RC_LOCKOUT;
951 }
952 return TPM_RC_SUCCESS;
953 }

```

#### 6.4.4.8 CheckAuthSession()

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

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

```

954 static TPM_RC
955 CheckAuthSession(
956     TPM_CC      commandCode,           // IN: commandCode
957     UINT32      sessionIndex,         // IN: index of session to be processed
958     TPM2B_DIGEST *cpHash,            // IN: cpHash
959     TPM2B_DIGEST *nameHash           // IN: nameHash
960 )
961 {
962     TPM_RC      result;
963     SESSION     *session = NULL;
964     TPM_HANDLE  sessionHandle = s_sessionHandles[sessionIndex];
965     TPM_HANDLE  associatedHandle = s_associatedHandles[sessionIndex];
966     TPM_HT      sessionHandleType = HandleGetType(sessionHandle);
967

```

```

968     pAssert(sessionHandle != TPM_RH_UNASSIGNED);
969
970     if(sessionHandle != TPM_RS_PW)
971         session = SessionGet(sessionHandle);
972
973     // If the authorization session is not a policy session, or if the policy
974     // session requires authorization, then check lockout.
975     if( HandleGetType(sessionHandle) != TPM_HT_POLICY_SESSION
976         || session->attributes.isAuthValueNeeded
977         || session->attributes.isPasswordNeeded)
978     {
979         // See if entity is subject to lockout.
980         if(!IsDAExempted(associatedHandle))
981         {
982             // If NV is unavailable, and current cycle state recorded in NV is not
983             // SHUTDOWN_NONE, refuse to check any authorization because we would
984             // not be able to handle a DA failure.
985             result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
986             if(result != TPM_RC_SUCCESS)
987                 return result;
988         }
989     }
990
991     if(associatedHandle == TPM_RH_PLATFORM)
992     {
993         // If the physical presence is required for this command, check for PP
994         // assertion. If it isn't asserted, no point going any further.
995         if( PhysicalPresenceIsRequired(commandCode)
996            && !_plat__PhysicalPresenceAsserted()
997            )
998             return TPM_RC_PP;
999     }
1000    // If a policy session is required, make sure that it is being used.
1001    if( IsPolicySessionRequired(commandCode, sessionIndex)
1002        && sessionHandleType != TPM_HT_POLICY_SESSION)
1003        return TPM_RC_AUTH_TYPE;
1004
1005    // If this is a PW authorization, check it and return.
1006    if(sessionHandle == TPM_RS_PW)
1007    {
1008        if(IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
1009            return CheckPWAAuthSession(sessionIndex);
1010        else
1011            return TPM_RC_AUTH_UNAVAILABLE;
1012    }
1013    // If this is a policy session, ...
1014    if(sessionHandleType == TPM_HT_POLICY_SESSION)
1015    {
1016        // ... see if the entity has a policy, ...
1017        if( !IsAuthPolicyAvailable(associatedHandle, commandCode, sessionIndex))
1018            return TPM_RC_AUTH_UNAVAILABLE;
1019        // ... and check the policy session.
1020        result = CheckPolicyAuthSession(sessionIndex, commandCode,
1021                                       cpHash, nameHash);
1022        if (result != TPM_RC_SUCCESS)
1023            return result;
1024    }
1025    else
1026    {
1027        // For non policy, the entity being accessed must allow authorization
1028        // with an auth value. This is required even if the auth value is not
1029        // going to be used in an HMAC because it is bound.
1030        if(!IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
1031            return TPM_RC_AUTH_UNAVAILABLE;
1032    }
1033    // At this point, the session must be either a policy or an HMAC session.

```

```

1034     session = SessionGet(s_sessionHandles[sessionIndex]);
1035
1036     if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1037         && session->attributes.isPasswordNeeded == SET)
1038     {
1039         // For policy session that requires a password, check it as PWAP session.
1040         return CheckPWAAuthSession(sessionIndex);
1041     }
1042     else
1043     {
1044         // For other policy or HMAC sessions, have its HMAC checked.
1045         return CheckSessionHMAC(sessionIndex, cpHash);
1046     }
1047 }
1048 #ifdef TPM_CC_GetCommandAuditDigest

```

#### 6.4.4.9 CheckCommandAudit()

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

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

```

1049 static TPM_RC
1050 CheckCommandAudit(
1051     TPM_CC      commandCode,      // IN: Command code
1052     UINT32      handleNum,        // IN: number of element in handle array
1053     TPM_HANDLE  handles[],        // IN: array of handles
1054     BYTE        *parmBufferStart, // IN: start of parameter buffer
1055     UINT32      parmBufferSize    // IN: size of parameter buffer
1056 )
1057 {
1058     TPM_RC      result = TPM_RC_SUCCESS;
1059
1060     // If audit is implemented, need to check to see if auditing is being done
1061     // for this command.
1062     if(CommandAuditIsRequired(commandCode))
1063     {
1064         // If the audit digest is clear and command audit is required, NV must be
1065         // available so that TPM2_GetCommandAuditDigest() is able to increment
1066         // audit counter. If NV is not available, the function bails out to prevent
1067         // the TPM from attempting an operation that would fail anyway.
1068         if( gr.commandAuditDigest.t.size == 0
1069             || commandCode == TPM_CC_GetCommandAuditDigest)
1070         {
1071             result = NvIsAvailable();
1072             if(result != TPM_RC_SUCCESS)
1073                 return result;
1074         }
1075         ComputeCpHash(gp.auditHashAlg, commandCode, handleNum,
1076                     handles, parmBufferSize, parmBufferStart,
1077                     &s_cpHashForCommandAudit, NULL);
1078     }
1079
1080     return TPM_RC_SUCCESS;
1081 }
1082 #endif

```

### 6.4.4.10 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.

Error Returns	Meaning
Parsing Error	failure

```

1083 TPM_RC
1084 ParseSessionBuffer(
1085     TPM_CC      commandCode,      // IN: Command code
1086     UINT32      handleNum,        // IN: number of element in handle array
1087     TPM_HANDLE  handles[],        // IN: array of handles
1088     BYTE        *sessionBufferStart, // IN: start of session buffer
1089     UINT32      sessionBufferSize, // IN: size of session buffer
1090     BYTE        *parmBufferStart,  // IN: start of parameter buffer
1091     UINT32      parmBufferSize    // IN: size of parameter buffer
1092 )
1093 {
1094     TPM_RC      result;
1095     UINT32      i;
1096     INT32       size = 0;
1097     TPM2B_AUTH  extraKey;
1098     UINT32      sessionIndex;
1099     SESSION     *session;
1100     TPM2B_DIGEST cpHash;
1101     TPM2B_DIGEST nameHash;
1102     TPM_ALG_ID  cpHashAlg = TPM_ALG_NULL; // algID for the last computed
1103                                           // cpHash
1104
1105     // Check if a command allows any session in its session area.
1106     if(!IsSessionAllowed(commandCode))
1107         return TPM_RC_AUTH_CONTEXT;
1108
1109     // Default-initialization.
1110     s_sessionNum = 0;
1111     cpHash.t.size = 0;
1112
1113     result = RetrieveSessionData(commandCode, &s_sessionNum,
1114                                 sessionBufferStart, sessionBufferSize);
1115     if(result != TPM_RC_SUCCESS)
1116         return result;
1117
1118     // There is no command in the TPM spec that has more handles than
1119     // MAX_SESSION_NUM.
1120     pAssert(handleNum <= MAX_SESSION_NUM);
1121
1122     // Associate the session with an authorization handle.
1123     for(i = 0; i < handleNum; i++)
1124     {
1125         if(CommandAuthRole(commandCode, i) != AUTH_NONE)
1126         {
1127             // If the received session number is less than the number of handle
1128             // that requires authorization, an error should be returned.
1129             // Note: for all the TPM 2.0 commands, handles requiring
1130             // authorization come first in a command input.
1131             if(i > (s_sessionNum - 1))
1132                 return TPM_RC_AUTH_MISSING;
1133
1134             // Record the handle associated with the authorization session
1135             s_associatedHandles[i] = handles[i];
1136         }
1137     }

```

```

1138
1139 // Consistency checks are done first to avoid auth failure when the command
1140 // will not be executed anyway.
1141 for(sessionIndex = 0; sessionIndex < s_sessionNum; sessionIndex++)
1142 {
1143     // PW session must be an authorization session
1144     if(s_sessionHandles[sessionIndex] == TPM_RS_PW )
1145     {
1146         if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1147             return TPM_RC_HANDLE + g_rcIndex[sessionIndex];
1148     }
1149     else
1150     {
1151         session = SessionGet(s_sessionHandles[sessionIndex]);
1152
1153         // A trial session can not appear in session area, because it cannot
1154         // be used for authorization, audit or encrypt/decrypt.
1155         if(session->attributes.isTrialPolicy == SET)
1156             return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
1157
1158         // See if the session is bound to a DA protected entity
1159         if(session->attributes.isDaBound == SET)
1160         {
1161             result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1162             if(result != TPM_RC_SUCCESS)
1163                 return result;
1164         }
1165         // If the current cpHash is the right one, don't re-compute.
1166         if(cpHashAlg != session->authHashAlg) // different so compute
1167         {
1168             cpHashAlg = session->authHashAlg; // save this new algID
1169             ComputeCpHash(session->authHashAlg, commandCode, handleNum,
1170                 handles, parmBufferSize, parmBufferStart,
1171                 &cpHash, &nameHash);
1172         }
1173         // If this session is for auditing, save the cpHash.
1174         if(s_attributes[sessionIndex].audit)
1175             s_cpHashForAudit = cpHash;
1176     }
1177
1178     // if the session has an associated handle, check the auth
1179     if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1180     {
1181         result = CheckAuthSession(commandCode, sessionIndex,
1182             &cpHash, &nameHash);
1183         if(result != TPM_RC_SUCCESS)
1184             return RcSafeAddToResult(result,
1185                 TPM_RC_S + g_rcIndex[sessionIndex]);
1186     }
1187     else
1188     {
1189         // a session that is not for authorization must either be encrypt,
1190         // decrypt, or audit
1191         if(
1192             s_attributes[sessionIndex].audit == CLEAR
1193             && s_attributes[sessionIndex].encrypt == CLEAR
1194             && s_attributes[sessionIndex].decrypt == CLEAR)
1195             return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
1196
1197         // check HMAC for encrypt/decrypt/audit only sessions
1198         result = CheckSessionHMAC(sessionIndex, &cpHash);
1199         if(result != TPM_RC_SUCCESS)
1200             return RcSafeAddToResult(result,
1201                 TPM_RC_S + g_rcIndex[sessionIndex]);
1202     }
1203 }

```

```

1204 #ifndef TPM_CC_GetCommandAuditDigest
1205     // Check if the command should be audited.
1206     result = CheckCommandAudit(commandCode, handleNum, handles,
1207                               parmBufferStart, parmBufferSize);
1208     if(result != TPM_RC_SUCCESS)
1209         return result;           // No session number to reference
1210 #endif
1211
1212     // Decrypt the first parameter if applicable. This should be the last operation
1213     // in session processing.
1214     // If the encrypt session is associated with a handle and the handle's
1215     // authValue is available, then authValue is concatenated with sessionAuth to
1216     // generate encryption key, no matter if the handle is the session bound entity
1217     // or not.
1218     if(s_decryptSessionIndex != UNDEFINED_INDEX)
1219     {
1220         // Get size of the leading size field in decrypt parameter
1221         if( s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED
1222            && IsAuthValueAvailable(s_associatedHandles[s_decryptSessionIndex],
1223                                   commandCode,
1224                                   s_decryptSessionIndex)
1225            )
1226         {
1227             extraKey.b.size=
1228                 EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1229                                   extraKey.b.buffer);
1230         }
1231         else
1232         {
1233             extraKey.b.size = 0;
1234         }
1235         size = EncryptDecryptSize(commandCode);
1236         pAssert(size < INT16_MAX);
1237         result = CryptParameterDecryption(
1238             s_sessionHandles[s_decryptSessionIndex],
1239             &s_nonceCaller[s_decryptSessionIndex].b,
1240             parmBufferSize, (UINT16)size,
1241             &extraKey,
1242             parmBufferStart);
1243         if(result != TPM_RC_SUCCESS)
1244             return RcSafeAddToResult(result,
1245                                     TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1246     }
1247
1248     return TPM_RC_SUCCESS;
1249 }

```

#### 6.4.4.11 CheckAuthNoSession()

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

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

```

1250 TPM_RC
1251 CheckAuthNoSession(
1252     TPM_CC      commandCode,           // IN: Command Code
1253     UINT32      handleNum,             // IN: number of handles in command
1254     TPM_HANDLE  handles[],             // IN: array of handles
1255     BYTE        *parmBufferStart,      // IN: start of parameter buffer
1256     UINT32      parmBufferSize         // IN: size of parameter buffer
1257 )

```



```

1258 {
1259     UINT32 i;
1260     TPM_RC      result = TPM_RC_SUCCESS;
1261
1262     // Check if the commandCode requires authorization
1263     for(i = 0; i < handleNum; i++)
1264     {
1265         if(CommandAuthRole(commandCode, i) != AUTH_NONE)
1266             return TPM_RC_AUTH_MISSING;
1267     }
1268
1269 #ifdef TPM_CC_GetCommandAuditDigest
1270     // Check if the command should be audited.
1271     result = CheckCommandAudit(commandCode, handleNum, handles,
1272                               parmBufferStart, parmBufferSize);
1273     if(result != TPM_RC_SUCCESS) return result;
1274 #endif
1275
1276     // Initialize number of sessions to be 0
1277     s_sessionNum = 0;
1278
1279     return TPM_RC_SUCCESS;
1280 }

```

## 6.4.5 Response Session Processing

### 6.4.5.1 Introduction

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

### 6.4.5.2 ComputeRpHash()

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

```

1281 static void
1282 ComputeRpHash(
1283     TPM_ALG_ID      hashAlg,           // IN: hash algorithm to compute rpHash
1284     TPM_CC          commandCode,       // IN: commandCode
1285     UINT32          resParmBufferSize, // IN: size of response parameter buffer
1286     BYTE            *resParmBuffer,    // IN: response parameter buffer
1287     TPM2B_DIGEST    *rpHash           // OUT: rpHash
1288 )
1289 {
1290     // The command result in rpHash is always TPM_RC_SUCCESS.
1291     TPM_RC      responseCode = TPM_RC_SUCCESS;
1292     HASH_STATE  hashState;
1293
1294     // rpHash := hash(responseCode || commandCode || parameters)
1295
1296     // Initiate hash creation.
1297     rpHash->t.size = CryptStartHash(hashAlg, &hashState);
1298
1299     // Add hash constituents.
1300     CryptUpdateDigestInt(&hashState, sizeof(TPM_RC), &responseCode);
1301     CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
1302     CryptUpdateDigest(&hashState, resParmBufferSize, resParmBuffer);
1303
1304     // Complete hash computation.
1305     CryptCompleteHash2B(&hashState, &rpHash->b);
1306
1307     return;

```

1308 }

### 6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```

1309 static void
1310 InitAuditSession(
1311     SESSION      *session      // session to be initialized
1312 )
1313 {
1314     // Mark session as an audit session.
1315     session->attributes.isAudit = SET;
1316
1317     // Audit session can not be bound.
1318     session->attributes.isBound = CLEAR;
1319
1320     // Size of the audit log is the size of session hash algorithm digest.
1321     session->u2.auditDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
1322
1323     // Set the original digest value to be 0.
1324     MemorySet(&session->u2.auditDigest.t.buffer,
1325              0,
1326              session->u2.auditDigest.t.size);
1327
1328     return;
1329 }

```

### 6.4.5.4 Audit()

This function updates the audit digest in an audit session.

```

1330 static void
1331 Audit(
1332     SESSION      *auditSession, // IN: loaded audit session
1333     TPM_CC       commandCode,   // IN: commandCode
1334     UINT32       resParmBufferSize, // IN: size of response parameter buffer
1335     BYTE         *resParmBuffer  // IN: response parameter buffer
1336 )
1337 {
1338     TPM2B_DIGEST rpHash;        // rpHash for response
1339     HASH_STATE   hashState;
1340
1341     // Compute rpHash
1342     ComputeRpHash(auditSession->authHashAlg,
1343                  commandCode,
1344                  resParmBufferSize,
1345                  resParmBuffer,
1346                  &rpHash);
1347
1348     // auditDigestnew := hash (auditDigestold || cpHash || rpHash)
1349
1350     // Start hash computation.
1351     CryptStartHash(auditSession->authHashAlg, &hashState);
1352
1353     // Add old digest.
1354     CryptUpdateDigest2B(&hashState, &auditSession->u2.auditDigest.b);
1355
1356     // Add cpHash and rpHash.
1357     CryptUpdateDigest2B(&hashState, &s_cpHashForAudit.b);
1358     CryptUpdateDigest2B(&hashState, &rpHash.b);
1359
1360     // Finalize the hash.

```

```

1361     CryptCompleteHash2B(&hashState, &auditSession->u2.auditDigest.b);
1362
1363     return;
1364 }
1365 #ifdef TPM_CC_GetCommandAuditDigest

```

#### 6.4.5.5 CommandAudit()

This function updates the command audit digest.

```

1366 static void
1367 CommandAudit(
1368     TPM_CC      commandCode,      // IN: commandCode
1369     UINT32      resParmBufferSize, // IN: size of response parameter buffer
1370     BYTE        *resParmBuffer    // IN: response parameter buffer
1371 )
1372 {
1373     if(CommandAuditIsRequired(commandCode))
1374     {
1375         TPM2B_DIGEST    rpHash;      // rpHash for response
1376         HASH_STATE      hashState;
1377
1378         // Compute rpHash.
1379         ComputeRpHash(gp.auditHashAlg, commandCode, resParmBufferSize,
1380                     resParmBuffer, &rpHash);
1381
1382         // If the digest.size is one, it indicates the special case of changing
1383         // the audit hash algorithm. For this case, no audit is done on exit.
1384         // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1385         // force an update to the NV on exit so that the change in digest will
1386         // be recorded. So, it is safe to exit here without setting any flags
1387         // because the digest change will be written to NV when this code exits.
1388         if(gr.commandAuditDigest.t.size == 1)
1389         {
1390             gr.commandAuditDigest.t.size = 0;
1391             return;
1392         }
1393
1394         // If the digest size is zero, need to start a new digest and increment
1395         // the audit counter.
1396         if(gr.commandAuditDigest.t.size == 0)
1397         {
1398             gr.commandAuditDigest.t.size = CryptGetHashDigestSize(gp.auditHashAlg);
1399             MemorySet(gr.commandAuditDigest.t.buffer,
1400                     0,
1401                     gr.commandAuditDigest.t.size);
1402
1403             // Bump the counter and save its value to NV.
1404             gp.auditCounter++;
1405             NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
1406             g_updateNV = TRUE;
1407         }
1408
1409         // auditDigestnew := hash (auditDigestold || cpHash || rpHash)
1410
1411         // Start hash computation.
1412         CryptStartHash(gp.auditHashAlg, &hashState);
1413
1414         // Add old digest.
1415         CryptUpdateDigest2B(&hashState, &gr.commandAuditDigest.b);
1416
1417         // Add cpHash
1418         CryptUpdateDigest2B(&hashState, &s_cpHashForCommandAudit.b);
1419

```

```

1420     // Add rpHash
1421     CryptUpdateDigest2B(&hashState, &rpHash.b);
1422
1423     // Finalize the hash.
1424     CryptCompleteHash2B(&hashState, &gr.commandAuditDigest.b);
1425 }
1426 return;
1427 }
1428 #endif

```

#### 6.4.5.6 UpdateAuditSessionStatus()

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

- n) 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;
- o) reports exclusive audit session;
- p) extends audit log; and
- q) clears exclusive audit session if no audit session found in the command.

```

1429 static void
1430 UpdateAuditSessionStatus(
1431     TPM_CC      commandCode, // IN: commandCode
1432     UINT32      resParmBufferSize, // IN: size of response parameter buffer
1433     BYTE        *resParmBuffer // IN: response parameter buffer
1434 )
1435 {
1436     UINT32      i;
1437     TPM_HANDLE  auditSession = TPM_RH_UNASSIGNED;
1438
1439     // Iterate through sessions
1440     for (i = 0; i < s_sessionNum; i++)
1441     {
1442         SESSION *session;
1443
1444         // PW session do not have a loaded session and can not be an audit
1445         // session either. Skip it.
1446         if(s_sessionHandles[i] == TPM_RS_PW) continue;
1447
1448         session = SessionGet(s_sessionHandles[i]);
1449
1450         // If a session is used for audit
1451         if(s_attributes[i].audit == SET)
1452         {
1453             // An audit session has been found
1454             auditSession = s_sessionHandles[i];
1455
1456             // If the session has not been an audit session yet, or
1457             // the auditSetting bits indicate a reset, initialize it and set
1458             // it to be the exclusive session
1459             if( session->attributes.isAudit == CLEAR
1460                || s_attributes[i].auditReset == SET
1461            )
1462             {
1463                 InitAuditSession(session);
1464                 g_exclusiveAuditSession = auditSession;
1465             }
1466             else
1467             {
1468                 // Check if the audit session is the current exclusive audit
1469                 // session and, if not, clear previous exclusive audit session.
1470                 if(g_exclusiveAuditSession != auditSession)

```

```

1471         g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1472     }
1473
1474     // Report audit session exclusivity.
1475     if(g_exclusiveAuditSession == auditSession)
1476     {
1477         s_attributes[i].auditExclusive = SET;
1478     }
1479     else
1480     {
1481         s_attributes[i].auditExclusive = CLEAR;
1482     }
1483
1484     // Extend audit log.
1485     Audit(session, commandCode, resParmBufferSize, resParmBuffer);
1486 }
1487
1488
1489 // If no audit session is found in the command, and the command allows
1490 // a session then, clear the current exclusive
1491 // audit session.
1492 if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(commandCode))
1493 {
1494     g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1495 }
1496
1497 return;
1498 }

```

#### 6.4.5.7 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```

1499 static void
1500 ComputeResponseHMAC(
1501     UINT32         sessionIndex,           // IN: session index to be processed
1502     SESSION        *session,              // IN: loaded session
1503     TPM_CC         commandCode,           // IN: commandCode
1504     TPM2B_NONCE    *nonceTPM,            // IN: nonceTPM
1505     UINT32         resParmBufferSize,     // IN: size of response parameter
1506     // buffer
1507     BYTE           *resParmBuffer,        // IN: response parameter buffer
1508     TPM2B_DIGEST   *hmac                 // OUT: authHMAC
1509 )
1510 {
1511     TPM2B_TYPE(KEY, (sizeof(TPMT_HA) * 2));
1512     TPM2B_KEY       key;                  // HMAC key
1513     BYTE            marshalBuffer[sizeof(TPMA_SESSION)];
1514     BYTE            *buffer;
1515     UINT32          marshalSize;
1516     HMAC_STATE      hmacState;
1517     TPM2B_DIGEST    rp_hash;
1518
1519     // Compute rpHash.
1520     ComputeRpHash(session->authHashAlg, commandCode, resParmBufferSize,
1521         resParmBuffer, &rp_hash);
1522
1523     // Generate HMAC key
1524     MemoryCopy2B(&key.b, &session->sessionKey.b);
1525
1526     // Check if the session has an associated handle and the associated entity is
1527     // the one that the session is started with.
1528     // If so, add the authValue of this entity to the HMAC key.
1529     if( s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED

```

```

1530     && !( HandleGetType(s_sessionHandles[sessionIndex])
1531           == TPM_HT_POLICY_SESSION
1532           && session->attributes.isAuthValueNeeded == CLEAR)
1533     && !IsSessionBindEntity(s_associatedHandles[sessionIndex], session))
1534 {
1535     key.t.size = key.t.size +
1536                 EntityGetAuthValue(s_associatedHandles[sessionIndex],
1537                                   &key.t.buffer[key.t.size]);
1538 }
1539
1540 // if the HMAC key size for a policy session is 0, the response HMAC is
1541 // computed according to the input HMAC
1542 if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1543     && key.t.size == 0
1544     && s_inputAuthValues[sessionIndex].t.size == 0)
1545 {
1546     hmac->t.size = 0;
1547     return;
1548 }
1549
1550 // Start HMAC computation.
1551 hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
1552
1553 // Add hash components.
1554 CryptUpdateDigest2B(&hmacState, &rp_hash.b);
1555 CryptUpdateDigest2B(&hmacState, &nonceTPM->b);
1556 CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
1557
1558 // Add session attributes.
1559 buffer = marshalBuffer;
1560 marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1561 CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
1562
1563 // Finalize HMAC.
1564 CryptCompleteHMAC2B(&hmacState, &hmac->b);
1565
1566 return;
1567 }

```

#### 6.4.5.8 BuildSingleResponseAuth()

Function to compute response for an authorization session.

```

1568 static void
1569 BuildSingleResponseAuth(
1570     UINT32      sessionIndex,           // IN: session index to be processed
1571     TPM_CC      commandCode,           // IN: commandCode
1572     UINT32      resParmBufferSize,     // IN: size of response parameter buffer
1573     BYTE        *resParmBuffer,        // IN: response parameter buffer
1574     TPM2B_AUTH *auth                   // OUT: authHMAC
1575 )
1576 {
1577     // For password authorization, field is empty.
1578     if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1579     {
1580         auth->t.size = 0;
1581     }
1582     else
1583     {
1584         // Fill in policy/HMAC based session response.
1585         SESSION *session = SessionGet(s_sessionHandles[sessionIndex]);
1586
1587         // If the session is a policy session with isPasswordNeeded SET, the auth
1588         // field is empty.

```

```

1589     if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1590         && session->attributes.isPasswordNeeded == SET)
1591         auth->t.size = 0;
1592     else
1593         // Compute response HMAC.
1594         ComputeResponseHMAC(sessionIndex,
1595                             session,
1596                             commandCode,
1597                             &session->nonceTPM,
1598                             resParmBufferSize,
1599                             resParmBuffer,
1600                             auth);
1601     }
1602
1603     return;
1604 }

```

#### 6.4.5.9 UpdateTPMNonce()

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

```

1605 static void
1606 UpdateTPMNonce(
1607     TPM2B_NONCE    nonces[]           // OUT: nonceTPM
1608 )
1609 {
1610     UINT32    i;
1611     for(i = 0; i < s_sessionNum; i++)
1612     {
1613         SESSION    *session;
1614         // For PW session, nonce is 0.
1615         if(s_sessionHandles[i] == TPM_RS_PW)
1616         {
1617             nonces[i].t.size = 0;
1618             continue;
1619         }
1620         session = SessionGet(s_sessionHandles[i]);
1621         // Update nonceTPM in both internal session and response.
1622         CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
1623         nonces[i] = session->nonceTPM;
1624     }
1625     return;
1626 }

```

#### 6.4.5.10 UpdateInternalSession()

Updates internal sessions:

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

```

1627 static void
1628 UpdateInternalSession(void)
1629 {
1630     UINT32    i;
1631     for(i = 0; i < s_sessionNum; i++)
1632     {
1633         // For PW session, no update.
1634         if(s_sessionHandles[i] == TPM_RS_PW) continue;
1635
1636         if(s_attributes[i].continueSession == CLEAR)
1637         {

```

```

1638         // Close internal session.
1639         SessionFlush(s_sessionHandles[i]);
1640     }
1641     else
1642     {
1643         // If nonce is rolling in a policy session, the policy related data
1644         // will be re-initialized.
1645         if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION)
1646         {
1647             SESSION      *session = SessionGet(s_sessionHandles[i]);
1648
1649             // When the nonce rolls it starts a new timing interval for the
1650             // policy session.
1651             SessionResetPolicyData(session);
1652             session->startTime = go.clock;
1653         }
1654     }
1655 }
1656 return;
1657 }

```

#### 6.4.5.11 BuildResponseSession()

Function to build Session buffer in a response.

```

1658 void
1659 BuildResponseSession(
1660     TPM_ST      tag,                // IN: tag
1661     TPM_CC      commandCode,        // IN: commandCode
1662     UINT32      resHandleSize,      // IN: size of response handle buffer
1663     UINT32      resParmSize,        // IN: size of response parameter buffer
1664     UINT32      *resSessionSize     // OUT: response session area
1665 )
1666 {
1667     BYTE          *resParmBuffer;
1668     TPM2B_NONCE  responseNonces[MAX_SESSION_NUM];
1669
1670     // Compute response parameter buffer start.
1671     resParmBuffer = MemoryGetResponseBuffer(commandCode) + sizeof(TPM_ST) +
1672                     sizeof(UINT32) + sizeof(TPM_RC) + resHandleSize;
1673
1674     // For TPM_ST_SESSIONS, there is parameterSize field.
1675     if(tag == TPM_ST_SESSIONS)
1676         resParmBuffer += sizeof(UINT32);
1677
1678     // Session nonce should be updated before parameter encryption
1679     if(tag == TPM_ST_SESSIONS)
1680     {
1681         UpdateTPMNonce(responseNonces);
1682
1683         // Encrypt first parameter if applicable. Parameter encryption should
1684         // happen after nonce update and before any rpHash is computed.
1685         // If the encrypt session is associated with a handle, the authValue of
1686         // this handle will be concatenated with sessionAuth to generate
1687         // encryption key, no matter if the handle is the session bound entity
1688         // or not. The authValue is added to sessionAuth only when the authValue
1689         // is available.
1690         if(s_encryptSessionIndex != UNDEFINED_INDEX)
1691         {
1692             UINT32      size;
1693             TPM2B_AUTH  extraKey;
1694
1695             // Get size of the leading size field
1696             if( s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED

```



```

1697         && IsAuthValueAvailable(s_associatedHandles[s_encryptSessionIndex],
1698                                 commandCode, s_encryptSessionIndex)
1699     )
1700     {
1701         extraKey.b.size =
1702             EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1703                                 extraKey.b.buffer);
1704     }
1705     else
1706     {
1707         extraKey.b.size = 0;
1708     }
1709     size = EncryptDecryptSize(commandCode);
1710     pAssert(size < UINT16_MAX);
1711     CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1712                             &s_nonceCaller[s_encryptSessionIndex].b,
1713                             (UINT16)size,
1714                             &extraKey,
1715                             resParmBuffer);
1716
1717     }
1718
1719 }
1720 // Audit session should be updated first regardless of the tag.
1721 // A command with no session may trigger a change of the exclusivity state.
1722 UpdateAuditSessionStatus(commandCode, resParmSize, resParmBuffer);
1723
1724 // Audit command.
1725 CommandAudit(commandCode, resParmSize, resParmBuffer);
1726
1727 // Process command with sessions.
1728 if(tag == TPM_ST_SESSIONS)
1729 {
1730     UINT32         i;
1731     BYTE           *buffer;
1732     TPM2B_DIGEST   responseAuths[MAX_SESSION_NUM];
1733
1734     pAssert(s_sessionNum > 0);
1735
1736     // Iterate over each session in the command session area, and create
1737     // corresponding sessions for response.
1738     for(i = 0; i < s_sessionNum; i++)
1739     {
1740         BuildSingleResponseAuth(
1741             i,
1742             commandCode,
1743             resParmSize,
1744             resParmBuffer,
1745             &responseAuths[i]);
1746         // Make sure that continueSession is SET on any Password session.
1747         // This makes it marginally easier for the management software
1748         // to keep track of the closed sessions.
1749         if( s_attributes[i].continueSession == CLEAR
1750             && s_sessionHandles[i] == TPM_RS_PW)
1751         {
1752             s_attributes[i].continueSession = SET;
1753         }
1754     }
1755
1756     // Assemble Response Sessions.
1757     *resSessionSize = 0;
1758     buffer = resParmBuffer + resParmSize;
1759     for(i = 0; i < s_sessionNum; i++)
1760     {
1761         *resSessionSize += TPM2B_NONCE_Marshal(&responseNonces[i],
1762                                                 &buffer, NULL);

```

```
1763         *resSessionSize += TPMA_SESSION_Marshal(&s_attributes[i],
1764                                     &buffer, NULL);
1765         *resSessionSize += TPM2B_DIGEST_Marshal(&responseAuths[i],
1766                                     &buffer, NULL);
1767     }
1768
1769     // Update internal sessions after completing response buffer computation.
1770     UpdateInternalSession();
1771 }
1772 else
1773 {
1774     // Process command with no session.
1775     *resSessionSize = 0;
1776 }
1777
1778 return;
1779 }
```

## 7 Command Support Functions

### 7.1 Introduction

This clause contains support routines that are called by the command action code in part 3. The functions are grouped by the command group that is supported by the functions.

### 7.2 Attestation Command Support (Attest\_spt.c)

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

#### 7.2.1.1 FillInAttestInfo()

Fill in common fields of TPMS\_ATTEST structure.

Error Returns	Meaning
TPM_RC_KEY	key referenced by <i>signHandle</i> is not a signing key
TPM_RC_SCHEME	both <i>scheme</i> and key's default scheme are empty; or <i>scheme</i> is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from <i>scheme</i>

```
3 TPM_RC
4 FillInAttestInfo(
5     TPMI_DH_OBJECT      signHandle,    // IN: handle of signing object
6     TPMT_SIG_SCHEME     *scheme,       // IN/OUT: scheme to be used for signing
7     TPM2B_DATA          *data,         // IN: qualifying data
8     TPMS_ATTEST         *attest       // OUT: attest structure
9 )
10 {
11     TPM_RC      result;
12     TPMI_RH_HIERARCHY  signHierarhcy;
13
14     result = CryptSelectSignScheme(signHandle, scheme);
15     if(result != TPM_RC_SUCCESS)
16         return result;
17
18     // Magic number
19     attest->magic = TPM_GENERATED_VALUE;
20
21     if(signHandle == TPM_RH_NULL)
22     {
23         BYTE      *buffer;
24         // For null sign handle, the QN is TPM_RH_NULL
25         buffer = attest->qualifiedSigner.t.name;
26         attest->qualifiedSigner.t.size =
27             TPM_HANDLE_Marshal(&signHandle, &buffer, NULL);
28     }
29     else
30     {
31         // Certifying object qualified name
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         TPME_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         KDFa(hashAlg, &gp.shProof.b, "OBFUSCATE",
68             &attest->qualifiedSigner.b, NULL, 128, (BYTE *)&obfuscation[0], NULL);
69
70         // Obfuscate data
71         attest->firmwareVersion += obfuscation[0];
72         attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
73         attest->clockInfo.restartCount += (UINT32)obfuscation[1];
74     }
75
76     // External data
77     if(CryptIsSchemeAnonymous(scheme->scheme))
78         attest->extraData.t.size = 0;
79     else
80     {
81         // If we move the data to the attestation structure, then we will not use
82         // it in the signing operation except as part of the signed data
83         attest->extraData = *data;
84         data->t.size = 0;
85     }
86
87     return TPM_RC_SUCCESS;
88 }

```

### 7.2.1.2 SignAttestInfo()

Sign a TPMS\_ATTEST structure. If *signHandle* is TPM\_RH\_NULL, a null signature is returned.

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

89 **TPM\_RC**

```

90  SignAttestInfo(
91      TPMI_DH_OBJECT          signHandle,          // IN: handle of sign object
92      TPMT_SIG_SCHEME        *scheme,             // IN: sign scheme
93      TPMS_ATTEST            *certifyInfo,        // IN: the data to be signed
94      TPM2B_DATA             *qualifyingData,     // IN: extra data for the signing
95                                     // process
96      TPM2B_ATTEST          *attest,             // OUT: marshaled attest blob to
97                                     // be signed
98      TPMT_SIGNATURE         *signature          // OUT: signature
99  )
100 {
101     TPM_RC          result;
102     TPMI_ALG_HASH  hashAlg;
103     BYTE           *buffer;
104     HASH_STATE     hashState;
105     TPM2B_DIGEST   digest;
106
107
108     // Marshal TPMS_ATTEST structure for hash
109     buffer = attest->t.attestationData;
110     attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);
111
112     if(signHandle == TPM_RH_NULL)
113     {
114         signature->sigAlg = TPM_ALG_NULL;
115     }
116     else
117     {
118         // Attestation command may cause the orderlyState to be cleared due to
119         // the reporting of clock info. If this is the case, check if NV is
120         // available first
121         if(gp.orderlyState != SHUTDOWN_NONE)
122         {
123             // The command needs NV update. Check if NV is available.
124             // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
125             // this point
126             result = NvIsAvailable();
127             if(result != TPM_RC_SUCCESS)
128                 return result;
129         }
130
131         // Compute hash
132         hashAlg = scheme->details.any.hashAlg;
133         digest.t.size = CryptStartHash(hashAlg, &hashState);
134         CryptUpdateDigest(&hashState, attest->t.size, attest->t.attestationData);
135         CryptCompleteHash2B(&hashState, &digest.b);
136
137         // If there is qualifying data, need to rehash the the data
138         // hash(qualifyingData || hash(attestationData))
139         if(qualifyingData->t.size != 0)
140         {
141             CryptStartHash(hashAlg, &hashState);
142             CryptUpdateDigest(&hashState,
143                             qualifyingData->t.size,
144                             qualifyingData->t.buffer);
145             CryptUpdateDigest(&hashState, digest.t.size, digest.t.buffer);
146             CryptCompleteHash2B(&hashState, &digest.b);
147         }
148
149         // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
150         // TPM_RC_ATTRIBUTES error may be returned at this point
151         return CryptSign(signHandle,
152                         scheme,
153                         &digest,
154                         signature);
155     }

```

```

156
157     return TPM_RC_SUCCESS;
158 }

```

### 7.3 Context Management Command Support (Context\_spt.c)

```

1 #include "InternalRoutines.h"
2 #include "Context_spt_fp.h"

```

#### 7.3.1.1 ComputeContextProtectionKey()

This function retrieves the symmetric protection key for context encryption. It is used by TPM2\_ConextSave() and TPM2\_ContextLoad() to create the symmetric encryption key and iv.

```

3 void
4 ComputeContextProtectionKey(
5     TPM2B_CONTEXT *contextBlob, // IN: context blob
6     TPM2B_SYM_KEY *symKey,     // OUT: the symmetric key
7     TPM2B_IV *iv,              // OUT: the IV.
8 )
9 {
10     UINT16      symKeyBits; // number of bits in the parent's
11                                     // symmetric key
12     TPM2B_AUTH *proof = NULL; // the proof value to use. Is null for
13                                     // everything but a primary object in
14                                     // the Endorsement Hierarchy
15
16     BYTE      kdfResult[sizeof(TPMU_HA) * 2]; // Value produced by the KDF
17
18     TPM2B_DATA      sequence2B, handle2B;
19
20     // Get proof value
21     proof = HierarchyGetProof(contextBlob->hierarchy);
22
23     // Get sequence value in 2B format
24     sequence2B.t.size = sizeof(contextBlob->sequence);
25     MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence,
26               sizeof(contextBlob->sequence));
27
28     // Get handle value in 2B format
29     handle2B.t.size = sizeof(contextBlob->savedHandle);
30     MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle,
31               sizeof(contextBlob->savedHandle));
32
33     // Get the symmetric encryption key size
34     symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
35     symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
36     // Get the size of the IV for the algorithm
37     iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);
38
39     // KDFa to generate symmetric key and IV value
40     KDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, "CONTEXT", &sequence2B.b,
41          &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL);
42
43     // Copy part of the returned value as the key
44     MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size);
45
46     // Copy the rest as the IV
47     MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
48
49     return;
50 }

```

### 7.3.1.2 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

```

51 void
52 ComputeContextIntegrity(
53     TPMS_CONTEXT          *contextBlob,      // IN: context blob
54     TPM2B_DIGEST         *integrity        // OUT: integrity
55 )
56 {
57     HMAC_STATE           hmacState;
58     TPM2B_AUTH           *proof;
59     UINT16               integritySize;
60
61     // Get proof value
62     proof = HierarchyGetProof(contextBlob->hierarchy);
63
64     // Start HMAC
65     integrity->t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
66                                         &proof->b, &hmacState);
67
68     // Adding total reset counter
69     CryptUpdateDigestInt(&hmacState, sizeof(gp.totalResetCount),
70                         &gp.totalResetCount);
71
72     // Adding clearCount
73     if(contextBlob->savedHandle == 0x80000002)
74         CryptUpdateDigestInt(&hmacState, sizeof(gr.clearCount), &gr.clearCount);
75
76     // Adding sequence
77     CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->sequence),
78                         &contextBlob->sequence);
79
80     // Adding handle
81     CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->savedHandle),
82                         &contextBlob->savedHandle);
83
84     // Compute integrity size at the beginning of context blob
85     integritySize = sizeof(integrity->t.size)
86                   + CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
87
88     // Adding sensitive contextData, skip the leading integrity area
89     CryptUpdateDigest(&hmacState, contextBlob->contextBlob.t.size - integritySize,
90                     contextBlob->contextBlob.t.buffer + integritySize);
91
92     // Complete HMAC
93     CryptCompleteHMAC2B(&hmacState, &integrity->b);
94
95     return;
96 }

```

## 7.4 Policy Command Support (Policy\_spt.c)

```

1 #include "InternalRoutines.h"
2 #include "Policy_spt_fp.h"

```

### 7.4.1.1 ValidatePolicyID()

Validate *nonceTPM* parameter for TPM2\_PolicySigned(), and TPM2\_PolicySecret().

Error Returns	Meaning
TPM_RC_VALUE	if fails

```

3  TPM_RC
4  ValidatePolicyID(
5      TPM2B_NONCE      *nonceTPM,          // IN: nonceTPM
6      SESSION          *session           // IN: policy session
7  )
8  {
9      if(nonceTPM->t.size != 0)
10     {
11         if(!Memory2BEqual(&nonceTPM->b, &session->nonceTPM.b))
12             return TPM_RC_VALUE;
13     }
14     return TPM_RC_SUCCESS;
15 }

```

#### 7.4.1.2 ValidateExpiration()

Validate expiration parameter for TPM2\_PolicySigned() and TPM2\_PolicySecret()

Error Returns	Meaning
TPM_RC_VALUE	if fails

```

16  TPM_RC
17  ValidateExpiration(
18      UINT32            expiration,        // IN: expiration in millisecond
19      SESSION          *session           // IN: policy session
20  )
21  {
22      TPM_RC            result = TPM_RC_SUCCESS;
23
24      if(expiration != 0)
25      {
26          // Cannot compare time if clock stop advancing. A TPM_RC_NV_UNAVAILABLE
27          // or TPM_RC_NV_RATE error may be returned here.
28          result = NvIsAvailable();
29          if(result != TPM_RC_SUCCESS)
30              return result;
31
32          if((UINT64) expiration * 1000 < go.clock - session->startTime)
33              return TPM_RC_EXPIRED;
34      }
35
36      return TPM_RC_SUCCESS;
37 }

```

#### 7.4.1.3 UpdateTimeout()

Update timeout in a policy session

```

38  void
39  UpdateTimeout(
40      UINT64            timeout,          // IN: the new timeout value
41      SESSION          *session           // IN: the session
42  )
43  {
44      // If the timeout has not been set, then set it to the new value
45      if(session->timeOut == 0)
46          session->timeOut = timeout;

```



```

47     else if(session->timeOut > timeout)
48         session->timeOut = timeout;
49
50     return;
51 }

```

#### 7.4.1.4 PolicyUpdate()

Update policy hash Update the *policyDigest* in policy session by extending *policyRef* and *objectName* to it.

```

52 void
53 PolicyUpdate(
54     TPM_CC             commandCode,           // IN: command code
55     TPM2B_NAME         *name,                // IN: name of entity
56     TPM2B_NONCE        *ref,                // IN: the reference data
57     SESSION            *session              // IN/OUT: policy session to be updated
58 )
59 {
60     HASH_STATE          hashState;
61     UINT16              policyDigestSize;
62
63     // Start hash
64     policyDigestSize = CryptStartHash(session->authHashAlg, &hashState);
65
66     // policyDigest size should always be the digest size of session hash alg.
67     pAssert(session->u2.policyDigest.t.size == policyDigestSize);
68
69     // add old digest
70     CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);
71
72     // add commandCode
73     CryptUpdateDigestInt(&hashState, sizeof(commandCode), &commandCode);
74
75     // add name if applicable
76     if(name != NULL)
77         CryptUpdateDigest2B(&hashState, &name->b);
78
79     // Complete the digest and get the results
80     CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);
81
82     // Start second hash computation
83     CryptStartHash(session->authHashAlg, &hashState);
84
85     // add policyDigest
86     CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);
87
88     // add policyRef
89     if(ref != NULL)
90         CryptUpdateDigest2B(&hashState, &ref->b);
91
92     // Complete second digest
93     CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);
94
95     return;
96 }

```

### 7.5 NV Command Support (NV\_spt.c)

```

1  #include "InternalRoutines.h"
2  #include "NV_spt_fp.h"

```

Common routine for validating a read Used by TPM2\_NV\_Read(), TPM2\_NV\_ReadLock() and TPM2\_PolicyNV()

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

```

3  TPM_RC
4  NvReadAccessChecks (
5      TPM_HANDLE    authHandle,    // IN: the handle that provided the
6                                  // authorization
7      TPM_HANDLE    nvHandle       // IN: the handle of the NV index to be
8                                  // written
9  )
10 {
11     NV_INDEX        nvIndex;
12
13     // Get NV index info
14     NvGetIndexInfo(nvHandle, &nvIndex);
15
16     // If data is read locked, returns an error
17     if(nvIndex.publicArea.attributes.TPMA_NV_READLOCKED == SET)
18         return TPM_RC_NV_LOCKED;
19
20     // If the index has not been written, then the value cannot be read
21     if(nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
22         return TPM_RC_NV_UNINITIALIZED;
23
24     // If the authorization was provided by the owner or platform, then check
25     // that the attributes allow the read.  If the authorization handle
26     // is the same as the index, then the checks were made when the authorization
27     // was checked..
28     if(authHandle == TPM_RH_OWNER)
29     {
30         // If Owner provided auth then ONWERWRITE must be SET
31         if(! nvIndex.publicArea.attributes.TPMA_NV_OWNERREAD)
32             return TPM_RC_NV_AUTHORIZATION;
33     }
34     else if(authHandle == TPM_RH_PLATFORM)
35     {
36         // If Platform provided auth then PPWRITE must be SET
37         if(!nvIndex.publicArea.attributes.TPMA_NV_PPREAD)
38             return TPM_RC_NV_AUTHORIZATION;
39
40         // If neither Owner nor Platform provided auth, make sure that it was
41         // provided by this index.
42     }
43     else
44     { // make sure that the handles match
45         if(authHandle != nvHandle)
46             return TPM_RC_NV_AUTHORIZATION;
47
48         // If the hierarchy that the object was created is disabled, only
49         // another hierarchy handle can be used to authorize the access.
50         if( ( nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET
51             && g_phEnable == CLEAR)
52            || ( nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR
53             && gc.shEnable == CLEAR)
54         )
55             return TPM_RC_NV_AUTHORIZATION;
56     }
57

```

```

58     return TPM_RC_SUCCESS;
59 }

```

Common routine for validating a write Used by TPM2\_NV\_Write(), TPM2\_NV\_Increment(), TPM2\_SetBits(), and TPM2\_NV\_WriteLock()

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	Authorization fails
TPM_RC_NV_LOCKED	Write locked

```

60 TPM_RC
61 NvWriteAccessChecks(
62     TPM_HANDLE     authHandle,    // IN: the handle that provided the
63                                     // authorization
64     TPM_HANDLE     nvHandle      // IN: the handle of the NV index to be
65                                     // written
66 )
67 {
68     NV_INDEX       nvIndex;
69
70     // Get NV index info
71     NvGetIndexInfo(nvHandle, &nvIndex);
72
73     // If data is write locked, returns an error
74     if(nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED == SET)
75         return TPM_RC_NV_LOCKED;
76
77     // If the authorization was provided by the owner or platform, then check
78     // that the attributes allow the write. If the authorization handle
79     // is the same as the index, then the checks were made when the authorization
80     // was checked..
81     if(authHandle == TPM_RH_OWNER)
82     {
83         // If Owner provided auth then ONWERWRITE must be SET
84         if(! nvIndex.publicArea.attributes.TPMA_NV_OWNERWRITE)
85             return TPM_RC_NV_AUTHORIZATION;
86     }
87     else if(authHandle == TPM_RH_PLATFORM)
88     {
89         // If Platform provided auth then PPWRITE must be SET
90         if(!nvIndex.publicArea.attributes.TPMA_NV_PPWRITE)
91             return TPM_RC_NV_AUTHORIZATION;
92
93         // If neither Owner nor Platform provided auth, make sure that it was
94         // provided by this index.
95     }
96     else
97     { // make sure that the handles match
98         if(authHandle != nvHandle)
99             return TPM_RC_NV_AUTHORIZATION;
100
101         // If the hierarchy that the object was created is disabled, only
102         // another hierarchy handle can be used to authorize the access.
103         if( ( nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET
104             && g_phEnable == CLEAR)
105           || ( nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR
106             && gc.shEnable == CLEAR)
107         )
108             return TPM_RC_NV_AUTHORIZATION;
109     }
110
111     return TPM_RC_SUCCESS;
112 }

```

## 7.6 Object Command Support (Object\_spt.c)

```

1  #include "InternalRoutines.h"
2  #include "Object_spt_fp.h"
3  #include <Platform.h>

```

Check if the crypto sets in two public areas are equal

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

```

4  static TPM_RC
5  EqualCryptSet(
6      TPMT_PUBLIC      *publicArea1,          // IN: public area 1
7      TPMT_PUBLIC      *publicArea2          // IN: public area 2
8  )
9  {
10     UINT16            size1;
11     UINT16            size2;
12     BYTE              params1[sizeof(TPMU_PUBLIC_PARMS)];
13     BYTE              params2[sizeof(TPMU_PUBLIC_PARMS)];
14     BYTE              *buffer;
15
16     // Compare name hash
17     if(publicArea1->nameAlg != publicArea2->nameAlg)
18         return TPM_RC_HASH;
19
20     // Compare algorithm
21     if(publicArea1->type != publicArea2->type)
22         return TPM_RC_TYPE;
23
24     // TPMU_PUBLIC_PARMS field should be identical
25     buffer = params1;
26     size1 = TPMU_PUBLIC_PARMS_Marshal(&publicArea1->parameters, &buffer,
27                                     NULL, publicArea1->type);
28     buffer = params2;
29     size2 = TPMU_PUBLIC_PARMS_Marshal(&publicArea2->parameters, &buffer,
30                                     NULL, publicArea2->type);
31
32     if(size1 != size2 || !MemoryEqual(params1, params2, size1))
33         return TPM_RC_ASYMMETRIC;
34
35     return TPM_RC_SUCCESS;
36 }

```

### 7.6.1.1 AreAttributesForParent()

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

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

```

37  BOOL
38  AreAttributesForParent(
39      OBJECT      *parentObject          // IN: parent handle
40  )

```

```

41 {
42     if(!ObjectDataIsStorage(&parentObject->publicArea))
43         return FALSE;
44
45     // parent object must have both public and sensitive portion loaded
46     if(parentObject->attributes.publicOnly == SET)
47         return FALSE;
48
49     return TRUE;
50 }

```

### 7.6.1.2 SchemeChecks()

This function validates the schemes in the public area of an object. This function is called by TPM2\_LoadExternal() and *PublicAttributesValidation()*.

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public params
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 <i>decrypt</i> , <i>sign</i> , <i>restricted</i> and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unexpected object type; or non-duplicable storage key and its parent have different types

```

51 TPM_RC
52 SchemeChecks(
53     BOOL                load,                // IN: TRUE if load checks, FALSE if
54                                     //     TPM2_Create()
55     TPMI_DH_OBJECT      parentHandle,        // IN: input parent handle
56     TPMT_PUBLIC         *publicArea         // IN: public area of the object
57 )
58 {
59
60     // Checks for an asymmetric key
61     if(CryptIsAsymAlgorithm(publicArea->type))
62     {
63         TPMT_ASYM_SCHEME *keyScheme;
64         keyScheme = &publicArea->parameters.asymDetail.scheme;
65
66         // An asymmetric key can't be injected
67         // This is only checked when creating an object
68         if(!load && (publicArea->objectAttributes.sensitiveDataOrigin == CLEAR))
69             return TPM_RC_ATTRIBUTES;
70
71         if(load && !CryptAreKeySizesConsistent(publicArea))
72             return TPM_RC_KEY;
73
74         // Keys that are both signing and decrypting must have TPM_ALG_NULL
75         // for scheme
76         if( publicArea->objectAttributes.sign == SET
77            && publicArea->objectAttributes.decrypt == SET
78            && keyScheme->scheme != TPM_ALG_NULL)

```

```

79     return TPM_RC_SCHEME;
80
81     // A restrict sign key must have a non-NULL scheme
82     if(    publicArea->objectAttributes.restricted == SET
83         && publicArea->objectAttributes.sign == SET
84         && keyScheme->scheme == TPM_ALG_NULL)
85         return TPM_RC_SCHEME;
86
87     // Keys must have a valid sign or decrypt scheme, or a TPM_ALG_NULL
88     // scheme
89     if(    keyScheme->scheme != TPM_ALG_NULL
90         && (    (    publicArea->objectAttributes.sign == SET
91                 && !CryptIsSignScheme(keyScheme->scheme)
92             )
93             || (    publicArea->objectAttributes.decrypt == SET
94                 && !CryptIsDecryptScheme(keyScheme->scheme)
95             )
96         )
97     )
98     return TPM_RC_SCHEME;
99
100    // Special checks for an ECC key
101    if(publicArea->type == TPM_ALG_ECC)
102    {
103        TPM_ECC_CURVE    curveID = publicArea->parameters.eccDetail.curveID;
104        const TPMT_ECC_SCHEME *curveScheme = CryptGetCurveSignScheme(curveID);
105        // The curveId must be valid or the unmarshaling is busted.
106        pAssert(curveScheme != NULL);
107
108        // If the curveID requires a specific scheme, then the key must select
109        // the same scheme
110        if(curveScheme->scheme != TPM_ALG_NULL)
111        {
112            if(keyScheme->scheme != curveScheme->scheme)
113                return TPM_RC_SCHEME;
114            // The scheme can allow any hash, or not...
115            if(    curveScheme->details.any.hashAlg != TPM_ALG_NULL
116                && (    keyScheme->details.anySig.hashAlg
117                    != curveScheme->details.any.hashAlg
118                )
119            )
120                return TPM_RC_SCHEME;
121        }
122        // For now, the KDF must be TPM_ALG_NULL
123        if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
124            return TPM_RC_KDF;
125    }
126
127    // Checks for a storage key (restricted + decryption)
128    if(    publicArea->objectAttributes.restricted == SET
129        && publicArea->objectAttributes.decrypt == SET)
130    {
131        // A storage key must have a valid protection key
132        if(    publicArea->parameters.asymDetail.symmetric.algorithm
133            == TPM_ALG_NULL)
134            return TPM_RC_SYMMETRIC;
135
136        // A storage key must have a null scheme
137        if(publicArea->parameters.asymDetail.scheme.scheme != TPM_ALG_NULL)
138            return TPM_RC_SCHEME;
139
140        // A storage key must match its parent algorithms unless
141        // it is duplicable or a primary (including Temporary Primary Objects)
142        if(    HandleGetType(parentHandle) != TPM_HT_PERMANENT
143            && publicArea->objectAttributes.fixedParent == SET
144        )

```

```

145     {
146         // If the object to be created is a storage key, and is fixedParent,
147         // its crypto set has to match its parent's crypto set. TPM_RC_TYPE,
148         // TPM_RC_HASH or TPM_RC_ASYMMETRIC may be returned at this point
149         return EqualCryptSet(publicArea,
150                               &(ObjectGet(parentHandle)->publicArea));
151     }
152 }
153 else
154 {
155     // Non-storage keys must have TPM_ALG_NULL for the symmetric algorithm
156     if( publicArea->parameters.asymDetail.symmetric.algorithm
157         != TPM_ALG_NULL)
158         return TPM_RC_SYMMETRIC;
159
160     } // End of asymmetric decryption key checks
161 } // End of asymmetric checks
162
163 // Check for bit attributes
164 else if(publicArea->type == TPM_ALG_KEYEDHASH)
165 {
166     TPMT_KEYEDHASH_SCHEME *scheme
167     = &publicArea->parameters.keyedHashDetail.scheme;
168     // If both sign and decrypt are set the scheme must be TPM_ALG_NULL
169     // and the scheme selected when the key is used.
170     // If neither sign nor decrypt is set, the scheme must be TPM_ALG_NULL
171     // because this is a data object.
172     if( ( publicArea->objectAttributes.sign == SET
173         && publicArea->objectAttributes.decrypt == SET
174         )
175         || ( publicArea->objectAttributes.sign == CLEAR
176             && publicArea->objectAttributes.decrypt == CLEAR
177             )
178         )
179     {
180         if(scheme->scheme != TPM_ALG_NULL)
181             return TPM_RC_SCHEME;
182         return TPM_RC_SUCCESS;
183     }
184     // If this is a decryption key, make sure that is is XOR and that there
185     // is a KDF
186     else if(publicArea->objectAttributes.decrypt)
187     {
188         if( scheme->scheme != TPM_ALG_XOR
189             || scheme->details.xor.hashAlg == TPM_ALG_NULL)
190             return TPM_RC_SCHEME;
191         if(scheme->details.xor.kdf == TPM_ALG_NULL)
192             return TPM_RC_KDF;
193         return TPM_RC_SUCCESS;
194     }
195 }
196 // only supported signing scheme for keyedHash object is HMAC
197 if( scheme->scheme != TPM_ALG_HMAC
198     || scheme->details.hmac.hashAlg == TPM_ALG_NULL)
199     return TPM_RC_SCHEME;
200
201 // end of the checks for keyedHash
202 return TPM_RC_SUCCESS;
203 }
204 else if (publicArea->type == TPM_ALG_SYMCIPHER)
205 {
206     // Must be a decrypting key and may not be a signing key
207     if( publicArea->objectAttributes.decrypt == CLEAR
208         || publicArea->objectAttributes.sign == SET
209         )
210         return TPM_RC_ATTRIBUTES;

```

```

211     }
212     else
213         return TPM_RC_TYPE;
214
215     return TPM_RC_SUCCESS;
216 }

```

### 7.6.1.3 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()

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public params
TPM_RC_ATTRIBUTES	<i>fixedTPM</i> , <i>fixedParent</i> , or <i>encryptedDuplication</i> attributes are inconsistent between themselves or with those of the parent object; inconsistent <i>restricted</i> , <i>decrypt</i> and <i>sign</i> attributes; attempt to inject sensitive data for an asymmetric key; attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area
TPM_RC_SCHEME	inconsistent attributes <i>decrypt</i> , <i>sign</i> , <i>restricted</i> and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	<i>authPolicy</i> size does not match digest size of the name algorithm in <i>publicArea</i>
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

```

217 TPM_RC
218 PublicAttributesValidation(
219     BOOL                load,                // IN: TRUE if load checks, FALSE if
220                                     //     TPM2_Create()
221     TPMI_DH_OBJECT      parentHandle,        // IN: input parent handle
222     TPMT_PUBLIC         *publicArea         // IN: public area of the object
223 )
224 {
225     OBJECT              *parentObject = NULL;
226
227     if(HandleGetType(parentHandle) != TPM_HT_PERMANENT)
228         parentObject = ObjectGet(parentHandle);
229
230     // Check authPolicy digest consistency
231     if( publicArea->authPolicy.t.size != 0
232        && ( publicArea->authPolicy.t.size
233            != CryptGetHashDigestSize(publicArea->nameAlg)
234          )
235        )
236         return TPM_RC_SIZE;
237
238     // If the parent is fixedTPM (including a Primary Object) the object must have
239     // the same value for fixedTPM and fixedParent
240     if( parentObject == NULL
241        || parentObject->publicArea.objectAttributes.fixedTPM == SET)
242     {

```



```

243     if( publicArea->objectAttributes.fixedParent
244         != publicArea->objectAttributes.fixedTPM
245         )
246         return TPM_RC_ATTRIBUTES;
247     }
248     else
249         // The parent is not fixedTPM so the object can't be fixedTPM
250         if(publicArea->objectAttributes.fixedTPM == SET)
251             return TPM_RC_ATTRIBUTES;
252
253     // A restricted object cannot be both sign and decrypt and it can't be neither
254     // sign not decrypt
255     if ( publicArea->objectAttributes.restricted == SET
256         && ( publicArea->objectAttributes.decrypt
257             == publicArea->objectAttributes.sign)
258         )
259         return TPM_RC_ATTRIBUTES;
260
261     // A fixedTPM object can not have encryptedDuplication bit SET
262     if( publicArea->objectAttributes.fixedTPM == SET
263         && publicArea->objectAttributes.encryptedDuplication == SET)
264         return TPM_RC_ATTRIBUTES;
265
266     // If a parent object has fixedTPM CLEAR, the child must have the
267     // same encryptedDuplication value as parent.
268     // Primary objects are considered to have a fixedTPM parent (the seeds).
269     if(
270         ( parentObject != NULL
271           && parentObject->publicArea.objectAttributes.fixedTPM == CLEAR)
272         // Get here if parent is not fixed TPM
273         && ( publicArea->objectAttributes.encryptedDuplication
274             != parentObject->publicArea.objectAttributes.encryptedDuplication
275             )
276         )
277         return TPM_RC_ATTRIBUTES;
278     return SchemeChecks(load, parentHandle, publicArea);
279 }

```

#### 7.6.1.4 FillInCreationData()

Fill in creation data for an object.

```

280 void
281 FillInCreationData(
282     TPMI_DH_OBJECT      parentHandle,      // IN: handle of parent
283     TPMI_ALG_HASH       nameHashAlg,      // IN: name hash algorithm
284     TPML_PCR_SELECTION *creationPCR,      // IN: PCR selection
285     TPM2B_DATA          *outsideData,      // IN: outside data
286     TPM2B_CREATION_DATA *outCreation,      // OUT: creation data for output
287     TPM2B_DIGEST        *creationDigest   // OUT: creation digest
288 )
289 {
290     BYTE      creationBuffer[sizeof(TPMS_CREATION_DATA)];
291     BYTE      *buffer;
292     HASH_STATE hashState;
293
294     // Fill in TPMS_CREATION_DATA in outCreation
295
296     // Compute PCR digest
297     PCRComputeCurrentDigest(nameHashAlg, creationPCR,
298                             &outCreation->t.creationData.pcrDigest);
299
300     // Put back PCR selection list
301     outCreation->t.creationData.pcrSelect = *creationPCR;

```

```

302
303 // Get locality
304 outCreation->t.creationData.locality
305     = LocalityGetAttributes(_plat__LocalityGet());
306
307 outCreation->t.creationData.parentNameAlg = TPM_ALG_NULL;
308
309 // If the parent is either a primary seed or TPM_ALG_NULL, then the Name
310 // and QN of the parent are the parent's handle.
311 if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
312 {
313     BYTE *buffer = &outCreation->t.creationData.parentName.t.name[0];
314     outCreation->t.creationData.parentName.t.size =
315         TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
316
317     // Parent qualified name of a Temporary Object is the same as parent's
318     // name
319     MemoryCopy2B(&outCreation->t.creationData.parentQualifiedName.b,
320                 &outCreation->t.creationData.parentName.b);
321
322 }
323 else // Regular object
324 {
325     OBJECT *parentObject = ObjectGet(parentHandle);
326
327     // Set name algorithm
328     outCreation->t.creationData.parentNameAlg =
329         parentObject->publicArea.nameAlg;
330     // Copy parent name
331     outCreation->t.creationData.parentName = parentObject->name;
332
333     // Copy parent qualified name
334     outCreation->t.creationData.parentQualifiedName =
335         parentObject->qualifiedName;
336 }
337
338 // Copy outside information
339 outCreation->t.creationData.outsideInfo = *outsideData;
340
341 // Marshal creation data to canonical form
342 buffer = creationBuffer;
343 outCreation->t.size = TPMS_CREATION_DATA_Marshal(&outCreation->t.creationData,
344         &buffer, NULL);
345
346 // Compute hash for creation field in public template
347 creationDigest->t.size = CryptStartHash(nameHashAlg, &hashState);
348 CryptUpdateDigest(&hashState, outCreation->t.size, creationBuffer);
349 CryptCompleteHash2B(&hashState, &creationDigest->b);
350
351 return;
352 }

```

### 7.6.1.5 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

Return Value	Meaning
--------------	---------

```

353 static UINT16
354 GetIV2BSize(
355     TPM_HANDLE protectorHandle // IN: the protector handle
356 )
357 {

```

```

358     OBJECT          *protector = NULL; // Pointer to the protector object
359     TPM_ALG_ID      symAlg;
360     UINT16          keyBits;
361
362     // Determine the symmetric algorithm and size of key
363     if(protectorHandle == TPM_RH_NULL)
364     {
365         // Use the context encryption algorithm and key size
366         symAlg = CONTEXT_ENCRYPT_ALG;
367         keyBits = CONTEXT_ENCRYPT_KEY_BITS;
368     }
369     else
370     {
371         protector = ObjectGet(protectorHandle);
372         symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
373         keyBits= protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
374     }
375
376     // The IV size is a UINT16 size field plus the block size of the symmetric
377     // algorithm
378     return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
379 }

```

#### 7.6.1.6 GetSeedForKDF()

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

```

380     TPM2B_SEED*
381     GetSeedForKDF(
382         TPM_HANDLE      protectorHandle, // IN: the protector handle
383         TPM2B_SEED      *seedIn        // IN: the optional input seed
384     )
385 {
386     OBJECT          *protector = NULL; // Pointer to the protector
387
388     // Get seed for encryption key. Use input seed if provided.
389     // Otherwise, using protector object's seedValue. TPM_RH_NULL is the only
390     // exception that we may not have a loaded object as protector. In such a
391     // case, use nullProof as seed.
392     if(seedIn != NULL)
393     {
394         return seedIn;
395     }
396     else
397     {
398         if(protectorHandle == TPM_RH_NULL)
399         {
400             return (TPM2B_SEED *) &gr.nullProof;
401         }
402         else
403         {
404             protector = ObjectGet(protectorHandle);
405             return (TPM2B_SEED *) &protector->sensitive.seedValue;
406         }
407     }
408 }

```

#### 7.6.1.7 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.

```

409 static void
410 ComputeProtectionKeyParms(
411     TPM_HANDLE      protectorHandle,    // IN: the protector handle
412     TPM_ALG_ID      hashAlg,           // IN: hash algorithm for KDFa
413     TPM2B_NAME      *name,             // IN: name of the object
414     TPM2B_SEED      *seedIn,          // IN: optional seed for duplication
415                                     // blob. For non duplication blob,
416                                     // this parameter should be NULL
417     TPM_ALG_ID      *symAlg,           // OUT: the symmetric algorithm
418     UINT16          *keyBits,          // OUT: the symmetric key size in bits
419     TPM2B_SYM_KEY   *symKey            // OUT: the symmetric key
420 )
421 {
422     TPM2B_SEED      *seed = NULL;
423     OBJECT          *protector = NULL; // Pointer to the protector
424
425     // Determine the algorithms for the KDF and the encryption/decryption
426     // For TPM_RH_NULL, using context settings
427     if(protectorHandle == TPM_RH_NULL)
428     {
429         // Use the context encryption algorithm and key size
430         *symAlg = CONTEXT_ENCRYPT_ALG;
431         symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
432         *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
433     }
434     else
435     {
436         TPMT_SYM_DEF_OBJECT *symDef;
437         protector = ObjectGet(protectorHandle);
438         symDef = &protector->publicArea.parameters.asymDetail.symmetric;
439         *symAlg = symDef->algorithm;
440         *keyBits = symDef->keyBits.sym;
441         symKey->t.size = (*keyBits + 7) / 8;
442     }
443
444     // Get seed for KDF
445     seed = GetSeedForKDF(protectorHandle, seedIn);
446
447     // KDFa to generate symmetric key and IV value
448     KDFa(hashAlg, (TPM2B *)seed, "STORAGE", (TPM2B *)name, NULL,
449          symKey->t.size * 8, symKey->t.buffer, NULL);
450
451     return;
452 }

```

### 7.6.1.8 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.

```

453 static void
454 ComputeOuterIntegrity(
455     TPM2B_NAME      *name,             // IN: the name of the object
456     TPM_HANDLE      protectorHandle,    // IN: The handle of the object
457                                     // that provides protection. For
458                                     // object, it is parent handle.
459                                     // For credential, it is the handle
460                                     // of encrypt object. For a

```

```

461                                     //      Temporary Object, it is
462                                     //      TPM_RH_NULL
463     TPMI_ALG_HASH      hashAlg,      // IN: algorithm to use for integrity
464     TPM2B_SEED         *seedIn,      // IN: an external seed may be
465                                     //      provided for duplication blob.
466                                     //      For non duplication blob, this
467                                     //      parameter should be NULL
468     UINT32             sensitiveSize, // IN: size of the marshaled sensitive
469                                     //      data
470     BYTE               *sensitiveData, // IN: sensitive area
471     TPM2B_DIGEST       *integrity     // OUT: integrity
472 )
473 {
474     HMAC_STATE         hmacState;
475
476     TPM2B_DIGEST       hmacKey;
477     TPM2B_SEED         *seed = NULL;
478
479     // Get seed for KDF
480     seed = GetSeedForKDF(protectorHandle, seedIn);
481
482     // Determine the HMAC key bits
483     hmacKey.t.size = CryptGetHashDigestSize(hashAlg);
484
485     // KDFa to generate HMAC key
486     KDFa(hashAlg, (TPM2B *)seed, "INTEGRITY", NULL, NULL,
487          hmacKey.t.size * 8, hmacKey.t.buffer, NULL);
488
489     // Start HMAC and get the size of the digest which will become the integrity
490     integrity->t.size = CryptStartHMAC2B(hashAlg, &hmacKey.b, &hmacState);
491
492     // Adding the marshaled sensitive area to the integrity value
493     CryptUpdateDigest(&hmacState, sensitiveSize, sensitiveData);
494
495     // Adding name
496     CryptUpdateDigest2B(&hmacState, (TPM2B *)name);
497
498     // Compute HMAC
499     CryptCompleteHMAC2B(&hmacState, &integrity->b);
500
501     return;
502 }

```

### 7.6.1.9 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap

```

503 static void
504 ComputeInnerIntegrity(
505     TPM_ALG_ID         hashAlg,      // IN: hash algorithm for inner wrap
506     TPM2B_NAME         *name,        // IN: the name of the object
507     UINT16             dataSize,     // IN: the size of sensitive data
508     BYTE               *sensitiveData, // IN: sensitive data
509     TPM2B_DIGEST       *integrity     // OUT: inner integrity
510 )
511 {
512     HASH_STATE         hashState;
513
514     // Start hash and get the size of the digest which will become the integrity
515     integrity->t.size = CryptStartHash(hashAlg, &hashState);
516
517     // Adding the marshaled sensitive area to the integrity value
518     CryptUpdateDigest(&hashState, dataSize, sensitiveData);
519 }

```

```

520     // Adding name
521     CryptUpdateDigest2B(&hashState, &name->b);
522
523     // Compute hash
524     CryptCompleteHash2B(&hashState, &integrity->b);
525
526     return;
527
528 }

```

#### 7.6.1.10 ProduceInnerIntegrity()

This function produces an inner integrity for regular private, credential or duplication blob. It requires the sensitive data being marshaled to the *innerBuffer*, with the leading bytes reserved for integrity hash. It assumes the sensitive data starts at address (*innerBuffer* + integrity size). This function integrity at the beginning of the inner buffer. It returns the total size of buffer with the inner wrap.

```

529 static UINT16
530 ProduceInnerIntegrity(
531     TPM2B_NAME          *name,           // IN: the name of the object
532     TPM_ALG_ID          hashAlg,        // IN: hash algorithm for inner wrap
533     UINT16              dataSize,       // IN: the size of sensitive data,
534                                     // excluding the leading integrity
535                                     // buffer size
536     BYTE                *innerBuffer    // IN/OUT: inner buffer with
537                                     // sensitive data in it. At
538                                     // input, the leading bytes of
539                                     // this buffer is reserved for
540                                     // integrity
541 )
542 {
543     BYTE                *sensitiveData; // pointer to the sensitive data
544
545     TPM2B_DIGEST        integrity;
546     UINT16              integritySize;
547     BYTE                *buffer;        // Auxiliary buffer pointer
548
549     // sensitiveData points to the beginning of sensitive data in innerBuffer
550     integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
551     sensitiveData = innerBuffer + integritySize;
552
553     ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);
554
555     // Add integrity at the beginning of inner buffer
556     buffer = innerBuffer;
557     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
558
559     return dataSize + integritySize;
560 }

```

#### 7.6.1.11 CheckInnerIntegrity()

This function checks integrity of inner blob.

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

```

561 static TPM_RC
562 CheckInnerIntegrity(

```

```

563     TPM2B_NAME      *name,           // IN: the name of the object
564     TPM_ALG_ID      hashAlg,         // IN: hash algorithm for inner wrap
565     UINT16           dataSize,        // IN: the size of sensitive data,
566                                     // including the leading integrity
567                                     // buffer size
568     BYTE             *innerBuffer     // IN/OUT: inner buffer with
569                                     // sensitive data in it
570 )
571 {
572     TPM_RC           result;
573
574     TPM2B_DIGEST     integrity;
575     TPM2B_DIGEST     integrityToCompare;
576     BYTE             *buffer;         // Auxiliary buffer pointer
577     INT32            size;
578
579     // Unmarshal integrity
580     buffer = innerBuffer;
581     size = (INT32) dataSize;
582     result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
583     if(result != TPM_RC_SUCCESS)
584         return result;
585
586     // Compute integrity to compare
587     ComputeInnerIntegrity(hashAlg, name, (UINT16) size, buffer,
588                           &integrityToCompare);
589
590     // Compare outer blob integrity
591     if(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
592         return TPM_RC_INTEGRITY;
593
594     return TPM_RC_SUCCESS;
595 }

```

#### 7.6.1.12 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
599                                     // that provides protection. For
600                                     // object, it is parent handle.
601                                     // For credential, it is the handle
602                                     // of encrypt object.
603         TPM2B_NAME     *name,         // IN: the name of the object
604         TPM_ALG_ID     hashAlg,       // IN: hash algorithm for outer wrap
605         TPM2B_SEED     *seed,         // IN: an external seed may be
606                                     // provided for duplication blob.
607                                     // For non duplication blob, this
608                                     // parameter should be NULL
609         BOOL           useIV,         // IN: indicate if an IV is used
610         UINT16         dataSize,      // IN: the size of sensitive data,
611                                     // excluding the leading integrity
612                                     // buffer size or the optional iv
613                                     // size

```

```

614     BYTE          *outerBuffer          // IN/OUT: outer buffer with
615                                     // sensitive data in it
616 )
617 {
618     TPM_ALG_ID     symAlg;
619     UINT16         keyBits;
620     TPM2B_SYM_KEY  symKey;
621     TPM2B_IV       ivRNG;              // IV from RNG
622     TPM2B_IV       *iv = NULL;
623     UINT16         ivSize = 0;        // size of iv area, including the size field
624
625     BYTE          *sensitiveData; // pointer to the sensitive data
626
627     TPM2B_DIGEST  integrity;
628     UINT16         integritySize;
629     BYTE          *buffer;           // Auxiliary buffer pointer
630
631     // Compute the beginning of sensitive data. The outer integrity should
632     // always exist if this function is called to make an outer wrap
633     integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
634     sensitiveData = outerBuffer + integritySize;
635
636     // If iv is used, adjust the pointer of sensitive data and add iv before it
637     if(useIV)
638     {
639         ivSize = GetIV2BSize(protector);
640
641         // Generate IV from RNG. The iv data size should be the total IV area
642         // size minus the size of size field
643         ivRNG.t.size = ivSize - sizeof(UINT16);
644         CryptGenerateRandom(ivRNG.t.size, ivRNG.t.buffer);
645
646         // Marshal IV to buffer
647         buffer = sensitiveData;
648         TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
649
650         // adjust sensitive data starting after IV area
651         sensitiveData += ivSize;
652
653         // Use iv for encryption
654         iv = &ivRNG;
655     }
656
657     // Compute symmetric key parameters for outer buffer encryption
658     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
659                               &symAlg, &keyBits, &symKey);
660     // Encrypt inner buffer in place
661     CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
662                           TPM_ALG_CFB, symKey.t.buffer, iv, dataSize,
663                           sensitiveData);
664
665     // Compute outer integrity. Integrity computation includes the optional IV
666     // area
667     ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
668                           outerBuffer + integritySize, &integrity);
669
670     // Add integrity at the beginning of outer buffer
671     buffer = outerBuffer;
672     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
673
674     // return the total size in outer wrap
675     return dataSize + integritySize + ivSize;
676
677 }

```



### 7.6.1.13 UnwrapOuter()

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

- d) check integrity of outer blob
- e) decrypt outer blob

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

```

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

```

```

726     return TPM_RC_INTEGRITY;
727
728     // Get the symmetric algorithm parameters used for encryption
729     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
730                               &symAlg, &keyBits, &symKey);
731
732     // Retrieve IV if it is used
733     if(useIV)
734     {
735         result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
736         if(result != TPM_RC_SUCCESS)
737             return result;
738         // The input iv size for CFB must match the encryption algorithm block
739         // size
740         if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
741             return TPM_RC_VALUE;
742         iv = &ivIn;
743     }
744
745     // Decrypt private in place
746     CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
747                           TPM_ALG_CFB, symKey.t.buffer, iv,
748                           (UINT16) size, sensitiveData);
749
750     return TPM_RC_SUCCESS;
751 }
752

```

#### 7.6.1.14 SensitiveToPrivate

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

- f) marshal TPM2B\_SENSITIVE structure into the buffer of TPM2B\_PRIVATE
- g) apply encryption to the sensitive area.
- h) apply outer integrity computation.

```

753 void
754 SensitiveToPrivate(
755     TPMT_SENSITIVE      *sensitive,    // IN: sensitive structure
756     TPM2B_NAME          *name,         // IN: the name of the object
757     TPM_HANDLE          parentHandle,  // IN: The parent's handle
758     TPM_ALG_ID          nameAlg,       // IN: hash algorithm in public
759                                     // area. This parameter is used
760                                     // when parentHandle is NULL, in
761                                     // which case the object is
762                                     // temporary.
763     TPM2B_PRIVATE       *outPrivate    // OUT: output private structure
764 )
765 {
766     BYTE                *buffer;       // Auxiliary buffer pointer
767     BYTE                *sensitiveData; // pointer to the sensitive data
768     UINT16              dataSize;      // data blob size
769     TPMI_ALG_HASH       hashAlg;       // hash algorithm for integrity
770     UINT16              integritySize;
771     UINT16              ivSize;
772
773     pAssert(name != NULL && name->t.size != 0);
774
775     // Find the hash algorithm for integrity computation
776     if(parentHandle == TPM_RH_NULL)
777     {
778         // For Temporary Object, using self name algorithm
779         hashAlg = nameAlg;

```

```

780     }
781     else
782     {
783         // Otherwise, using parent's name algorithm
784         hashAlg = ObjectGetNameAlg(parentHandle);
785     }
786
787     // Starting of sensitive data without wrappers
788     sensitiveData = outPrivate->t.buffer;
789
790     // Compute the integrity size
791     integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
792
793     // Reserve space for integrity
794     sensitiveData += integritySize;
795
796     // Get iv size
797     ivSize = GetIV2BSize(parentHandle);
798
799     // Reserve space for iv
800     sensitiveData += ivSize;
801
802     // Marshal sensitive area, leaving the leading 2 bytes for size
803     buffer = sensitiveData + sizeof(UINT16);
804     dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
805
806     // Adding size before the data area
807     buffer = sensitiveData;
808     UINT16_Marshal(&dataSize, &buffer, NULL);
809
810     // Adjust the dataSize to include the size field
811     dataSize += sizeof(UINT16);
812
813     // Adjust the pointer to inner buffer including the iv
814     sensitiveData = outPrivate->t.buffer + ivSize;
815
816     //Produce outer wrap, including encryption and HMAC
817     outPrivate->t.size = ProduceOuterWrap(parentHandle, name, hashAlg, NULL,
818                                         TRUE, dataSize, outPrivate->t.buffer);
819
820     return;
821 }

```

#### 7.6.1.15 PrivateToSensitive()

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

- i) check the integrity HMAC of the input private area
- j) decrypt the private buffer
- k) unmarshal TPMT\_SENSITIVE structure into the buffer of TPMT\_SENSITIVE

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 <i>iV</i> of the correct size

```

822 TPM_RC
823 PrivateToSensitive(
824     TPM2B_PRIVATE          *inPrivate,    // IN: input private structure

```

```

825     TPM2B_NAME          *name,           // IN: the name of the object
826     TPM_HANDLE         parentHandle,    // IN: The parent's handle
827     TPM_ALG_ID         nameAlg,         // IN: hash algorithm in public
828                                     // area. It is passed separately
829                                     // because we only pass name,
830                                     // rather than the whole public
831                                     // area of the object. This
832                                     // parameter is used in
833                                     // the following two cases: 1.
834                                     // primary objects. 2. duplication
835                                     // blob with inner wrap. In other
836                                     // cases, this parameter will be
837                                     // ignored
838     TPMT_SENSITIVE     *sensitive       // OUT: sensitive structure
839 )
840 {
841     TPM_RC              result;
842
843     BYTE                *buffer;
844     INT32               size;
845     BYTE                *sensitiveData; // pointer to the sensitive data
846     UINT16              dataSize;
847     UINT16              dataSizeInput;
848     TPMI_ALG_HASH       hashAlg;        // hash algorithm for integrity
849     OBJECT              *parent = NULL;
850
851     UINT16              integritySize;
852     UINT16              ivSize;
853
854     // Make sure that name is provided
855     pAssert(name != NULL && name->t.size != 0);
856
857     // Find the hash algorithm for integrity computation
858     if(parentHandle == TPM_RH_NULL)
859     {
860         // For Temporary Object, using self name algorithm
861         hashAlg = nameAlg;
862     }
863     else
864     {
865         // Otherwise, using parent's name algorithm
866         hashAlg = ObjectGetNameAlg(parentHandle);
867     }
868
869     // unwrap outer
870     result = UnwrapOuter(parentHandle, name, hashAlg, NULL, TRUE,
871                          inPrivate->t.size, inPrivate->t.buffer);
872     if(result != TPM_RC_SUCCESS)
873         return result;
874
875     // Compute the inner integrity size.
876     integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
877
878     // Get iv size
879     ivSize = GetIV2BSize(parentHandle);
880
881     // The starting of sensitive data and data size without outer wrapper
882     sensitiveData = inPrivate->t.buffer + integritySize + ivSize;
883     dataSize = inPrivate->t.size - integritySize - ivSize;
884
885     // Unmarshal input data size
886     buffer = sensitiveData;
887     size = (INT32) dataSize;
888     result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
889     if(result != TPM_RC_SUCCESS)
890         return result;

```

```

891     if((dataSizeInput + sizeof(UINT16)) != dataSize)
892         return TPM_RC_SENSITIVE;
893
894     // Unmarshal sensitive buffer to sensitive structure
895     result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
896     if(result != TPM_RC_SUCCESS || size != 0)
897     {
898         if(parent != NULL && parent->publicArea.objectAttributes.fixedTPM == SET)
899             FAIL(TPM_RC_FAILURE);
900         else
901             return TPM_RC_SENSITIVE;
902     }
903
904     // Always remove trailing zeros at load so that it is not necessary to check
905     // each time auth is checked.
906     MemoryRemoveTrailingZeros(&(sensitive->authValue));
907     return TPM_RC_SUCCESS;
908 }

```

### 7.6.1.16 SensitiveToDuplicate()

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

- l) marshal TPMT\_SENSITIVE structure into the buffer of TPM2B\_PRIVATE
- m) apply inner wrap to the sensitive area if required
- n) apply outer wrap if required

```

909 void
910 SensitiveToDuplicate(
911     TPMT_SENSITIVE *sensitive, // IN: sensitive structure
912     TPM2B_NAME *name, // IN: the name of the object
913     TPM_HANDLE parentHandle, // IN: The new parent's handle
914     TPM_ALG_ID nameAlg, // IN: hash algorithm in public
915     // area. It is passed separately
916     // because we only pass name,
917     // rather than the whole public
918     // area of the object.
919     TPM2B_SEED *seed, // IN: the external seed.
920     // If external seed is provided
921     // with size of 0, no outer wrap
922     // should be applied to duplication
923     // blob.
924     TPMT_SYM_DEF_OBJECT *symDef, // IN: Symmetric key definition.
925     // If the symmetric key algorithm
926     // is NULL, no inner wrap should be
927     // applied
928     TPM2B_DATA *innerSymKey, // IN: a symmetric key may be
929     // provided to encrypt the inner
930     // wrap of a duplication blob.
931     TPM2B_PRIVATE *outPrivate // OUT: output private structure
932 )
933 {
934     BYTE *buffer; // Auxiliary buffer pointer
935     BYTE *sensitiveData; // pointer to the sensitive data
936     TPMI_ALG_HASH outerHash = TPM_ALG_NULL; // The hash algorithm for outer wrap
937     TPMI_ALG_HASH innerHash = TPM_ALG_NULL; // The hash algorithm for inner wrap
938     UINT16 dataSize; // data blob size
939     BOOL doInnerWrap = FALSE;
940     BOOL doOuterWrap = FALSE;
941
942     // Make sure that name is provided
943     pAssert(name != NULL && name->t.size != 0);
944

```

```

945 // Make sure symDef and innerSymKey are not NULL
946 pAssert(symDef != NULL && innerSymKey != NULL);
947
948 // Starting of sensitive data without wrappers
949 sensitiveData = outPrivate->t.buffer;
950
951 // Find out if inner wrap is required
952 if(symDef->algorithm != TPM_ALG_NULL)
953 {
954     doInnerWrap = TRUE;
955     // Use self nameAlg as inner hash algorithm
956     innerHash = nameAlg;
957     // Adjust sensitive data pointer
958     sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
959 }
960
961 // Find out if outer wrap is required
962 if(seed->t.size != 0)
963 {
964     doOuterWrap = TRUE;
965     // Use parent nameAlg as outer hash algorithm
966     outerHash = ObjectGetNameAlg(parentHandle);
967     // Adjust sensitive data pointer
968     sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
969 }
970
971 // Marshal sensitive area, leaving the leading 2 bytes for size
972 buffer = sensitiveData + sizeof(UINT16);
973 dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
974
975 // Adding size before the data area
976 buffer = sensitiveData;
977 UINT16_Marshal(&dataSize, &buffer, NULL);
978
979 // Adjust the dataSize to include the size field
980 dataSize += sizeof(UINT16);
981
982 // Apply inner wrap for duplication blob. It includes both integrity and
983 // encryption
984 if(doInnerWrap)
985 {
986     BYTE *innerBuffer = NULL;
987     BOOL symKeyInput = TRUE;
988     innerBuffer = outPrivate->t.buffer;
989     // Skip outer integrity space
990     if(doOuterWrap)
991         innerBuffer += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
992     dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
993                                     innerBuffer);
994
995     // Generate inner encryption key if needed
996     if(innerSymKey->t.size == 0)
997     {
998         innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
999         CryptGenerateRandom(innerSymKey->t.size, innerSymKey->t.buffer);
1000
1001         // TPM generates symmetric encryption. Set the flag to FALSE
1002         symKeyInput = FALSE;
1003     }
1004     else
1005     {
1006         // assume the input key size should matches the symmetric definition
1007         pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1008     }
1009 }
1010

```

```

1011     // Encrypt inner buffer in place
1012     CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
1013                          symDef->keyBits.sym, TPM_ALG_CFB,
1014                          innerSymKey->t.buffer, NULL, dataSize,
1015                          innerBuffer);
1016
1017     // If the symmetric encryption key is imported, clear the buffer for
1018     // output
1019     if(symKeyInput)
1020         innerSymKey->t.size = 0;
1021 }
1022
1023 // Apply outer wrap for duplication blob. It includes both integrity and
1024 // encryption
1025 if(doOuterWrap)
1026 {
1027     dataSize = ProduceOuterWrap(parentHandle, name, outerHash, seed, FALSE,
1028                                dataSize, outPrivate->t.buffer);
1029 }
1030
1031 // Data size for output
1032 outPrivate->t.size = dataSize;
1033
1034 return;
1035 }

```

#### 7.6.1.17 DuplicateToSensitive()

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

- o) check the integrity HMAC of the input private area
- p) decrypt the private buffer
- q) unmarshal TPMT\_SENSITIVE structure into the buffer of TPMT\_SENSITIVE

Error Returns	Meaning
TPM_RC_INSUFFICIENT	unmarshaling sensitive data from <i>inPrivate</i> failed
TPM_RC_INTEGRITY	<i>inPrivate</i> data integrity is broken
TPM_RC_SIZE	unmarshaling sensitive data from <i>inPrivate</i> failed

```

1036 TPM_RC
1037 DuplicateToSensitive(
1038     TPM2B_PRIVATE      *inPrivate,    // IN: input private structure
1039     TPM2B_NAME         *name,        // IN: the name of the object
1040     TPM_HANDLE         parentHandle,  // IN: The parent's handle
1041     TPM_ALG_ID         nameAlg,      // IN: hash algorithm in public
1042                                     // area.
1043     TPM2B_SEED         *seed,        // IN: an external seed may be
1044                                     // provided.
1045                                     // If external seed is provided
1046                                     // with size of 0, no outer wrap
1047                                     // is applied
1048     TPMT_SYM_DEF_OBJECT *symDef,     // IN: Symmetric key definition.
1049                                     // If the symmetric key algorithm
1050                                     // is NULL, no inner wrap is
1051                                     // applied
1052     TPM2B_DATA         *innerSymKey, // IN: a symmetric key may be
1053                                     // provided to decrypt the inner
1054                                     // wrap of a duplication blob.
1055     TPMT_SENSITIVE     *sensitive    // OUT: sensitive structure

```

```

1056 )
1057 {
1058     TPM_RC         result;
1059
1060     BYTE           *buffer;
1061     INT32          size;
1062     BYTE           *sensitiveData; // pointer to the sensitive data
1063     UINT16         dataSize;
1064     UINT16         dataSizeInput;
1065
1066     // Make sure that name is provided
1067     pAssert(name != NULL && name->t.size != 0);
1068
1069     // Make sure symDef and innerSymKey are not NULL
1070     pAssert(symDef != NULL && innerSymKey != NULL);
1071
1072     // Starting of sensitive data
1073     sensitiveData = inPrivate->t.buffer;
1074     dataSize = inPrivate->t.size;
1075
1076     // Find out if inner wrap is applied
1077     if(seed->t.size != 0)
1078     {
1079         TPMI_ALG_HASH    outerHash = TPM_ALG_NULL;
1080
1081         // Use parent nameAlg as outer hash algorithm
1082         outerHash = ObjectGetNameAlg(parentHandle);
1083         result = UnwrapOuter(parentHandle, name, outerHash, seed, FALSE,
1084                             dataSize, sensitiveData);
1085         if(result != TPM_RC_SUCCESS)
1086             return result;
1087
1088         // Adjust sensitive data pointer and size
1089         sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1090         dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1091     }
1092     // Find out if inner wrap is applied
1093     if(symDef->algorithm != TPM_ALG_NULL)
1094     {
1095         TPMI_ALG_HASH    innerHash = TPM_ALG_NULL;
1096
1097         // assume the input key size should matches the symmetric definition
1098         pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1099
1100         // Decrypt inner buffer in place
1101         CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
1102                              symDef->keyBits.sym, TPM_ALG_CFB,
1103                              innerSymKey->t.buffer, NULL, dataSize,
1104                              sensitiveData);
1105
1106         // Use self nameAlg as inner hash algorithm
1107         innerHash = nameAlg;
1108
1109         // Check inner integrity
1110         result = CheckInnerIntegrity(name, innerHash, dataSize, sensitiveData);
1111         if(result != TPM_RC_SUCCESS)
1112             return result;
1113
1114         // Adjust sensitive data pointer and size
1115         sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
1116         dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
1117     }
1118
1119     // Unmarshal input data size
1120     buffer = sensitiveData;
1121     size = (INT32) dataSize;

```



```

1122     result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1123     if(result != TPM_RC_SUCCESS)
1124         return result;
1125     if((dataSizeInput + sizeof(UINT16)) != dataSize)
1126         return TPM_RC_SIZE;
1127     // Unmarshal sensitive buffer to sensitive structure
1128     result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1129     if(result != TPM_RC_SUCCESS)
1130         return result;
1131     if(size != 0)
1132         return TPM_RC_SIZE;
1133
1134     // Always remove trailing zeros at load so that it is not necessary to check
1135     // each time auth is checked.
1136     MemoryRemoveTrailingZeros(&(sensitive->authValue));
1137     return TPM_RC_SUCCESS;
1138 }

```

### 7.6.1.18 SecretToCredential

This function prepare the credential blob from a secret (a TPM2B\_DIGEST) The operations in this function:

- r) marshal TPM2B\_DIGEST structure into the buffer of TPM2B\_ID\_OBJECT
- s) encrypt the private buffer, excluding the leading integrity HMAC area
- t) compute integrity HMAC and append to the beginning of the buffer.
- u) Set the total size of TPM2B\_ID\_OBJECT buffer

```

1139 void
1140 SecretToCredential(
1141     TPM2B_DIGEST      *secret,          // IN: secret information
1142     TPM2B_NAME        *name,           // IN: the name of the object
1143     TPM2B_SEED        *seed,          // IN: an external seed.
1144     TPM_HANDLE        protector,       // IN: The protector's handle
1145     TPM2B_ID_OBJECT   *outIDObject     // OUT: output credential
1146 )
1147 {
1148     BYTE              *buffer;         // Auxiliary buffer pointer
1149     BYTE              *sensitiveData; // pointer to the sensitive data
1150     TPMI_ALG_HASH     outerHash;      // The hash algorithm for outer wrap
1151     UINT16            dataSize;       // data blob size
1152
1153     pAssert(secret != NULL && outIDObject != NULL);
1154
1155     // use protector's name algorithm as outer hash
1156     outerHash = ObjectGetNameAlg(protector);
1157
1158     // Marshal secret area to credential buffer, leave space for integrity
1159     sensitiveData = outIDObject->t.credential
1160                   + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1161
1162     // Marshal secret area
1163     buffer = sensitiveData;
1164     dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1165
1166     // Apply outer wrap
1167     outIDObject->t.size = ProduceOuterWrap(protector,
1168                                           name,
1169                                           outerHash,
1170                                           seed,
1171                                           FALSE,
1172                                           dataSize,

```

```

1173                                     outIDObject->t.credential);
1174     return;
1175 }

```

### 7.6.1.19 CredentialToSecret()

Unwrap a credential. Check the integrity, decrypt and retrieve data to a TPM2B\_DIGEST structure. The operations in this function:

- v) check the integrity HMAC of the input credential area
- w) decrypt the credential buffer
- x) unmarshal TPM2B\_DIGEST structure into the buffer of TPM2B\_DIGEST

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

```

1176 TPM_RC
1177 CredentialToSecret(
1178     TPM2B_ID_OBJECT      *inIDObject,    // IN: input credential blob
1179     TPM2B_NAME           *name,          // IN: the name of the object
1180     TPM2B_SEED           *seed,          // IN: an external seed.
1181     TPM_HANDLE           protector,      // IN: The protector's handle
1182     TPM2B_DIGEST         *secret         // OUT: secret information
1183 )
1184 {
1185     TPM_RC      result;
1186     BYTE        *buffer;
1187     INT32       size;
1188     TPML_ALG_HASH outerHash;           // The hash algorithm for outer wrap
1189     BYTE        *sensitiveData;       // pointer to the sensitive data
1190     UINT16      dataSize;
1191
1192     // use protector's name algorithm as outer hash
1193     outerHash = ObjectGetNameAlg(protector);
1194
1195     // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1196     result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
1197                         inIDObject->t.size, inIDObject->t.credential);
1198     if(result != TPM_RC_SUCCESS)
1199         return result;
1200
1201     // Compute the beginning of sensitive data
1202     sensitiveData = inIDObject->t.credential
1203                   + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1204     dataSize = inIDObject->t.size
1205               - (sizeof(UINT16) + CryptGetHashDigestSize(outerHash));
1206
1207     // Unmarshal secret buffer to TPM2B_DIGEST structure
1208     buffer = sensitiveData;
1209     size = (INT32) dataSize;
1210     result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1211     if(result != TPM_RC_SUCCESS)
1212         return result;
1213     if(size != 0)
1214         return TPM_RC_SIZE;
1215
1216     return TPM_RC_SUCCESS;

```

1217 }

## 8 Subsystem

### 8.1 CommandAudit.c

#### 8.1.1 Introduction

This file contains the functions that support command audit.

#### 8.1.2 Includes

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

#### 8.1.3 Functions

##### 8.1.3.1 CommandAuditPreInstall\_Init()

This function initializes the command audit list. This function is simulates the behavior of manufacturing. A function is used instead of a structure definition because this is easier than figuring out the initialization value for a bit array.

This function would not be implemented outside of a manufacturing or simulation environment.

```
2 void
3 CommandAuditPreInstall_Init(void)
4 {
5     // Clear all the audit commands
6     MemorySet(gp.auditComands, 0x00,
7             ((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8);
8
9     // TPM_CC_SetCommandCodeAuditStatus always being audited
10    if(CommandIsImplemented(TPM_CC_SetCommandCodeAuditStatus))
11        CommandAuditSet(TPM_CC_SetCommandCodeAuditStatus);
12
13    // Set initial command audit hash algorithm to be context integrity hash
14    // algorithm
15    gp.auditHashAlg = CONTEXT_INTEGRITY_HASH_ALG;
16
17    // Set up audit counter to be 0
18    gp.auditCounter = 0;
19
20    // Write command audit persistent data to NV
21    NvWriteReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
22    NvWriteReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
23    NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
24
25    return;
26 }
```

##### 8.1.3.2 CommandAuditStartup()

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

```
27 void
28 CommandAuditStartup(
29     STARTUP_TYPE                type                // IN: start up type
30 )
31 {
32     if(type == SU_RESET)
```

```

33     {
34         // Reset the digest size to initialize the digest
35         gr.commandAuditDigest.t.size = 0;
36     }
37 }
38 }

```

### 8.1.3.3 CommandAuditSet()

This function will SET the audit flag for a command. This function will not SET the audit flag for a command that is not implemented. This ensures that the audit status is not SET when TPM2\_GetCapability() is used to read the list of audited commands.

This function is only used by TPM2\_SetCommandCodeAuditStatus().

The actions in TPM2\_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

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

```

39 BOOL
40 CommandAuditSet(
41     TPM_CC      commandCode      // IN: command code
42 )
43 {
44     UINT32      bitPos;
45
46     // Only SET a bit if the corresponding command is implemented
47     if(CommandIsImplemented(commandCode))
48     {
49         // Can't audit shutdown
50         if(commandCode != TPM_CC_Shutdown)
51         {
52             bitPos = commandCode - TPM_CC_FIRST;
53             if(!BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
54             {
55                 // Set bit
56                 BitSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
57                 return TRUE;
58             }
59         }
60     }
61     // No change
62     return FALSE;
63 }

```

### 8.1.3.4 CommandAuditClear()

This function will CLEAR the audit flag for a command. It will not CLEAR the audit flag for TPM\_CC\_SetCommandCodeAuditStatus().

This function is only used by TPM2\_SetCommandCodeAuditStatus().

The actions in TPM2\_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

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

```

64  BOOL
65  CommandAuditClear(
66      TPM_CC      commandCode      // IN: command code
67  )
68  {
69      UINT32      bitPos;
70
71      // Do nothing if the command is not implemented
72      if(CommandIsImplemented(commandCode))
73      {
74          // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
75          // cleared
76          if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
77          {
78              bitPos = commandCode - TPM_CC_FIRST;
79              if(BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
80              {
81                  // Clear bit
82                  BitClear(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
83                  return TRUE;
84              }
85          }
86      }
87      // No change
88      return FALSE;
89  }

```

### 8.1.3.5 CommandAuditIsRequired()

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

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

```

90  BOOL
91  CommandAuditIsRequired(
92      TPM_CC      commandCode      // IN: command code
93  )
94  {
95      UINT32      bitPos;
96
97      bitPos = commandCode - TPM_CC_FIRST;
98
99      // Check the bit map. If the bit is SET, command audit is required
100     if((gp.auditComands[bitPos/8] & (1 << (bitPos % 8))) != 0)
101         return TRUE;
102     else
103         return FALSE;
104
105 }

```

### 8.1.3.6 CommandAuditCapGetCCList()

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

The list starts at the input *commandCode*.

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

```

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

```

### 8.1.3.7 CommandAuditGetDigest

This command is used to create a digest of the commands being audited. The commands are processed in ascending numeric order with a list of TPM\_CC being added to a hash. This operates as if all the audited command codes were concatenated and then hashed.

```

150 void
151 CommandAuditGetDigest(
152     TPM2B_DIGEST    *digest              // OUT: command digest
153 )
154 {

```

```

155     UINT32         i;
156     HASH_STATE    hashState;
157
158     // Start hash
159     digest->t.size = CryptStartHash(gp.auditHashAlg, &hashState);
160
161     // Add command code
162     for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
163     {
164         if(CommandAuditIsRequired(i))
165         {
166             CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &i);
167         }
168     }
169
170     // Complete hash
171     CryptCompleteHash2B(&hashState, &digest->b);
172
173     return;
174 }

```

## 8.2 DA.c

### 8.2.1 Introduction

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

### 8.2.2 Includes and Data Definitions

```

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

```

#### 8.2.2.1 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(void)
5  {
6     gp.failedTries = 0;
7     gp.maxTries = 3;
8     gp.recoveryTime = 1000;           // in seconds (~16.67 minutes)
9     gp.lockoutRecovery = 1000;       // in seconds
10    gp.lockOutAuthEnabled = TRUE;     // Use of lockoutAuth is enabled
11
12    // Record persistent DA parameter changes to NV
13    NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
14    NvWriteReserved(NV_MAX_TRIES, &gp.maxTries);
15    NvWriteReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
16    NvWriteReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
17    NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
18
19    return;
20 }

```



### 8.2.2.2 DStartup()

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.

```

21 void
22 DStartup(
23     STARTUP_TYPE          type          // IN: startup type
24 )
25 {
26     // For TPM Reset, if lockoutRecovery is 0, enable use of lockoutAuth.
27     if(type == SU_RESET)
28     {
29         if(gp.lockoutRecovery == 0)
30         {
31             gp.lockOutAuthEnabled = TRUE;
32             // Record the changes to NV
33             NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
34         }
35     }
36
37     // If DA has not been disabled and the previous shutdown is not orderly
38     // failedTries is not already at its maximum then increment 'failedTries'
39     if( gp.recoveryTime != 0
40         && g_prevOrderlyState == SHUTDOWN_NONE
41         && gp.failedTries < gp.maxTries)
42     {
43         gp.failedTries++;
44         // Record the change to NV
45         NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
46     }
47
48     // Reset self healing timers
49     s_selfHealTimer = g_time;
50     s_lockoutTimer = g_time;
51
52     return;
53 }

```

### 8.2.2.3 DRegisterFailure

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.

```

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

```

### 8.2.2.4 DAsSelfHeal()

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.

```

67 void
68 DAsSelfHeal(void)
69 {
70     // Regular auth self healing logic
71     // If no failed authorization tries, do nothing. Otherwise, try to
72     // decrease failedTries
73     if(gp.failedTries != 0)
74     {
75         // if recovery time is 0, DA logic has been disabled. Clear failed tries
76         // immediately
77         if(gp.recoveryTime == 0)
78         {
79             gp.failedTries = 0;
80             // Update NV record
81             NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
82         }
83         else
84         {
85             UINT64 decreaseCount;
86
87             // In the unlikely event that failedTries should become larger than
88             // maxTries
89             if(gp.failedTries > gp.maxTries)
90                 gp.failedTries = gp.maxTries;
91
92             // How much can failedTried be decreased
93             decreaseCount = ((g_time - s_selfHealTimer) / 1000) / gp.recoveryTime;
94
95             if(gp.failedTries <= (UINT32) decreaseCount)
96                 // should not set failedTries below zero
97                 gp.failedTries = 0;
98             else
99                 gp.failedTries -= (UINT32) decreaseCount;
100
101             // the cast prevents overflow of the product
102             s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
103             if(decreaseCount != 0)
104                 // If there was a change to the failedTries, record the changes
105                 // to NV
106                 NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
107         }
108     }
109
110     // LockoutAuth self healing logic
111     // If lockoutAuth is enabled, do nothing. Otherwise, try to see if we
112     // may enable it
113     if(!gp.lockOutAuthEnabled)
114     {
115         // if lockout authorization recovery time is 0, a reboot is required to
116         // re-enable use of lockout authorization. Self-healing would not
117         // apply in this case.
118         if(gp.lockoutRecovery != 0)
119         {
120             if(((g_time - s_lockoutTimer)/1000) >= gp.lockoutRecovery)
121             {
122                 gp.lockOutAuthEnabled = TRUE;
123                 // Record the changes to NV
124

```

```

125         NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
126     }
127 }
128 }
129
130 return;
131 }

```

## 8.3 Hierarchy.c

### 8.3.1 Introduction

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

### 8.3.2 Includes

```

1 #include "InternalRoutines.h"

```

#### 8.3.2.1 HierarchyPreInstall()

This function performs the initialization functions for the hierarchy when the TPM is simulated. This function should not be called if the TPM is not in a manufacturing mode at the manufacturer, or in a simulated environment.

```

2 void
3 HierarchyPreInstall_Init(void)
4 {
5     // Allow lockout clear command
6     gp.disableClear = FALSE;
7
8     // Initialize Primary Seeds
9     gp.EPSeed.t.size = PRIMARY_SEED_SIZE;
10    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.EPSeed.t.buffer);
11    gp.SPSeed.t.size = PRIMARY_SEED_SIZE;
12    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.SPSeed.t.buffer);
13    gp.PPSeed.t.size = PRIMARY_SEED_SIZE;
14    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.PPSeed.t.buffer);
15
16    // Initialize owner, endorsement and lockout auth
17    gp.ownerAuth.t.size = 0;
18    gp.endorsementAuth.t.size = 0;
19    gp.lockoutAuth.t.size = 0;
20
21    // Initialize owner and endorsement policy
22    gp.ownerAlg = TPM_ALG_NULL;
23    gp.ownerPolicy.t.size = 0;
24    gp.endorsementAlg = TPM_ALG_NULL;
25    gp.endorsementPolicy.t.size = 0;
26
27    // Initialize ehProof, shProof and phProof
28    gp.phProof.t.size = PROOF_SIZE;
29    gp.shProof.t.size = PROOF_SIZE;
30    gp.ehProof.t.size = PROOF_SIZE;
31    CryptGenerateRandom(gp.phProof.t.size, gp.phProof.t.buffer);
32    CryptGenerateRandom(gp.shProof.t.size, gp.shProof.t.buffer);
33    CryptGenerateRandom(gp.ehProof.t.size, gp.ehProof.t.buffer);
34
35    // Write hierarchy data to NV
36    NvWriteReserved(NV_DISABLE_CLEAR, &gp.disableClear);
37    NvWriteReserved(NV_EP_SEED, &gp.EPSeed);
38    NvWriteReserved(NV_SP_SEED, &gp.SPSeed);

```

```

39     NvWriteReserved(NV_PP_SEED, &gp.PPSeed);
40     NvWriteReserved(NV_OWNER_AUTH, &gp.ownerAuth);
41     NvWriteReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
42     NvWriteReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
43     NvWriteReserved(NV_OWNER_ALG, &gp.ownerAlg);
44     NvWriteReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
45     NvWriteReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
46     NvWriteReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
47     NvWriteReserved(NV_PH_PROOF, &gp.phProof);
48     NvWriteReserved(NV_SH_PROOF, &gp.shProof);
49     NvWriteReserved(NV_EH_PROOF, &gp.ehProof);
50
51     return;
52 }

```

### 8.3.2.2 HierarchyStartup()

This function is called at TPM2\_Startup() to initialize the hierarchy related values.

```

53 void
54 HierarchyStartup(
55     STARTUP_TYPE          type          // IN: start up type
56 )
57 {
58     // phEnable is SET on any startup
59     g_phEnable = TRUE;
60
61     // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
62     // TPM_RESTART
63     if(type != SU_RESUME)
64     {
65         gc.platformAuth.t.size = 0;
66         gc.platformPolicy.t.size = 0;
67
68         // enable the storage and endorsement hierarchies
69         gc.shEnable = gc.ehEnable = TRUE;
70     }
71
72     // nullProof and nullSeed is updated at every TPM_RESET
73     if(type == SU_RESET)
74     {
75         gr.nullProof.t.size = PROOF_SIZE;
76         CryptGenerateRandom(gr.nullProof.t.size,
77                             gr.nullProof.t.buffer);
78         gr.nullSeed.t.size = PRIMARY_SEED_SIZE;
79         CryptGenerateRandom(PRIMARY_SEED_SIZE, gr.nullSeed.t.buffer);
80     }
81
82     return;
83 }

```

### 8.3.2.3 HierarchyGetProof()

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

```

84 TPM2B_AUTH *
85 HierarchyGetProof(
86     TPMI_RH_HIERARCHY     hierarchy     // IN: hierarchy constant
87 )
88 {
89     switch(hierarchy)
90     {
91     case TPMI_RH_PLATFORM:

```

```

92     // phProof for TPM_RH_PLATFORM
93     return &gp.phProof;
94     break;
95     case TPM_RH_ENDORSEMENT:
96     // ehProof for TPM_RH_ENDORSEMENT
97     return &gp.ehProof;
98     break;
99     case TPM_RH_OWNER:
100    // shProof for TPM_RH_OWNER
101    return &gp.shProof;
102    break;
103    case TPM_RH_NULL:
104    // nullProof for TPM_RH_NULL
105    return &gr.nullProof;
106    break;
107    default:
108    pAssert(FALSE);
109    return NULL;
110    break;
111    }
112
113 }

```

#### 8.3.2.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```

114 TPM2B_SEED *
115 HierarchyGetPrimarySeed(
116     TPMI_RH_HIERARCHY hierarchy // IN: hierarchy
117 )
118 {
119     switch(hierarchy)
120     {
121     case TPM_RH_PLATFORM:
122         return &gp.PPSeed;
123         break;
124     case TPM_RH_OWNER:
125         return &gp.SPSeed;
126         break;
127     case TPM_RH_ENDORSEMENT:
128         return &gp.EPSeed;
129         break;
130     case TPM_RH_NULL:
131         return &gr.nullSeed;
132     default:
133         pAssert(FALSE);
134         return NULL;
135         break;
136     }
137 }

```

#### 8.3.2.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE: The TPM\_RH\_NULL hierarchy is always enabled.

Return Value	Meaning
TRUE	hierarchy is enabled
FALSE	hierarchy is disabled

```

138  BOOL
139  HierarchyIsEnabled(
140      TPMI_RH_HIERARCHY hierarchy    // IN: hierarchy
141  )
142  {
143      switch(hierarchy)
144      {
145          case TPM_RH_PLATFORM:
146              if(g_phEnable)
147                  return TRUE;
148              break;
149          case TPM_RH_OWNER:
150              if(gc.shEnable)
151                  return TRUE;
152              break;
153          case TPM_RH_ENDORSEMENT:
154              if(gc.ehEnable)
155                  return TRUE;
156              break;
157          case TPM_RH_NULL:
158              return TRUE;
159              break;
160          default:
161              pAssert(FALSE);
162              break;
163      }
164      return FALSE;
165  }

```

## 8.4 NV.c

### 8.4.1 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.

### 8.4.2 Includes, Defines and Data Definitions

```

1  #define NV_C
2  #include "InternalRoutines.h"
3  #include <Platform.h>

```

NV Index/evict object iterator value

```

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

```

### 8.4.3 NV Utility Functions

#### 8.4.3.1 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()*.

```

7  void
8  NvCheckState(void)
9  {
10     int    func_return;
11
12     func_return = _plat_IsNvAvailable();
13     if(func_return == 0)
14     {
15         s_NvIsAvailable = TPM_RC_SUCCESS;
16     }
17     else if(func_return == 1)
18     {
19         s_NvIsAvailable = TPM_RC_NV_UNAVAILABLE;
20     }
21     else
22     {
23         s_NvIsAvailable = TPM_RC_NV_RATE;
24     }
25
26     return;
27 }

```

#### 8.4.3.2 NvIsAvailable()

This function returns the NV availability parameter.

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

```

28  TPM_RC
29  NvIsAvailable(void)
30  {
31     return s_NvIsAvailable;
32 }

```

#### 8.4.3.3 NvCommit

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

```

33  BOOL
34  NvCommit(void)
35  {
36     if(_plat_NvCommit() == 0)
37         return TRUE;
38     else
39         return FALSE;
40 }

```

#### 8.4.3.4 NvReadMaxCount()

This function returns the max NV counter value.

```

41  static UINT64
42  NvReadMaxCount(void)
43  {
44      UINT64      countValue;
45      _plat__NvMemoryRead(s_maxCountAddr, sizeof(UINT64), &countValue);
46      return countValue;
47  }

```

#### 8.4.3.5 NvWriteMaxCount()

This function updates the max counter value to NV memory.

```

48  static void
49  NvWriteMaxCount(
50      UINT64      maxCount
51  )
52  {
53      _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &maxCount);
54      return;
55  }

```

### 8.4.4 NV Index and Persistent Object Access Functions

#### 8.4.4.1 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

#### 8.4.4.2 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.

```

56  static UINT32
57  NvNext(
58      NV_ITER      *iter
59  )
60  {
61      NV_ITER      currentIter;
62
63      // If iterator is at the beginning of list
64      if(*iter == NV_ITER_INIT)
65      {
66          // Initialize iterator
67          *iter = s_evictNvStart;
68      }

```



```

69
70 // If iterator reaches the end of NV space, or iterator indicates list end
71 if(*iter + sizeof(UINT32) > s_evictNvEnd || *iter == 0)
72     return 0;
73
74 // Save the current iter offset
75 currentIter = *iter;
76
77 // Adjust iter pointer pointing to next entity
78 // Read pointer value
79 _plat__NvMemoryRead(*iter, sizeof(UINT32), iter);
80
81 if(*iter == 0) return 0;
82
83 return currentIter + sizeof(UINT32); // entity stores after the pointer
84 }

```

#### 8.4.4.3 NvGetEnd()

Function to find the end of the NV dynamic data list

```

85 static UINT32
86 NvGetEnd(void)
87 {
88     NV_ITER        iter = NV_ITER_INIT;
89     UINT32         endAddr = s_evictNvStart;
90     UINT32         currentAddr;
91
92     while((currentAddr = NvNext(&iter)) != 0)
93         endAddr = currentAddr;
94
95     if(endAddr != s_evictNvStart)
96     {
97         // Read offset
98         endAddr -= sizeof(UINT32);
99         _plat__NvMemoryRead(endAddr, sizeof(UINT32), &endAddr);
100    }
101
102    return endAddr;
103 }

```

#### 8.4.4.4 NvGetFreeByte

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

```

104 static UINT32
105 NvGetFreeByte(void)
106 {
107     return s_evictNvEnd - NvGetEnd();
108 }

```

#### 8.4.4.5 NvGetEvictObjectSize

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

```

109 static UINT32
110 NvGetEvictObjectSize(void)
111 {
112     return sizeof(TPM_HANDLE) + sizeof(OBJECT) + sizeof(UINT32);
113 }

```

#### 8.4.4.6 NvGetCounterSize

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

```

114 static UINT32
115 NvGetCounterSize(void)
116 {
117     // It takes an offset field, a handle and the sizeof(NV_INDEX) and
118     // sizeof(UINT64) for counter data
119     return sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + sizeof(UINT64) + sizeof(UINT32);
120 }

```

#### 8.4.4.7 NvTestSpace()

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

Return Value	Meaning
TRUE	space available
FALSE	no enough space

```

121 static BOOL
122 NvTestSpace(
123     UINT32          size,           // IN: size of the entity to be added
124     BOOL            isIndex        // IN: TRUE if the entity is an index
125 )
126 {
127     UINT32          remainByte = NvGetFreeByte();
128
129     // For NV Index, need to make sure that we do not allocate and Index if this
130     // would mean that the TPM cannot allocate the minimum number of evict
131     // objects.
132     if(isIndex)
133     {
134         // Get the number of persistent objects allocated
135         UINT32          persistentNum = NvCapGetPersistentNumber();
136
137         // If we have not allocated the requisite number of evict objects, then we
138         // need to reserve space for them.
139         // NOTE: some of this is not written as simply as it might seem because
140         // the values are all unsigned and subtracting needs to be done carefully
141         // so that an underflow doesn't cause problems.
142         if(persistentNum < MIN_EVICT_OBJECTS)
143         {
144             UINT32          needed = (MIN_EVICT_OBJECTS - persistentNum)
145                                     * NvGetEvictObjectSize();
146             if(needed > remainByte)
147                 remainByte = 0;
148             else
149                 remainByte -= needed;
150         }
151         // if the requisite number of evict objects have been allocated then
152         // no need to reserve additional space
153     }
154     // This checks for the size of the value being added plus the index value.
155     // NOTE: This does not check to see if the end marker can be placed in
156     // memory because the end marker will not be written if it will not fit.
157     return (size + sizeof(UINT32) <= remainByte);
158 }

```

#### 8.4.4.8 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).

```

159 static void
160 NvAdd(
161     UINT32          totalSize,      // IN: total size needed for this entity
162                                     // For evict object, totalSize is the
163                                     // same as bufferSize. For NV Index,
164                                     // totalSize is bufferSize plus index
165                                     // data size
166     UINT32          bufferSize,    // IN: size of initial buffer
167     BYTE            *entity        // IN: initial buffer
168 )
169 {
170     UINT32          endAddr;
171     UINT32          nextAddr;
172     UINT32          listEnd = 0;
173
174     // Get the end of data list
175     endAddr = NvGetEnd();
176
177     // Calculate the value of next pointer, which is the size of a pointer +
178     // the entity data size
179     nextAddr = endAddr + sizeof(UINT32) + totalSize;
180
181     // Write next pointer
182     _plat__NvMemoryWrite(endAddr, sizeof(UINT32), &nextAddr);
183
184     // Write entity data
185     _plat__NvMemoryWrite(endAddr + sizeof(UINT32), bufferSize, entity);
186
187     // Write the end of list if it is not going to exceed the NV space
188     if(nextAddr + sizeof(UINT32) <= s_evictNvEnd)
189         _plat__NvMemoryWrite(nextAddr, sizeof(UINT32), &listEnd);
190
191     // Set the flag so that NV changes are committed before the command completes.
192     g_updateNV = TRUE;
193 }

```

#### 8.4.4.9 NvDelete()

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

```

194 static void
195 NvDelete(
196     UINT32          entityAddr      // IN: address of entity to be deleted
197 )
198 {
199     UINT32          next;
200     UINT32          entrySize;
201     UINT32          entryAddr = entityAddr - sizeof(UINT32);
202     UINT32          listEnd = 0;
203
204     // Get the offset of the next entry.
205     _plat__NvMemoryRead(entryAddr, sizeof(UINT32), &next);
206
207     // The size of this entry is the difference between the current entry and the
208     // next entry.
209     entrySize = next - entryAddr;
210 }

```

```

211 // Move each entry after the current one to fill the freed space.
212 // Stop when we have reached the end of all the indexes. There are two
213 // ways to detect the end of the list. The first is to notice that there
214 // is no room for anything else because we are at the end of NV. The other
215 // indication is that we find an end marker.
216
217 // The loop condition checks for the end of NV.
218 while(next + sizeof(UINT32) <= s_evictNvEnd)
219 {
220     UINT32     size, oldAddr, newAddr;
221
222     // Now check for the end marker
223     _plat__NvMemoryRead(next, sizeof(UINT32), &oldAddr);
224     if(oldAddr == 0)
225         break;
226
227     size = oldAddr - next;
228
229     // Move entry
230     _plat__NvMemoryMove(next, next - entrySize, size);
231
232     // Update forward link
233     newAddr = oldAddr - entrySize;
234     _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &newAddr);
235     next = oldAddr;
236 }
237 // Mark the end of list
238 _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &listEnd);
239
240 // Set the flag so that NV changes are committed before the command completes.
241 g_updateNV = TRUE;
242 }

```

## 8.4.5 RAM-based NV Index Data Access Functions

### 8.4.5.1 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/

### 8.4.5.2 NvTestRAMSpace()

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

Return Value	Meaning
TRUE	space available
FALSE	no enough space

```

243 static BOOL
244 NvTestRAMSpace(
245     UINT32     size // IN: size of the data to be added to RAM
246 )
247 {
248     if(s_ramIndexSize + size + sizeof(TPM_HANDLE) + sizeof(UINT32)
249        <= RAM_INDEX_SPACE)
250         return TRUE;
251     else

```

```

252     return FALSE;
253 }

```

### 8.4.5.3 NvGetRamIndexOffset

This function returns the offset of NV data in the RAM buffer

This function requires that NV Index is in RAM.

```

254 static UINT32
255 NvGetRAMIndexOffset(
256     TPMI_RH_NV_INDEX          handle          // IN: NV handle
257 )
258 {
259     UINT32          currAddr = 0;
260
261     while(currAddr < s_ramIndexSize)
262     {
263         TPMI_RH_NV_INDEX  currHandle;
264         UINT32            currSize;
265         currHandle = * (TPM_HANDLE *) &s_ramIndex[currAddr + sizeof(UINT32)];
266
267         // Found a match
268         if(currHandle == handle)
269
270             // data buffer follows the handle and size field
271             return currAddr + sizeof(TPMI_RH_NV_INDEX) + sizeof(UINT32);
272
273         currSize = * (UINT32 *) &s_ramIndex[currAddr];
274         currAddr += sizeof(UINT32) + currSize;
275     }
276
277     // We assume the index data is existing in RAM space
278     pAssert(FALSE);
279     return 0;
280 }

```

### 8.4.5.4 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.

```

281 static void
282 NvAddRAM(
283     TPMI_RH_NV_INDEX          handle,        // IN: NV handle
284     UINT32                    size          // IN: size of data
285 )
286 {
287     // Add data space at the end of reserved RAM buffer
288     * (UINT32 *) &s_ramIndex[s_ramIndexSize] = size + sizeof(TPMI_RH_NV_INDEX);
289     * (TPMI_RH_NV_INDEX *) &s_ramIndex[s_ramIndexSize + sizeof(UINT32)] = handle;
290     s_ramIndexSize += sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX) + size;
291
292     pAssert(s_ramIndexSize <= RAM_INDEX_SPACE);
293
294     // Update NV version of s_ramIndexSize
295     _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
296
297     // Write reserved RAM space to NV to reflect the newly added NV Index
298     _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
299
300     return;

```

```
301 }
```

#### 8.4.5.5 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

```
302 static void
303 NvDeleteRAM(
304     TPMI_RH_NV_INDEX      handle      // IN: NV handle
305 )
306 {
307     UINT32      nodeOffset;
308     UINT32      nextNode;
309     UINT32      size;
310
311     nodeOffset = NvGetRAMIndexOffset(handle);
312
313     // Move the pointer back to get the size field of this node
314     nodeOffset -= sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX);
315
316     // Get node size
317     size = * (UINT32 *) &s_ramIndex[nodeOffset];
318
319     // Get the offset of next node
320     nextNode = nodeOffset + sizeof(UINT32) + size;
321
322     // Move data
323     MemoryMove(s_ramIndex + nodeOffset, s_ramIndex + nextNode,
324               s_ramIndexSize - nextNode);
325
326     // Update RAM size
327     s_ramIndexSize -= size + sizeof(UINT32);
328
329     // Update NV version of s_ramIndexSize
330     _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
331
332     // Write reserved RAM space to NV to reflect the newly delete NV Index
333     _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
334
335     return;
336 }
```

#### 8.4.6 Utility Functions

##### 8.4.6.1 NvInitStatic()

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

```
337 static void
338 NvInitStatic(void)
339 {
340     UINT16      i;
341     UINT32      reservedAddr;
342
343     s_reservedSize[NV_DISABLE_CLEAR] = sizeof(gp.disableClear);
344     s_reservedSize[NV_OWNER_ALG] = sizeof(gp.ownerAlg);
345     s_reservedSize[NV_ENDORSEMENT_ALG] = sizeof(gp.endorsementAlg);
346     s_reservedSize[NV_OWNER_POLICY] = sizeof(gp.ownerPolicy);
347     s_reservedSize[NV_ENDORSEMENT_POLICY] = sizeof(gp.endorsementPolicy);
348     s_reservedSize[NV_OWNER_AUTH] = sizeof(gp.ownerAuth);
349 }
```

```

349     s_reservedSize[NV_ENDORSEMENT_AUTH] = sizeof(gp.endorsementAuth);
350     s_reservedSize[NV_LOCKOUT_AUTH] = sizeof(gp.lockoutAuth);
351     s_reservedSize[NV_EP_SEED] = sizeof(gp.EPSeed);
352     s_reservedSize[NV_SP_SEED] = sizeof(gp.SPSeed);
353     s_reservedSize[NV_PP_SEED] = sizeof(gp.PPSeed);
354     s_reservedSize[NV_PH_PROOF] = sizeof(gp.phProof);
355     s_reservedSize[NV_SH_PROOF] = sizeof(gp.shProof);
356     s_reservedSize[NV_EH_PROOF] = sizeof(gp.ehProof);
357     s_reservedSize[NV_TOTAL_RESET_COUNT] = sizeof(gp.totalResetCount);
358     s_reservedSize[NV_RESET_COUNT] = sizeof(gp.resetCount);
359     s_reservedSize[NV_PCR_POLICIES] = sizeof(gp.pcrPolicies);
360     s_reservedSize[NV_PCR_ALLOCATED] = sizeof(gp.pcrAllocated);
361     s_reservedSize[NV_PP_LIST] = sizeof(gp.ppList);
362     s_reservedSize[NV_FAILED_TRIES] = sizeof(gp.failedTries);
363     s_reservedSize[NV_MAX_TRIES] = sizeof(gp.maxTries);
364     s_reservedSize[NV_RECOVERY_TIME] = sizeof(gp.recoveryTime);
365     s_reservedSize[NV_LOCKOUT_RECOVERY] = sizeof(gp.lockoutRecovery);
366     s_reservedSize[NV_LOCKOUT_AUTH_ENABLED] = sizeof(gp.lockOutAuthEnabled);
367     s_reservedSize[NV_ORDERLY] = sizeof(gp.orderlyState);
368     s_reservedSize[NV_AUDIT_COMMANDS] = sizeof(gp.auditComands);
369     s_reservedSize[NV_AUDIT_HASH_ALG] = sizeof(gp.auditHashAlg);
370     s_reservedSize[NV_AUDIT_COUNTER] = sizeof(gp.auditCounter);
371     s_reservedSize[NV_ALGORITHM_SET] = sizeof(gp.algorithmSet);
372     s_reservedSize[NV_FIRMWARE_V1] = sizeof(gp.firmwareV1);
373     s_reservedSize[NV_FIRMWARE_V2] = sizeof(gp.firmwareV2);
374     s_reservedSize[NV_CLOCK] = sizeof(go.clock);
375     s_reservedSize[NV_STATE_CLEAR] = sizeof(gc);
376     s_reservedSize[NV_STATE_RESET] = sizeof(gr);
377
378     // Initialize reserved data address. In this implementation, reserved data
379     // is stored at the start of NV memory
380     reservedAddr = 0;
381     for(i = 0; i < NV_RESERVE_LAST; i++)
382     {
383         s_reservedAddr[i] = reservedAddr;
384         reservedAddr += s_reservedSize[i];
385     }
386
387     // Initialize auxiliary variable space for index/evict implementation.
388     // Auxiliary variables are stored after reserved data area
389     // RAM index copy starts at the beginning
390     s_ramIndexSizeAddr = reservedAddr;
391     s_ramIndexAddr = s_ramIndexSizeAddr + sizeof(UINT32);
392
393     // Maximum counter value
394     s_maxCountAddr = s_ramIndexAddr + RAM_INDEX_SPACE;
395
396     // dynamic memory start
397     s_evictNvStart = s_maxCountAddr + sizeof(UINT64);
398
399     // dynamic memory ends that the end of NV memory
400     s_evictNvEnd = NV_MEMORY_SIZE;
401
402     return;
403 }

```

#### 8.4.6.2 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.

404 **void**

```

405 NvInit(void)
406 {
407     UINT32     nullPointer = 0;
408     UINT64     zeroCounter = 0;
409
410     // Initialize static variables
411     NvInitStatic();
412
413     // Initialize RAM index space as un-used
414     _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &nullPointer);
415
416     // Initialize max counter value to 0
417     _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &zeroCounter);
418
419     // Initialize the next offset of the first entry in evict/index list to 0
420     _plat__NvMemoryWrite(s_evictNvStart, sizeof(TPM_HANDLE), &nullPointer);
421
422     return;
423
424 }

```

#### 8.4.6.3 NvReadReserved()

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

```

425 void
426 NvReadReserved(
427     NV_RESERVE     type,           // IN: type of reserved data
428     void            *buffer       // OUT: buffer receives the
429     // data.
430 )
431 {
432     // Input type should be valid
433     pAssert(type >= 0 && type < NV_RESERVE_LAST);
434
435     _plat__NvMemoryRead(s_reservedAddr[type], s_reservedSize[type], buffer);
436     return;
437 }

```

#### 8.4.6.4 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.

```

438 void
439 NvWriteReserved(
440     NV_RESERVE     type,           // IN: type of reserved data
441     void            *buffer       // IN: data buffer
442 )
443 {
444     // Input type should be valid
445     pAssert(type >= 0 && type < NV_RESERVE_LAST);
446
447     _plat__NvMemoryWrite(s_reservedAddr[type], s_reservedSize[type], buffer);
448
449     // Set the flag that a NV write happens
450     g_updateNV = TRUE;
451     return;
452 }

```



### 8.4.6.5 NvReadPersistent()

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

```

453 void
454 NvReadPersistent(void)
455 {
456     // Hierarchy persistent data
457     NvReadReserved(NV_DISABLE_CLEAR, &gp.disableClear);
458     NvReadReserved(NV_OWNER_ALG, &gp.ownerAlg);
459     NvReadReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
460     NvReadReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
461     NvReadReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
462     NvReadReserved(NV_OWNER_AUTH, &gp.ownerAuth);
463     NvReadReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
464     NvReadReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
465     NvReadReserved(NV_EP_SEED, &gp.EPSeed);
466     NvReadReserved(NV_SP_SEED, &gp.SPSeed);
467     NvReadReserved(NV_PP_SEED, &gp.PPSeed);
468     NvReadReserved(NV_PH_PROOF, &gp.phProof);
469     NvReadReserved(NV_SH_PROOF, &gp.shProof);
470     NvReadReserved(NV_EH_PROOF, &gp.ehProof);
471
472     // Time persistent data
473     NvReadReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
474     NvReadReserved(NV_RESET_COUNT, &gp.resetCount);
475
476     // PCR persistent data
477     NvReadReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
478     NvReadReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
479
480     // Physical Presence persistent data
481     NvReadReserved(NV_PP_LIST, &gp.ppList);
482
483     // Dictionary attack values persistent data
484     NvReadReserved(NV_FAILED_TRIES, &gp.failedTries);
485     NvReadReserved(NV_MAX_TRIES, &gp.maxTries);
486     NvReadReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
487     NvReadReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
488     NvReadReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
489
490     // Orderly State persistent data
491     NvReadReserved(NV_ORDERLY, &gp.orderlyState);
492
493     // Command audit values persistent data
494     NvReadReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
495     NvReadReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
496     NvReadReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
497
498     // Algorithm selection persistent data
499     NvReadReserved(NV_ALGORITHM_SET, &gp.algorithmSet);
500
501     // Firmware version persistent data
502     NvReadReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
503     NvReadReserved(NV_FIRMWARE_V2, &gp.firmwareV2);
504
505     return;
506 }

```

### 8.4.6.6 NvIsPlatformPersistentHandle()

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

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

```

507  BOOL
508  NvIsPlatformPersistentHandle(
509      TPM_HANDLE          handle          // IN: handle
510  )
511  {
512      return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
513  }

```

#### 8.4.6.7 NvIsOwnerPersistentHandle()

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

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

```

514  BOOL
515  NvIsOwnerPersistentHandle(
516      TPM_HANDLE          handle          // IN: handle
517  )
518  {
519      return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
520  }

```

#### 8.4.6.8 NvNextIndex()

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

```

521  static UINT32
522  NvNextIndex(
523      NV_ITER          *iter
524  )
525  {
526      UINT32          addr;
527      TPM_HANDLE      handle;
528
529      while((addr = NvNext(iter)) != 0)
530      {
531          // Read handle
532          _plat_NvMemoryRead(addr, sizeof(TPM_HANDLE), &handle);
533          if(HandleGetType(handle) == TPM_HT_NV_INDEX)
534              return addr;
535      }
536
537      pAssert(addr == 0);
538      return addr;
539  }

```

### 8.4.6.9 NvNextEvict()

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

```

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

```

### 8.4.6.10 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.

```

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

```

### 8.4.6.11 NvPowerOn()

This function is called at \_TPM\_Init() to initialize the NV environment.

```

579  void
580  NvPowerOn(void)
581  {
582      NvInitStatic();
583
584      return;
585  }

```

### 8.4.6.12 NvStateSave()

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

```

586 void
587 NvStateSave(void)
588 {
589     // Write RAM backed NV Index info to NV
590     // No need to save s_ramIndexSize because we save it to NV whenever it is
591     // updated.
592     _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
593
594     // Set the flag so that an NV write happens before the command completes.
595     g_updateNV = TRUE;
596
597     return;
598 }

```

### 8.4.6.13 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.

```

599 void
600 NvEntityStartup(
601     STARTUP_TYPE    type           // IN: start up type
602 )
603 {
604     NV_INDEX        iter = NV_ITER_INIT;
605     UINT32          currentAddr;    // offset points to the current entity
606
607     // Restore RAM index data
608     _plat__NvMemoryRead(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
609     _plat__NvMemoryRead(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
610
611     // If recovering from state save, do nothing
612     if(type == SU_RESUME)
613         return;
614
615     // Iterate all the NV Index to clear the locks
616     while((currentAddr = NvNextIndex(&iter)) != 0)
617     {
618         NV_INDEX    nvIndex;
619         UINT32      indexAddr;           // NV address points to index info
620         indexAddr = currentAddr + sizeof(TPM_HANDLE);
621
622         // Read NV Index info structure
623         _plat__NvMemoryRead(indexAddr, sizeof(NV_INDEX), &nvIndex);
624
625         // Clear read/write lock
626         if(nvIndex.publicArea.attributes.TPMA_NV_READLOCKED == SET)
627             nvIndex.publicArea.attributes.TPMA_NV_READLOCKED = CLEAR;
628         if(
629             nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED == SET
630             && nvIndex.publicArea.attributes.TPMA_NV_WRITEDEFINE == CLEAR)
631             nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED = CLEAR;

```

```

632 // Reset NV data for TPMA_NV_CLEAR_STCLEAR
633 if(nvIndex.publicArea.attributes.TPMA_NV_CLEAR_STCLEAR == SET)
634     nvIndex.publicArea.attributes.TPMA_NV_WRITTEN = CLEAR;
635
636 // Reset NV data for orderly values that are not counters
637 // NOTE: The function has already exited on a TPM Resume, so the only
638 // things being processed are TPM Restart and TPM Reset
639 if(    type == SU_RESET
640     && nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET
641     && nvIndex.publicArea.attributes.TPMA_NV_COUNTER == CLEAR)
642     nvIndex.publicArea.attributes.TPMA_NV_WRITTEN = CLEAR;
643
644 // Write NV Index info back
645 _plat__NvMemoryWrite(indexAddr, sizeof(NV_INDEX), &nvIndex);
646
647 // Set the flag that a NV write happens
648 g_updateNV = TRUE;
649
650 // Set the lower bits in an orderly counter to 1 for a non-orderly startup
651 if(    g_prevOrderlyState == SHUTDOWN_NONE
652     && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
653 {
654     if(    nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET
655         && nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET)
656     {
657         TPMI_RH_NV_INDEX    nvHandle;
658         UINT64              counter;
659
660         // Read NV handle
661         _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);
662
663         // Read the counter value saved to NV upon the last roll over.
664         // Do not use RAM backed storage for this once.
665         nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = CLEAR;
666         NvGetIntIndexData(nvHandle, &nvIndex, &counter);
667         nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = SET;
668
669         // Set the lower bits of counter to 1
670         counter |= MAX_ORDERLY_COUNT;
671
672         // Write back to RAM
673         NvWriteIndexData(nvHandle, &nvIndex, 0, 8, &counter);
674
675         // No write to NV because an orderly shutdown will update the
676         // counters.
677
678     }
679 }
680 }
681
682 return;
683
684 }

```

## 8.4.7 NV Access Functions

### 8.4.7.1 Introduction

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

### 8.4.7.2 NvIsUndefinedIndex()

This function is used to verify that an NV Index is not defined.

Error Returns	Meaning
TPM_RC_NV_DEFINED	the handle points to an existing NV Index
TPM_RC_HIERARCHY	the handle points to an existing NV Index that is created by a disabled hierarchy

```

685 TPM_RC
686 NvIsUndefinedIndex(
687     TPMI_RH_NV_INDEX handle    // IN: handle
688 )
689 {
690     UINT32     entityAddr;      // offset points to the entity
691     NV_INDEX   nvIndex;
692
693     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
694
695     // Find the address of index
696     entityAddr = NvFindHandle(handle);
697
698     // If handle is not found, return TPM_RC_SUCCESS
699     if(entityAddr == 0) return TPM_RC_SUCCESS;
700
701     // Read NV Index info structure
702     _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
703                         &nvIndex);
704
705     // if SHEnable is disabled, an ownerCreate NV Index should not be
706     // indicated as present
707     if(gc.shEnable == FALSE &&
708        nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
709         return TPM_RC_HIERARCHY;
710
711     // if PHEnable is disabled, a platformCreate NV Index should not be
712     // indicated as present
713     if(g_phEnable == FALSE &&
714        nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
715         return TPM_RC_HIERARCHY;
716
717     // NV Index is defined
718     return TPM_RC_NV_DEFINED;
719 }

```

### 8.4.7.3 NvIndexIsAccessible()

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

Error Returns	Meaning
TPM_RC_HANDLE	the handle points to an undefined NV Index
TPM_RC_HIERARCHY	the handle points to an existing NV Index that is created by a disabled hierarchy

```

720 TPM_RC
721 NvIndexIsAccessible(
722     TPMI_RH_NV_INDEX handle    // IN: handle
723 )

```

```

724 {
725     UINT32          entityAddr;    // offset points to the entity
726     NV_INDEX       nvIndex;
727
728     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
729
730     // Find the address of index
731     entityAddr = NvFindHandle(handle);
732
733     // If handle is not found, return TPM_RC_HANDLE
734     if(entityAddr == 0) return TPM_RC_HANDLE;
735
736     // Read NV Index info structure
737     _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
738                         &nvIndex);
739
740     // if shEnable is CLEAR, an ownerCreate NV Index should not be
741     // indicated as present
742     if(gc.shEnable == FALSE &&
743        nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
744         return TPM_RC_HIERARCHY;
745
746     // if phEnable is disabled, a platformCreate NV Index should not be
747     // indicated as present
748     if(g_phEnable == FALSE &&
749        nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
750         return TPM_RC_HIERARCHY;
751
752     // NV Index is accessible
753     return TPM_RC_SUCCESS;
754 }

```

#### 8.4.7.4 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.

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

```

755 static BOOL
756 NvIsUndefinedEvictHandle(
757     TPM_HANDLE     handle        // IN: handle
758 )
759 {
760     UINT32          entityAddr;    // offset points to the entity
761     pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
762
763     // Find the address of evict object
764     entityAddr = NvFindHandle(handle);
765
766     // If handle is not found, return TRUE
767     if(entityAddr == 0)
768         return TRUE;
769     else
770         return FALSE;
771 }

```

### 8.4.7.5 NvGetEvictObject()

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

Error Returns	Meaning
TPM_RC_REFERENCE_H0	the handle does not point to an existing persistent object
TPM_RC_HIERARCHY	the handle points to an existing persistent object belongs to a disabled hierarchy

```

772  TPM_RC
773  NvGetEvictObject(
774      TPM_HANDLE      handle,      // IN: handle
775      OBJECT          *object      // OUT: object data
776  )
777  {
778      UINT32          entityAddr;    // offset points to the entity
779
780      pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
781
782      // Find the address of evict object
783      entityAddr = NvFindHandle(handle);
784
785      // If handle is not found, return TPM_RC_REFERENCE_H0
786      if(entityAddr == 0) return TPM_RC_REFERENCE_H0;
787
788      // Read evict object
789      _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(OBJECT), object);
790
791      if(HierarchyIsEnabled(ObjectDataGetHierarchy(object)) == FALSE)
792          return TPM_RC_HIERARCHY;
793
794      return TPM_RC_SUCCESS;
795  }

```

### 8.4.7.6 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.

```

796  void
797  NvGetIndexInfo(
798      TPMI_RH_NV_INDEX handle,      // IN: handle
799      NV_INDEX        *nvIndex     // OUT: NV index structure
800  )
801  {
802      UINT32          entityAddr;    // offset points to the entity
803
804      pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
805
806      // Find the address of evict object
807      entityAddr = NvFindHandle(handle);
808      pAssert(entityAddr != 0);
809
810      // This implementation uses the default format so just
811      // read the data in
812      _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
813                          nvIndex);

```



```

814     return;
815 }
816

```

#### 8.4.7.7 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.

```

817 UINT64
818 NvInitialCounter(void)
819 {
820     UINT64         maxCount;
821     NV_ITER        iter = NV_ITER_INIT;
822     UINT32         currentAddr;
823
824     // Read the maxCount value
825     maxCount = NvReadMaxCount();
826
827     // Iterate all existing counters
828     while((currentAddr = NvNextIndex(&iter)) != 0)
829     {
830         TPMI_RH_NV_INDEX    nvHandle;
831         NV_INDEX           nvIndex;
832
833         // Read NV handle
834         _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);
835
836         // Get NV Index
837         NvGetIndexInfo(nvHandle, &nvIndex);
838         if(    nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
839             && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
840         {
841             UINT64         countValue;
842             // Read counter value
843             NvGetIntIndexData(nvHandle, &nvIndex, &countValue);
844             if(countValue > maxCount)
845                 maxCount = countValue;
846         }
847     }
848     // Initialize the new counter value to be maxCount + 1
849     // A counter is only initialized the first time it is written. The
850     // way to write a counter is with TPM2_NV_INCREMENT(). Since the
851     // "initial" value of a defined counter is the largest count value that
852     // may have existed in this index previously, then the first use would
853     // add one to that value.
854     return maxCount;
855 }

```

#### 8.4.7.8 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**.

```

856 void
857 NvGetIndexData(
858     TPMI_RH_NV_INDEX    handle,           // IN: handle
859     NV_INDEX           *nvIndex,        // IN: RAM image of index header

```

```

860     UINT32         offset,           // IN: offset of NV data
861     UINT16         size,             // IN: size of NV data
862     void           *data             // OUT: data buffer
863 )
864 {
865
866     pAssert(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET);
867
868     if(   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
869         || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET)
870     {
871         // Read bit or counter data in canonical form
872         UINT64     dataInInt;
873         NvGetIntIndexData(handle, nvIndex, &dataInInt);
874         UINT64_TO_BYTE_ARRAY(dataInInt, (BYTE *)data);
875     }
876     else
877     {
878         if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
879         {
880             UINT32     ramAddr;
881
882             // Get data from RAM buffer
883             ramAddr = NvGetRAMIndexOffset(handle);
884             MemoryCopy(data, s_ramIndex + ramAddr + offset, size);
885         }
886         else
887         {
888             UINT32     entityAddr;
889             entityAddr = NvFindHandle(handle);
890             // Get data from NV
891             // Skip NV Index info, read data buffer
892             entityAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
893             // Read the data
894             _plat__NvMemoryRead(entityAddr, size, data);
895         }
896     }
897     return;
898 }

```

#### 8.4.7.9 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.

```

899     void
900     NvGetIntIndexData(
901         TPMI_RH_NV_INDEX handle,           // IN: handle
902         NV_INDEX         *nvIndex,        // IN: RAM image of NV Index header
903         UINT64           *data            // IN: UINT64 pointer for counter or bits
904     )
905     {
906         // Validate that index has been written and is the right type
907         pAssert(   nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET
908                 && (   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
909                     || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET
910                 )
911         );
912
913         // bit and counter value is store in native format for TPM CPU. So we directly
914         // copy the contents of NV to output data buffer
915         if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
916         {
917             UINT32     ramAddr;

```

```

918
919     // Get data from RAM buffer
920     ramAddr = NvGetRAMIndexOffset(handle);
921     MemoryCopy(data, s_ramIndex + ramAddr, sizeof(*data));
922 }
923 else
924 {
925     UINT32     entityAddr;
926     entityAddr = NvFindHandle(handle);
927
928     // Get data from NV
929     // Skip NV Index info, read data buffer
930     _plat__NvMemoryRead(
931         entityAddr + sizeof(TPM_HANDLE) + sizeof(NV_INDEX),
932         sizeof(UINT64), data);
933 }
934
935 return;
936 }

```

#### 8.4.7.10 NvWriteIndexInfo()

This function is called to queue the write of NV Index data to persistent memory.

This function requires that NV Index is defined.

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

```

937 TPM_RC
938 NvWriteIndexInfo(
939     TPMI_RH_NV_INDEX    handle,    // IN: handle
940     NV_INDEX            *nvIndex   // IN: NV Index info to be written
941 )
942 {
943     UINT32     entryAddr;
944     TPM_RC     result;
945
946     // Get the starting offset for the index in the RAM image of NV
947     entryAddr = NvFindHandle(handle);
948     pAssert(entryAddr != 0);
949
950     // Step over the link value
951     entryAddr = entryAddr + sizeof(TPM_HANDLE);
952
953     // If the index data is actually changed, then a write to NV is required
954     if(_plat__NvIsDifferent(entryAddr, sizeof(NV_INDEX), nvIndex))
955     {
956         // Make sure that NV is available
957         result = NvIsAvailable();
958         if(result != TPM_RC_SUCCESS)
959             return result;
960         _plat__NvMemoryWrite(entryAddr, sizeof(NV_INDEX), nvIndex);
961         g_updateNV = TRUE;
962     }
963     return TPM_RC_SUCCESS;
964 }

```

### 8.4.7.11 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

```

965  TPM_RC
966  NvWriteIndexData(
967      TPMI_RH_NV_INDEX    handle,    // IN: handle
968      NV_INDEX            *nvIndex,   // IN: RAM copy of NV Index
969      UINT32              offset,     // IN: offset of NV data
970      UINT32              size,       // IN: size of NV data
971      void                *data       // OUT: data buffer
972  )
973  {
974      TPM_RC                result;
975      // Validate that write falls within range of the index
976      pAssert(nvIndex->publicArea.dataSize >= offset + size);
977
978      // Update TPMA_NV_WRITTEN bit if necessary
979      if(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
980      {
981          nvIndex->publicArea.attributes.TPMA_NV_WRITTEN = SET;
982          result = NvWriteIndexInfo(handle, nvIndex);
983          if(result != TPM_RC_SUCCESS)
984              return result;
985      }
986
987      // Check to see if process for an orderly index is required.
988      if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
989      {
990          UINT32            ramAddr;
991
992          // Write data to RAM buffer
993          ramAddr = NvGetRAMIndexOffset(handle);
994          MemoryCopy(s_ramIndex + ramAddr + offset, data, size);
995
996          // NV update does not happen for orderly index. Have
997          // to clear orderlyState to reflect that we have changed the
998          // NV and an orderly shutdown is required. Only going to do this if we
999          // are not processing a counter that has just rolled over
1000         if(g_updateNV == FALSE)
1001             g_clearOrderly = TRUE;
1002     }
1003     // Need to process this part if the Index isn't orderly or if it is
1004     // an orderly counter that just rolled over.
1005     if(g_updateNV || nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == CLEAR)
1006     {
1007         // Processing for an index with TPMA_NV_ORDERLY CLEAR
1008         UINT32            entryAddr = NvFindHandle(handle);
1009
1010         pAssert(entryAddr != 0);
1011
1012         // Offset into the index to the first byte of the data to be written
1013         entryAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
1014
1015         // If the data is actually changed, then a write to NV is required
1016         if(_plat_NvIsDifferent(entryAddr, size, data))

```

```

1017     {
1018         // Make sure that NV is available
1019         result = NvIsAvailable();
1020         if(result != TPM_RC_SUCCESS)
1021             return result;
1022         _plat_NvMemoryWrite(entryAddr, size, data);
1023         g_updateNV = TRUE;
1024     }
1025 }
1026
1027 return TPM_RC_SUCCESS;
1028 }

```

#### 8.4.7.12 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.

```

1029 UINT16
1030 NvGetName(
1031     TPMI_RH_NV_INDEX    handle,    // IN: handle of the index
1032     BYTE                *name      // OUT: name of the index
1033 )
1034 {
1035     UINT16                dataSize, digestSize;
1036     NV_INDEX              nvIndex;
1037     BYTE                  marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
1038     BYTE                  *buffer;
1039     HASH_STATE            hashState;
1040
1041     // Get NV public info
1042     NvGetIndexInfo(handle, &nvIndex);
1043
1044     // Marshal public area
1045     buffer = marshalBuffer;
1046     dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex.publicArea, &buffer, NULL);
1047
1048     // hash public area
1049     digestSize = CryptStartHash(nvIndex.publicArea.nameAlg, &hashState);
1050     CryptUpdateDigest(&hashState, dataSize, marshalBuffer);
1051
1052     // Complete digest leaving room for the nameAlg
1053     CryptCompleteHash(&hashState, digestSize, &name[2]);
1054
1055     // Include the nameAlg
1056     UINT16_TO_BYTE_ARRAY(nvIndex.publicArea.nameAlg, name);
1057     return digestSize + 2;
1058 }

```

#### 8.4.7.13 NvDefineIndex()

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

Error Returns	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```

1059 TPM_RC
1060 NvDefineIndex(

```

```

1061     TPMS_NV_PUBLIC      *publicArea,      // IN: A template for an area to create.
1062     TPM2B_AUTH          *authValue       // IN: The initial authorization value
1063 )
1064 {
1065     // The buffer to be written to NV memory
1066     BYTE                nvBuffer[sizeof(TPM_HANDLE) + sizeof(NV_INDEX)];
1067
1068     NV_INDEX            *nvIndex;         // a pointer to the NV_INDEX data in
1069                                         // nvBuffer
1070     UINT16              entrySize;       // size of entry
1071
1072     entrySize = sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + publicArea->dataSize;
1073
1074     // Check if we have enough space to create the NV Index
1075     // In this implementation, the only resource limitation is the available NV
1076     // space. Other implementation may have other limitation on counter or on
1077     // NV slot
1078     if(!NvTestSpace(entrySize, TRUE)) return TPM_RC_NV_SPACE;
1079
1080     // if the index to be defined is RAM backed, check RAM space availability
1081     // as well
1082     if(publicArea->attributes.TPMA_NV_ORDERLY == SET
1083        && !NvTestRAMSpace(publicArea->dataSize))
1084         return TPM_RC_NV_SPACE;
1085
1086
1087     // Copy input value to nvBuffer
1088     // Copy handle
1089     * (TPM_HANDLE *) nvBuffer = publicArea->nvIndex;
1090
1091     // Copy NV_INDEX
1092     nvIndex = (NV_INDEX *) (nvBuffer + sizeof(TPM_HANDLE));
1093     nvIndex->publicArea = *publicArea;
1094     nvIndex->authValue = *authValue;
1095
1096     // Add index to NV memory
1097     NvAdd(entrySize, sizeof(TPM_HANDLE) + sizeof(NV_INDEX), nvBuffer);
1098
1099     // If the data of NV Index is RAM backed, add the data area in RAM as well
1100     if(publicArea->attributes.TPMA_NV_ORDERLY == SET)
1101         NvAddRAM(publicArea->nvIndex, publicArea->dataSize);
1102
1103     return TPM_RC_SUCCESS;
1104 }

```

#### 8.4.7.14 NvAddEvictObject()

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

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

```

1105     TPM_RC
1106     NvAddEvictObject(
1107         TPMS_DH_OBJECT   evictHandle,    // IN: new evict handle
1108         OBJECT           *object        // IN: object to be added
1109     )
1110     {
1111         // The buffer to be written to NV memory
1112         BYTE                nvBuffer[sizeof(TPM_HANDLE) + sizeof(OBJECT)];
1113

```

```

1114     OBJECT          *nvObject;          // a pointer to the OBJECT data in
1115                                     // nvBuffer
1116     UINT16         entrySize;          // size of entry
1117
1118     // evict handle type should match the object hierarchy
1119     pAssert( ( NvIsPlatformPersistentHandle(evictHandle)
1120             && object->attributes.ppsHierarchy == SET)
1121             || ( NvIsOwnerPersistentHandle(evictHandle)
1122             && ( object->attributes.spsHierarchy == SET
1123             || object->attributes.epsHierarchy == SET)));
1124
1125     // An evict needs 4 bytes of handle + sizeof OBJECT
1126     entrySize = sizeof(TPM_HANDLE) + sizeof(OBJECT);
1127
1128     // Check if we have enough space to add the evict object
1129     // An evict object needs 8 bytes in index table + sizeof OBJECT
1130     // In this implementation, the only resource limitation is the available NV
1131     // space. Other implementation may have other limitation on evict object
1132     // handle space
1133     if(!NvTestSpace(entrySize, FALSE)) return TPM_RC_NV_SPACE;
1134
1135     // Allocate a new evict handle
1136     if(!NvIsUndefinedEvictHandle(evictHandle))
1137         return TPM_RC_NV_DEFINED;
1138
1139     // Copy evict object to nvBuffer
1140     // Copy handle
1141     * (TPM_HANDLE *) nvBuffer = evictHandle;
1142
1143     // Copy OBJECT
1144     nvObject = (OBJECT *) (nvBuffer + sizeof(TPM_HANDLE));
1145     *nvObject = *object;
1146
1147     // Set evict attribute and handle
1148     nvObject->attributes.evict = SET;
1149     nvObject->evictHandle = evictHandle;
1150
1151     // Add evict to NV memory
1152     NvAdd(entrySize, entrySize, nvBuffer);
1153
1154     return TPM_RC_SUCCESS;
1155 }
1156

```

#### 8.4.7.15 NvDeleteEntity()

This function will delete a NV Index or an evict object.

This function requires that the index/evict object has been defined.

```

1157 void
1158 NvDeleteEntity(
1159     TPM_HANDLE     handle          // IN: handle of entity to be deleted
1160 )
1161 {
1162     UINT32         entityAddr;     // pointer to entity
1163
1164     entityAddr = NvFindHandle(handle);
1165     pAssert(entityAddr != 0);
1166
1167     if(HandleGetType(handle) == TPM_HT_NV_INDEX)
1168     {
1169         NV_INDEX     nvIndex;
1170
1171         // Read the NV Index info

```

```

1172     _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
1173                       &nvIndex);
1174
1175     // If the entity to be deleted is a counter with the maximum counter
1176     // value, record it in NV memory
1177     if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
1178        && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
1179     {
1180         UINT64     countValue;
1181         UINT64     maxCount;
1182         NvGetIntIndexData(handle, &nvIndex, &countValue);
1183         maxCount = NvReadMaxCount();
1184         if(countValue > maxCount)
1185             NvWriteMaxCount(countValue);
1186     }
1187     // If the NV Index is RAM back, delete the RAM data as well
1188     if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
1189         NvDeleteRAM(handle);
1190 }
1191 NvDelete(entityAddr);
1192
1193 return;
1194
1195 }

```

#### 8.4.7.16 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*.

```

1196 void
1197 NvFlushHierarchy(
1198     TPMI_RH_HIERARCHY     hierarchy     // IN: hierarchy to be flushed.
1199 )
1200 {
1201     NV_ITER     iter = NV_ITER_INIT;
1202     UINT32     currentAddr;
1203
1204     while((currentAddr = NvNext(&iter)) != 0)
1205     {
1206         TPM_HANDLE     entityHandle;
1207
1208         // Read handle information.
1209         _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1210
1211         if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
1212         {
1213             // Handle NV Index
1214             NV_INDEX     nvIndex;
1215
1216             // If flush endorsement or platform hierarchy, no NV Index would be
1217             // flushed
1218             if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
1219                 continue;
1220             _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1221                               sizeof(NV_INDEX), &nvIndex);
1222
1223             // For storage hierarchy, flush OwnerCreated index
1224             if( nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
1225             {
1226                 // Delete the NV Index
1227                 NvDelete(currentAddr);
1228
1229                 // Re-iterate from beginning after a delete

```



```

1230         iter = NV_ITER_INIT;
1231
1232         // If the NV Index is RAM back, delete the RAM data as well
1233         if (nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
1234             NvDeleteRAM(entityHandle);
1235     }
1236 }
1237 else if (HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
1238 {
1239     OBJECT        object;
1240
1241     // Get evict object
1242     NvGetEvictObject(entityHandle, &object);
1243
1244     // If the evict object belongs to the hierarchy to be flushed
1245     if( (    hierarchy == TPM_RH_PLATFORM
1246         && object.attributes.ppsHierarchy == SET)
1247       || (    hierarchy == TPM_RH_OWNER
1248         && object.attributes.spsHierarchy == SET)
1249       || (    hierarchy == TPM_RH_ENDORSEMENT
1250         && object.attributes.epsHierarchy == SET)
1251     )
1252     {
1253         // Delete the evict object
1254         NvDelete(currentAddr);
1255
1256         // Re-iterate from beginning after a delete
1257         iter = NV_ITER_INIT;
1258     }
1259 }
1260 else
1261 {
1262     pAssert(FALSE);
1263 }
1264 }
1265
1266 return;
1267 }

```

#### 8.4.7.17 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().

```

1268 void
1269 NvSetGlobalLock(void)
1270 {
1271     NV_ITER        iter = NV_ITER_INIT;
1272     UINT32         currentAddr;
1273
1274     // Check all indices
1275     while((currentAddr = NvNextIndex(&iter)) != 0)
1276     {
1277         NV_INDEX    nvIndex;
1278
1279         // Read the index data
1280         _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1281                             sizeof(NV_INDEX), &nvIndex);
1282
1283         // See if it should be locked
1284         if (nvIndex.publicArea.attributes.TPMA_NV_GLOBALLOCK == SET)
1285         {
1286
1287             // if so, lock it

```

```

1288         nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED = SET;
1289
1290         _plat__NvMemoryWrite(currentAddr + sizeof(TPM_HANDLE),
1291                             sizeof(NV_INDEX), &nvIndex);
1292         // Set the flag that a NV write happens
1293         g_updateNV = TRUE;
1294     }
1295 }
1296
1297 return;
1298
1299 }

```

#### 8.4.7.18 InsertSort()

Sort a handle into handle list in ascending order. The total handle number in the list should not exceed MAX\_CAP\_HANDLES

```

1300 static void
1301 InsertSort(
1302     TPML_HANDLE *handleList, // IN/OUT: sorted handle list
1303     UINT32 count, // IN: maximum count in the handle
1304                // list
1305     TPM_HANDLE entityHandle // IN: handle to be inserted
1306 )
1307 {
1308     UINT32 i, j;
1309     UINT32 originalCount;
1310
1311     // For a corner case that the maximum count is 0, do nothing
1312     if(count == 0) return;
1313
1314     // For empty list, add the handle at the beginning and return
1315     if(handleList->count == 0)
1316     {
1317         handleList->handle[0] = entityHandle;
1318         handleList->count++;
1319         return;
1320     }
1321
1322     // Check if the maximum of the list has been reached
1323     originalCount = handleList->count;
1324     if(originalCount < count)
1325         handleList->count++;
1326
1327     // Insert the handle to the list
1328     for(i = 0; i < originalCount; i++)
1329     {
1330         if(handleList->handle[i] > entityHandle)
1331         {
1332             for(j = handleList->count - 1; j > i; j--)
1333             {
1334                 handleList->handle[j] = handleList->handle[j-1];
1335             }
1336             break;
1337         }
1338     }
1339
1340     // If a slot was found, insert the handle in this position
1341     if(i < originalCount || handleList->count > originalCount)
1342         handleList->handle[i] = entityHandle;
1343
1344     return;
1345 }

```

### 8.4.7.19 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.

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

```

1346  TPMI_YES_NO
1347  NvCapGetPersistent(
1348      TPMI_DH_OBJECT    handle,        // IN: start handle
1349      UINT32            count,         // IN: maximum number of returned handles
1350      TPML_HANDLE       *handleList    // OUT: list of handle
1351  )
1352  {
1353      TPMI_YES_NO        more = NO;
1354      NV_ITER            iter = NV_ITER_INIT;
1355      UINT32             currentAddr;
1356
1357      pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1358
1359      // Initialize output handle list
1360      handleList->count = 0;
1361
1362      // The maximum count of handles we may return is MAX_CAP_HANDLES
1363      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1364
1365      while((currentAddr = NvNextEvict(&iter)) != 0)
1366      {
1367          TPM_HANDLE      entityHandle;
1368
1369          // Read handle information.
1370          _plat_NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1371
1372          // Ignore persistent handles that have values less than the input handle
1373          if(entityHandle < handle)
1374              continue;
1375
1376          // if the handles in the list have reached the requested count, and there
1377          // are still handles need to be inserted, indicate that there are more.
1378          if(handleList->count == count)
1379              more = YES;
1380
1381          // A handle with a value larger than start handle is a candidate
1382          // for return. Insert sort it to the return list. Insert sort algorithm
1383          // is chosen here for simplicity based on the assumption that the total
1384          // number of NV Indices is small. For an implementation that may allow
1385          // large number of NV Indices, a more efficient sorting algorithm may be
1386          // used here.
1387          InsertSort(handleList, count, entityHandle);
1388
1389      }
1390      return more;
1391  }

```

### 8.4.7.20 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.

Return Value	Meaning
YES	if there are more handles to report
NO	all the available handles has been reported

```

1392 TPMI_YES_NO
1393 NvCapGetIndex(
1394     TPMI_DH_OBJECT    handle,        // IN: start handle
1395     UINT32            count,        // IN: maximum number of returned handles
1396     TPML_HANDLE       *handleList   // OUT: list of handle
1397 )
1398 {
1399     TPMI_YES_NO        more = NO;
1400     NV_ITER            iter = NV_ITER_INIT;
1401     UINT32             currentAddr;
1402
1403     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1404
1405     // Initialize output handle list
1406     handleList->count = 0;
1407
1408     // The maximum count of handles we may return is MAX_CAP_HANDLES
1409     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1410
1411     while((currentAddr = NvNextIndex(&iter)) != 0)
1412     {
1413         TPM_HANDLE     entityHandle;
1414
1415         // Read handle information.
1416         _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1417
1418         // Ignore index handles that have values less than the 'handle'
1419         if(entityHandle < handle)
1420             continue;
1421
1422         // if the count of handles in the list has reached the requested count,
1423         // and there are still handles to report, set more.
1424         if(handleList->count == count)
1425             more = YES;
1426
1427         // A handle with a value larger than start handle is a candidate
1428         // for return. Insert sort it to the return list. Insert sort algorithm
1429         // is chosen here for simplicity based on the assumption that the total
1430         // number of NV Indices is small. For an implementation that may allow
1431         // large number of NV Indices, a more efficient sorting algorithm may be
1432         // used here.
1433         InsertSort(handleList, count, entityHandle);
1434     }
1435     return more;
1436 }

```

#### 8.4.7.21 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```

1437 UINT32
1438 NvCapGetIndexNumber(void)
1439 {
1440     UINT32            num = 0;
1441     NV_ITER            iter = NV_ITER_INIT;
1442
1443     while(NvNextIndex(&iter) != 0) num++;
1444 }

```

```

1445     return num;
1446 }

```

#### 8.4.7.22 NvCapGetPersistentNumber()

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

```

1447 UINT32
1448 NvCapGetPersistentNumber(void)
1449 {
1450     UINT32          num = 0;
1451     NV_ITER        iter = NV_ITER_INIT;
1452
1453     while(NvNextEvict(&iter) != 0) num++;
1454
1455     return num;
1456 }

```

#### 8.4.7.23 NvCapGetPersistentAvail()

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

```

1457 UINT32
1458 NvCapGetPersistentAvail(void)
1459 {
1460     UINT32          availSpace;
1461     UINT32          objectSpace;
1462
1463     // Compute the available space in NV storage
1464     availSpace = NvGetFreeByte();
1465
1466     // Get the space needed to add a persistent object to NV storage
1467     objectSpace = NvGetEvictObjectSize();
1468
1469     return availSpace / objectSpace;
1470 }

```

#### 8.4.7.24 NvCapGetCounterNumber()

Get the number of defined NV Indexes that have NV TPMA\_NV\_COUNTER attribute SET.

```

1471 UINT32
1472 NvCapGetCounterNumber(void)
1473 {
1474     NV_ITER        iter = NV_ITER_INIT;
1475     UINT32          currentAddr;
1476     UINT32          num = 0;
1477
1478     while((currentAddr = NvNextIndex(&iter)) != 0)
1479     {
1480         NV_INDEX    nvIndex;
1481
1482         // Get NV Index info
1483         _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1484                             sizeof(NV_INDEX), &nvIndex);
1485         if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET) num++;
1486     }
1487
1488     return num;
1489 }

```

### 8.4.7.25 NvCapGetCounterAvail()

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

```

1490  UUINT32
1491  NvCapGetCounterAvail(void)
1492  {
1493      UUINT32      availNVSpace;
1494      UUINT32      availRAMSpace;
1495      UUINT32      counterNVSpace;
1496      UUINT32      counterRAMSpace;
1497      UUINT32      persistentNum = NvCapGetPersistentNumber();
1498
1499      // Get the available space in NV storage
1500      availNVSpace = NvGetFreeByte();
1501
1502      if (persistentNum < MIN_EVICT_OBJECTS)
1503      {
1504          // Some space have to be reserved for evict object. Adjust availNVSpace.
1505          UUINT32      reserved = (MIN_EVICT_OBJECTS - persistentNum)
1506                               * NvGetEvictObjectSize();
1507          if (reserved > availNVSpace)
1508              availNVSpace = 0;
1509          else
1510              availNVSpace -= reserved;
1511      }
1512
1513      // Get the space needed to add a counter index to NV storage
1514      counterNVSpace = NvGetCounterSize();
1515
1516      // Compute the available space in RAM
1517      availRAMSpace = RAM_INDEX_SPACE - s_ramIndexSize;
1518
1519      // Compute the space needed to add a counter index to RAM storage
1520      // It takes an size field, a handle and sizeof(UINT64) for counter data
1521      counterRAMSpace = sizeof(UUINT32) + sizeof(TPM_HANDLE) + sizeof(UUINT64);
1522
1523      // Return the min of counter number in NV and in RAM
1524      if(availNVSpace / counterNVSpace > availRAMSpace / counterRAMSpace)
1525          return availRAMSpace / counterRAMSpace;
1526      else
1527          return availNVSpace / counterNVSpace;
1528  }

```

## 8.5 Object.c

### 8.5.1 Introduction

This file contains the functions that manage the object store of the TPM.

### 8.5.2 Includes and Data Definitions

```

1  #define OBJECT_C
2  #include "InternalRoutines.h"
3  #include <Platform.h>

```

### 8.5.3 Functions

#### 8.5.3.1 ObjectStartup()

This function is called at TPM2\_Startup() to initialize the object subsystem.

```

4  void
5  ObjectStartup(void)
6  {
7      UINT32      i;
8
9      // object slots initialization
10     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
11     {
12         //Set the slot to not occupied
13         s_objects[i].occupied = FALSE;
14     }
15     return;
16 }

```

#### 8.5.3.2 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()*.

```

17 void
18 ObjectCleanupEvict(void)
19 {
20     UINT32      i;
21
22     // This has to be iterated because a command may have two handles
23     // and they may both be persistent.
24     // This could be made to be more efficient so that a search is not needed.
25     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
26     {
27         // If an object is a temporary evict object, flush it from slot
28         if(s_objects[i].object.entity.attributes.evict == SET)
29             s_objects[i].occupied = FALSE;
30     }
31
32     return;
33 }

```

#### 8.5.3.3 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.

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

```

34  BOOL
35  ObjectIsPresent(
36      TPMT_DH_OBJECT  handle      // IN: handle to be checked
37  )
38  {
39      UINT32          slotIndex;    // index of object slot

```

```

40
41     pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
42
43     // The index in the loaded object array is found by subtracting the first
44     // object handle number from the input handle number. If the indicated
45     // slot is occupied, then indicate that there is already is a loaded
46     // object associated with the handle.
47     slotIndex = handle - TRANSIENT_FIRST;
48     if(slotIndex >= MAX_LOADED_OBJECTS)
49         return FALSE;
50
51     return s_objects[slotIndex].occupied;
52 }

```

#### 8.5.3.4 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.

Return Value	Meaning
TRUE	object is an HMAC, hash, or event sequence object
FALSE	object is not an HMAC, hash, or event sequence object

```

53 BOOL
54 ObjectIsSequence(
55     OBJECT          *object          // IN: handle to be checked
56 )
57 {
58     pAssert(object != NULL);
59     if( object->attributes.hmacSeq == SET
60        || object->attributes.hashSeq == SET
61        || object->attributes.eventSeq == SET)
62         return TRUE;
63     else
64         return FALSE;
65 }

```

#### 8.5.3.5 ObjectGet()

This function is used to find the object structure associated with a handle.

This function requires that *handle* references a loaded object.

```

66 OBJECT*
67 ObjectGet(
68     TPMI_DH_OBJECT    handle          // IN: handle of the object
69 )
70 {
71     pAssert( handle >= TRANSIENT_FIRST
72             && handle - TRANSIENT_FIRST < MAX_LOADED_OBJECTS);
73     pAssert(s_objects[handle - TRANSIENT_FIRST].occupied == TRUE);
74
75     // In this implementation, the handle is determined by the slot occupied by the
76     // object.
77     return &s_objects[handle - TRANSIENT_FIRST].object.entity;
78 }

```



### 8.5.3.6 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.

```

79  UUINT16
80  ObjectGetName(
81      TPMI_DH_OBJECT    handle,           // IN: handle of the object
82      BYTE               *name           // OUT: name of the object
83  )
84  {
85      OBJECT             *object = ObjectGet(handle);
86      if(object->publicArea.nameAlg == TPM_ALG_NULL)
87          return 0;
88
89      // Copy the Name data to the output
90      MemoryCopy(name, object->name.t.name, object->name.t.size);
91      return object->name.t.size;
92  }

```

### 8.5.3.7 ObjectGetNameAlg()

This function is used to get the Name algorithm of a object.

This function requires that *handle* references a loaded object.

```

93  TPMI_ALG_HASH
94  ObjectGetNameAlg(
95      TPMI_DH_OBJECT    handle           // IN: handle of the object
96  )
97  {
98      OBJECT             *object = ObjectGet(handle);
99
100     return object->publicArea.nameAlg;
101 }

```

### 8.5.3.8 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.

```

102 void
103 ObjectGetQualifiedName(
104     TPMI_DH_OBJECT    handle,           // IN: handle of the object
105     TPM2B_NAME        *qualifiedName    // OUT: qualified name of the object
106 )
107 {
108     OBJECT             *object = ObjectGet(handle);
109     if(object->publicArea.nameAlg == TPM_ALG_NULL)
110         qualifiedName->t.size = 0;
111     else
112         // Copy the name
113         *qualifiedName = object->qualifiedName;
114
115     return;

```

116 }

### 8.5.3.9 ObjectDataGetHierarchy()

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

```

117 TPMI_RH_HIERARCHY
118 ObjectDataGetHierarchy(
119     OBJECT          *object          // IN :object
120 )
121 {
122     if(object->attributes.spsHierarchy)
123     {
124         return TPM_RH_OWNER;
125     }
126     else if(object->attributes.epsHierarchy)
127     {
128         return TPM_RH_ENDORSEMENT;
129     }
130     else if(object->attributes.ppsHierarchy)
131     {
132         return TPM_RH_PLATFORM;
133     }
134     else
135     {
136         return TPM_RH_NULL;
137     }
138 }
139 
```

### 8.5.3.10 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 an pointer to an object.

This function requires that *handle* references a loaded object.

```

140 TPMI_RH_HIERARCHY
141 ObjectGetHierarchy(
142     TPMI_DH_OBJECT   handle          // IN :object handle
143 )
144 {
145     OBJECT          *object = ObjectGet(handle);
146
147     return ObjectDataGetHierarchy(object);
148 }

```

### 8.5.3.11 ObjectAllocateSlot()

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

Return Value	Meaning
TRUE	allocate success
FALSE	do not have free slot

```

149 static BOOL
150 ObjectAllocateSlot(
151     TPMI_DH_OBJECT   *handle,          // OUT: handle of allocated object

```

```

152     OBJECT          **object          // OUT: points to the allocated object
153 )
154 {
155     UINT32          i;
156
157     // find an unoccupied handle slot
158     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
159     {
160         if(!s_objects[i].occupied)    // If found a free slot
161         {
162             // Mark the slot as occupied
163             s_objects[i].occupied = TRUE;
164             break;
165         }
166     }
167     // If we reach the end of object slot without finding a free one, return
168     // error.
169     if(i == MAX_LOADED_OBJECTS) return FALSE;
170
171     *handle = i + TRANSIENT_FIRST;
172     *object = &s_objects[i].object.entity;
173
174     // Initialize the object attributes
175     MemorySet(&((*object)->attributes), 0, sizeof(OBJECT_ATTRIBUTES));
176
177     return TRUE;
178 }

```

### 8.5.3.12 ObjectLoad()

This function loads an object into an internal object structure. If an error is returned, the internal state is unchanged.

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

```

179 TPM_RC
180 ObjectLoad(
181     TPMI_RH_HIERARCHY    hierarchy,    // IN: hierarchy to which the object
182                             // belongs
183     TPMT_PUBLIC          *publicArea,   // IN: public area
184     TPMT_SENSITIVE       *sensitive,    // IN: sensitive area (may be null)
185     TPM2B_NAME           *name,        // IN: object's name (may be null)
186     TPM_HANDLE           parentHandle,  // IN: handle of parent
187     BOOL                 skipChecks,    // IN: flag to indicate if it is OK to
188                             // skip consistency checks.
189     TPMI_DH_OBJECT       *handle       // OUT: object handle
190 )
191 {
192     OBJECT          *object = NULL;
193     OBJECT          *parent = NULL;
194     TPM_RC          result = TPM_RC_SUCCESS;
195     TPM2B_NAME      parentQN;          // Parent qualified name
196
197     // Try to allocate a slot for new object
198     if(!ObjectAllocatesSlot(handle, &object))
199         return TPM_RC_OBJECT_MEMORY;
200

```

```

201 // Initialize public
202 object->publicArea = *publicArea;
203 if(sensitive != NULL)
204     object->sensitive = *sensitive;
205
206 // Are the consistency checks needed
207 if(!skipChecks)
208 {
209     // Check if key size matches
210     if(!CryptObjectIsPublicConsistent(&object->publicArea))
211     {
212         result = TPM_RC_KEY;
213         goto ErrorExit;
214     }
215     if(sensitive != NULL)
216     {
217         // Check if public type matches sensitive type
218         result = CryptObjectPublicPrivateMatch(object);
219         if(result != TPM_RC_SUCCESS)
220             goto ErrorExit;
221     }
222 }
223 object->attributes.publicOnly = (sensitive == NULL);
224
225 // If 'name' is NULL, then there is nothing left to do for this
226 // object as it has no qualified name and it is not a member of any
227 // hierarchy and it is temporary
228 if(name == NULL || name->t.size == 0)
229 {
230     object->qualifiedName.t.size = 0;
231     object->name.t.size = 0;
232     object->attributes.temporary = SET;
233     return TPM_RC_SUCCESS;
234 }
235 // If parent handle is a permanent handle, it is a primary or temporary
236 // object
237 if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
238 {
239     // initialize QN
240     parentQN.t.size = 4;
241
242     // for a primary key, parent qualified name is the handle of hierarchy
243     UINT32_TO_BYTE_ARRAY(parentHandle, parentQN.t.name);
244 }
245 else
246 {
247     // Get hierarchy and qualified name of parent
248     ObjectGetQualifiedName(parentHandle, &parentQN);
249
250     // Check for stClear object
251     parent = ObjectGet(parentHandle);
252     if( publicArea->objectAttributes.stClear == SET
253        || parent->attributes.stClear == SET)
254         object->attributes.stClear = SET;
255 }
256
257 object->name = *name;
258
259 // Compute object qualified name
260 ObjectComputeQualifiedName(&parentQN, publicArea->nameAlg,
261                          name, &object->qualifiedName);
262
263 // Any object in TPM_RH_NULL hierarchy is temporary
264 if(hierarchy == TPM_RH_NULL)
265 {
266     object->attributes.temporary = SET;

```

```

267     }
268     else if(parentQN.t.size == sizeof(TPM_HANDLE))
269     {
270         // Otherwise, if the size of parent's qualified name is the size of a
271         // handle, this object is a primary object
272         object->attributes.primary = SET;
273     }
274     switch(hierarchy)
275     {
276     case TPM_RH_PLATFORM:
277         object->attributes.ppsHierarchy = SET;
278         break;
279     case TPM_RH_OWNER:
280         object->attributes.spsHierarchy = SET;
281         break;
282     case TPM_RH_ENDORSEMENT:
283         object->attributes.epsHierarchy = SET;
284         break;
285     case TPM_RH_NULL:
286         break;
287     default:
288         pAssert(FALSE);
289         break;
290     }
291     return TPM_RC_SUCCESS;
292
293 ErrorExit:
294     ObjectFlush(*handle);
295     return result;
296 }

```

### 8.5.3.13 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.

```

297 static BOOL
298 AllocateSequenceSlot(
299     TPM_HANDLE      *newHandle,           // OUT: receives the allocated handle
300     HASH_OBJECT     **object,           // OUT: receives pointer to allocated
301                                     // object
302     TPM2B_AUTH      *auth               // IN: the authValue for the slot
303 )
304 {
305     OBJECT          *objectHash;        // the hash as an object
306
307     if(!ObjectAllocateSlot(newHandle, &objectHash))
308         return FALSE;
309
310     *object = (HASH_OBJECT *)objectHash;
311
312     // Validate that the proper location of the hash state data relative to the
313     // object state data.
314     pAssert(&((*object)->auth) == &objectHash->publicArea.authPolicy);
315
316     // Set the common values that a sequence object shares with an ordinary object
317     // The type is TPM_ALG_NULL
318     (*object)->type = TPM_ALG_NULL;
319
320     // This has no name algorithm and the name is the Empty Buffer
321     (*object)->nameAlg = TPM_ALG_NULL;
322
323     // Clear the attributes
324     MemorySet(&((*object)->objectAttributes), 0, sizeof(TPMA_OBJECT));

```

```

325
326 // A sequence object is DA exempt.
327 (*object)->objectAttributes.noDA = SET;
328
329 if(auth != NULL)
330 {
331     MemoryRemoveTrailingZeros(auth);
332     (*object)->auth = *auth;
333 }
334 else
335     (*object)->auth.t.size = 0;
336 return TRUE;
337 }

```

#### 8.5.3.14 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

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

```

338 TPM_RC
339 ObjectCreateHMACSequence(
340     TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
341     TPM_HANDLE       handle,           // IN: the handle associated with
342                                     // sequence object
343     TPM2B_AUTH       *auth,           // IN: authValue
344     TPMI_DH_OBJECT   *newHandle       // OUT: HMAC sequence object handle
345 )
346 {
347     HASH_OBJECT      *hmacObject;
348     OBJECT           *keyObject;
349
350     // Try to allocate a slot for new object
351     if(!AllocateSequenceSlot(newHandle, &hmacObject, auth))
352         return TPM_RC_OBJECT_MEMORY;
353
354     // Set HMAC sequence bit
355     hmacObject->attributes.hmacSeq = SET;
356
357     // Get pointer to the HMAC key object
358     keyObject = ObjectGet(handle);
359
360     CryptStartHMACSequence2B(hashAlg, &keyObject->sensitive.sensitive.bits.b,
361                                &hmacObject->state.hmacState);
362
363     return TPM_RC_SUCCESS;
364 }

```

#### 8.5.3.15 ObjectCreateHashSequence()

This function creates a hash sequence object.

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

```

365 TPM_RC
366 ObjectCreateHashSequence(
367     TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
368     TPM2B_AUTH       *auth,           // IN: authValue

```

```

369     TPMI_DH_OBJECT      *newHandle          // OUT: sequence object handle
370 )
371 {
372     HASH_OBJECT        *hashObject;
373
374     // Try to allocate a slot for new object
375     if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
376         return TPM_RC_OBJECT_MEMORY;
377
378     // Set hash sequence bit
379     hashObject->attributes.hashSeq = SET;
380
381     // Start hash for hash sequence
382     CryptStartHashSequence(hashAlg, &hashObject->state.hashState[0]);
383
384     return TPM_RC_SUCCESS;
385 }

```

### 8.5.3.16 ObjectCreateEventSequence()

This function creates an event sequence object.

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

```

386 TPM_RC
387 ObjectCreateEventSequence(
388     TPM2B_AUTH          *auth,                // IN: authValue
389     TPMI_DH_OBJECT      *newHandle          // OUT: sequence object handle
390 )
391 {
392     HASH_OBJECT        *hashObject;
393     UINT32              count;
394     TPM_ALG_ID          hash;
395
396     // Try to allocate a slot for new object
397     if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
398         return TPM_RC_OBJECT_MEMORY;
399
400     // Set the event sequence attribute
401     hashObject->attributes.eventSeq = SET;
402
403
404     // Initialize hash states for each implemented PCR algorithms
405     for(count = 0; (hash = CryptGetHashAlgByIndex(count)) != TPM_ALG_NULL; count++)
406     {
407         // If this is a _TPM_Init or _TPM_HashStart, the sequence object will
408         // not leave the TPM so it doesn't need the sequence handling
409         if(auth == NULL)
410             CryptStartHash(hash, &hashObject->state.hashState[count]);
411         else
412             CryptStartHashSequence(hash, &hashObject->state.hashState[count]);
413     }
414     return TPM_RC_SUCCESS;
415 }

```

### 8.5.3.17 ObjectTerminateEvent()

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

```

416 void
417 ObjectTerminateEvent(void)

```

```

418 {
419     HASH_OBJECT      *hashObject;
420     int              count;
421     BYTE             buffer[MAX_DIGEST_SIZE];
422     hashObject = (HASH_OBJECT *)ObjectGet(g_DRTMHandle);
423
424     // Don't assume that this is a proper sequence object
425     if(hashObject->attributes.eventSeq)
426     {
427         // If it is, close any open hash contexts. This is done in case
428         // the crypto implementation has some context values that need to be
429         // cleaned up (hygiene).
430         //
431         for(count = 0; CryptGetHashAlgByIndex(count) != TPM_ALG_NULL; count++)
432         {
433             CryptCompleteHash(&hashObject->state.hashState[count], 0, buffer);
434         }
435         // Flush sequence object
436         ObjectFlush(g_DRTMHandle);
437     }
438
439     g_DRTMHandle = TPM_RH_UNASSIGNED;
440 }

```

### 8.5.3.18 ObjectContextLoad()

This function loads an object from a saved object context.

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

```

441 TPM_RC
442 ObjectContextLoad(
443     OBJECT          *object,           // IN: object structure from saved
444                                     // context
445     TPMT_DH_OBJECT  *handle           // OUT: object handle
446 )
447 {
448     OBJECT          *newObject;
449
450     // Try to allocate a slot for new object
451     if(!ObjectAllocateSlot(handle, &newObject))
452         return TPM_RC_OBJECT_MEMORY;
453
454     // Copy input object data to internal structure
455     *newObject = *object;
456
457     return TPM_RC_SUCCESS;
458 }

```

### 8.5.3.19 ObjectFlush()

This function frees an object slot.

This function requires that the object is loaded.

```

459 void
460 ObjectFlush(
461     TPMT_DH_OBJECT  handle           // IN: handle to be freed
462 )
463 {
464     UINT32          index = handle - TRANSIENT_FIRST;

```



```

465     pAssert(ObjectIsPresent(handle));
466
467     // Mark the handle slot as unoccupied
468     s_objects[index].occupied = FALSE;
469
470     // With no attributes
471     MemorySet((BYTE*)&(s_objects[index].object.entity.attributes),
472             0, sizeof(OBJECT_ATTRIBUTES));
473     return;
474 }

```

### 8.5.3.20 ObjectFlushHierarchy()

This function is called to flush all the loaded transient objects associated with a hierarchy when the hierarchy is disabled.

```

475 void
476 ObjectFlushHierarchy(
477     TPMI_RH_HIERARCHY    hierarchy           // IN: hierarchy to be flush
478 )
479 {
480     UINT16                i;
481
482     // iterate object slots
483     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
484     {
485         if(s_objects[i].occupied)           // If found an occupied slot
486         {
487             switch(hierarchy)
488             {
489                 case TPM_RH_PLATFORM:
490                     if(s_objects[i].object.entity.attributes.ppsHierarchy == SET)
491                         s_objects[i].occupied = FALSE;
492                     break;
493                 case TPM_RH_OWNER:
494                     if(s_objects[i].object.entity.attributes.spsHierarchy == SET)
495                         s_objects[i].occupied = FALSE;
496                     break;
497                 case TPM_RH_ENDORSEMENT:
498                     if(s_objects[i].object.entity.attributes.epsHierarchy == SET)
499                         s_objects[i].occupied = FALSE;
500                     break;
501                 default:
502                     pAssert(FALSE);
503                     break;
504             }
505         }
506     }
507
508     return;
509 }
510 }

```

### 8.5.3.21 ObjectLoadEvict()

This function loads a persistent object into a transient object slot.

This function requires that *handle* is associated with a persistent object.

Error Returns	Meaning
TPM_RC_REFERENCE_H0	The persistent object does not exist
TPM_RC_OBJECT_MEMORY	no object slot
TPM_RC_HIERARCHY	the handle points to an existing persistent object belongs to a disabled hierarchy

```

511 TPM_RC
512 ObjectLoadEvict(
513     TPM_HANDLE          *handle          // IN:OUT: evict object handle. If
514                                     // success, it will be replace by
515                                     // the loaded object handle
516 )
517 {
518     TPM_RC          result;
519     TPM_HANDLE      evictHandle = *handle; // Save the evict handle
520     OBJECT          *object;
521
522     // Try to allocate a slot for new object
523     if(!ObjectAllocatesSlot(handle, &object))
524         return TPM_RC_OBJECT_MEMORY;
525
526     // Copy persistent object to transient object slot. A TPM_RC_REFERENCE_H0
527     // or TPM_RC_HIERARCHY error may be returned at this point
528     result = NvGetEvictObject(evictHandle, object);
529
530     // Free object slot if fails.
531     if(result != TPM_RC_SUCCESS)
532         ObjectFlush(*handle);
533
534     return result;
535 }

```

### 8.5.3.22 ObjectComputeName()

This function computes the Name of an object from its public area.

```

536 void
537 ObjectComputeName(
538     TPMT_PUBLIC          *publicArea,    // IN: public area of an object
539     TPM2B_NAME           *name          // OUT: name of the object
540 )
541 {
542     TPM2B_PUBLIC          marshalBuffer;
543     BYTE                  *buffer;      // auxiliary marshal buffer pointer
544     HASH_STATE            hashState;    // hash state
545
546     // if the nameAlg is NULL then there is no name.
547     if(publicArea->nameAlg == TPM_ALG_NULL)
548     {
549         name->t.size = 0;
550         return;
551     }
552     // Start hash stack
553     name->t.size = CryptStartHash(publicArea->nameAlg, &hashState);
554
555     // Marshal the public area into its canonical form
556     buffer = marshalBuffer.b.buffer;
557
558     marshalBuffer.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
559
560     // Adding public area

```

```

561     CryptUpdateDigest2B(&hashState, &marshalBuffer.b);
562
563     // Complete hash leaving room for the name algorithm
564     CryptCompleteHash(&hashState, name->t.size, &name->t.name[2]);
565
566     // set the nameAlg
567     UINT16_TO_BYTE_ARRAY(publicArea->nameAlg, name->t.name);
568     name->t.size += 2;
569     return;
570 }

```

### 8.5.3.23 ObjectComputeQualifiedName()

This function computes the qualified name of an object.

```

571 void
572 ObjectComputeQualifiedName(
573     TPM2B_NAME      *parentQN,           // IN: parent's qualified name
574     TPM_ALG_ID      nameAlg,            // IN: name hash
575     TPM2B_NAME      *name,              // IN: name of the object
576     TPM2B_NAME      *qualifiedName     // OUT: qualified name of the object
577 )
578 {
579     HASH_STATE      hashState;          // hash state
580
581     //     QN_A = hash_A (QN of parent || NAME_A)
582
583     // Start hash
584     qualifiedName->t.size = CryptStartHash(nameAlg, &hashState);
585
586     // Add parent's qualified name
587     CryptUpdateDigest2B(&hashState, &parentQN->b);
588
589     // Add self name
590     CryptUpdateDigest2B(&hashState, &name->b);
591
592     // Complete hash leaving room for the name algorithm
593     CryptCompleteHash(&hashState, qualifiedName->t.size,
594                     &qualifiedName->t.name[2]);
595     UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
596     qualifiedName->t.size += 2;
597     return;
598 }

```

### 8.5.3.24 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.

Return Value	Meaning
TRUE	if the object is a storage key
FALSE	if the object is not a storage key

```

599 BOOL
600 ObjectDataIsStorage(
601     TPMT_PUBLIC      *publicArea        // IN: public area of the object
602 )
603 {
604     if( CryptIsAsymAlgorithm(publicArea->type)           // must be asymmetric,
605         && publicArea->objectAttributes.restricted == SET // restricted,
606         && publicArea->objectAttributes.decrypt == SET   // decryption key

```

```

607     && publicArea->objectAttributes.sign == CLEAR           // can not be sign key
608     )
609     return TRUE;
610 else
611     return FALSE;
612 }

```

### 8.5.3.25 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.

Return Value	Meaning
TRUE	if the object is a storage key
FALSE	if the object is not a storage key

```

613 BOOL
614 ObjectIsStorage(
615     TPMI_DH_OBJECT      handle           // IN: object handle
616 )
617 {
618     OBJECT              *object = ObjectGet(handle);
619     return ObjectDataIsStorage(&object->publicArea);
620 }

```

### 8.5.3.26 ObjectCapGetLoaded()

This function returns a list of handles of loaded object, starting from *handle*. *Handle* must be in the range of valid transient object handles, but does not have to be the handle of a loaded transient object.

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

```

621 TPMI_YES_NO
622 ObjectCapGetLoaded(
623     TPMI_DH_OBJECT      handle,           // IN: start handle
624     UINT32              count,           // IN: count of returned handles
625     TPML_HANDLE         *handleList      // OUT: list of handle
626 )
627 {
628     TPMI_YES_NO         more = NO;
629     UINT32              i;
630
631     pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
632
633     // Initialize output handle list
634     handleList->count = 0;
635
636     // The maximum count of handles we may return is MAX_CAP_HANDLES
637     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
638
639     // Iterate object slots to get loaded object handles
640     for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
641     {
642         if(s_objects[i].occupied == TRUE)
643         {
644             // A valid transient object can not be the copy of a persistent object
645             pAssert(s_objects[i].object.entity.attributes.evict == CLEAR);

```

```

646
647     if(handleList->count < count)
648     {
649         // If we have not filled up the return list, add this object
650         // handle to it
651         handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
652         handleList->count++;
653     }
654     else
655     {
656         // If the return list is full but we still have loaded object
657         // available, report this and stop iterating
658         more = YES;
659         break;
660     }
661 }
662 }
663
664 return more;
665 }

```

### 8.5.3.27 ObjectCapGetTransientAvail()

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

```

666 UINT32
667 ObjectCapGetTransientAvail(void)
668 {
669     UINT32     i;
670     UINT32     num = 0;
671
672     // Iterate object slot to get the number of unoccupied slots
673     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
674     {
675         if(s_objects[i].occupied == FALSE) num++;
676     }
677
678     return num;
679 }

```

## 8.6 PCR.c

### 8.6.1 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.

### 8.6.2 Includes, Defines, and Data Definitions

```

1  #define PCR_C
2  #include "InternalRoutines.h"
3  #include <Platform.h>

```

The initial value of PCR attributes. The value of these fields should be consistent with PC Client specification. In this implementation, we assume the total number of implemented PCR is 24.

```

4  static const PCR_Attributes s_initAttributes[] =
5  {
6      // PCR 0 - 15, static RTM
7      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
8      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
9      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
10     {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
11
12     {0, 0x0F, 0x1F},          // PCR 16, Debug
13     {0, 0x10, 0x1C},          // PCR 17, Locality 4
14     {0, 0x10, 0x1C},          // PCR 18, Locality 3
15     {0, 0x10, 0x0C},          // PCR 19, Locality 2
16     {0, 0x14, 0x0E},          // PCR 20, Locality 1
17     {0, 0x14, 0x04},          // PCR 21, Dynamic OS
18     {0, 0x14, 0x04},          // PCR 22, Dynamic OS
19     {0, 0x0F, 0x1F},          // PCR 23, App specific
20     {0, 0x0F, 0x1F},          // PCR 24, testing policy
21 };

```

### 8.6.3 Functions

#### 8.6.3.1 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.

Return Value	Meaning
TRUE:	PCR belongs an auth group
FALSE:	PCR does not belong an auth group

```

22  BOOL
23  PCRBelongsAuthGroup(
24      TPMI_DH_PCR    handle,          // IN: handle of PCR
25      UINT32         *groupIndex      // OUT: group index if PCR belongs a
26                                      // group that allows authValue. If PCR
27                                      // does not belong to an auth group,
28                                      // the value in this parameter is
29                                      // invalid
30  )
31  {
32      #if NUM_AUTHVALUE_PCR_GROUP > 0
33          // Platform specification determines to which auth group a PCR belongs (if
34          // any). In this implementation, we assume there is only
35          // one auth group which contains PCR[20-22]. If the platform specification
36          // requires differently, the implementation should be changed accordingly
37          if(handle >= 20 && handle <= 22)
38          {
39              *groupIndex = 0;
40              return TRUE;
41          }
42      #endif
43      return FALSE;
44  }

```

### 8.6.3.2 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.

Return Value	Meaning
TRUE:	PCR belongs a policy group
FALSE:	PCR does not belong a policy group

```

46  BOOL
47  PCRBelongsPolicyGroup(
48      TPMI_DH_PCR    handle,           // IN: handle of PCR
49      UINT32         *groupIndex      // OUT: group index if PCR belongs a
50                                     // group that allows policy. If PCR
51                                     // does not belong to a policy group,
52                                     // the value in this parameter is
53                                     // invalid
54  )
55  {
56      #if NUM_POLICY_PCR_GROUP > 0
57          // Platform specification decides if a PCR belongs to a policy group and
58          // belongs to which group. In this implementation, we assume there is only
59          // one policy group which contains PCR20-22. If the platform specification
60          // requires differently, the implementation should be changed accordingly
61          if(handle >= 20 && handle <= 22)
62          {
63              *groupIndex = 0;
64              return TRUE;
65          }
66      #endif
67      return FALSE;
68  }

```

### 8.6.3.3 PCRBelongsTCBGroup()

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

Return Value	Meaning
TRUE:	PCR belongs to TCB group
FALSE:	PCR does not belong to TCB group

```

69  static BOOL
70  PCRBelongsTCBGroup(
71      TPMI_DH_PCR    handle           // IN: handle of PCR
72  )
73  {
74      #if ENABLE_PCR_NO_INCREMENT == YES
75          // Platform specification decides if a PCR belongs to a TCB group. In this
76          // implementation, we assume PCR[20-22] belong to TCB group. If the platform
77          // specification requires differently, the implementation should be
78          // changed accordingly
79          if(handle >= 20 && handle <= 22)
80              return TRUE;
81      #endif
82      return FALSE;
83  }
84

```

### 8.6.3.4 PCRPolicyIsAvailable()

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

Return Value	Meaning
TRUE	the PCR should be authorized by policy
FALSE	the PCR does not allow policy

```

85  BOOL
86  PCRPolicyIsAvailable(
87      TPMI_DH_PCR    handle           // IN: PCR handle
88  )
89  {
90      UINT32         groupIndex;
91
92      return PCRBelongsPolicyGroup(handle, &groupIndex);
93  }
```

### 8.6.3.5 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.

```

94  void
95  PCRGetAuthValue(
96      TPMI_DH_PCR    handle,           // IN: PCR handle
97      TPM2B_AUTH     *auth             // OUT: authValue of PCR
98  )
99  {
100     UINT32         groupIndex;
101
102     if(PCRBelongsAuthGroup(handle, &groupIndex))
103     {
104         *auth = gc.pcrAuthValues.auth[groupIndex];
105     }
106     else
107     {
108         auth->t.size = 0;
109     }
110
111     return;
112 }
```

### 8.6.3.6 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.

```

113  TPMI_ALG_HASH
114  PCRGetAuthPolicy(
115      TPMI_DH_PCR    handle,           // IN: PCR handle
116      TPM2B_DIGEST   *policy          // OUT: policy of PCR
117  )
118  {
119      UINT32         groupIndex;
120
121      if(PCRBelongsPolicyGroup(handle, &groupIndex))
122      {
123          *policy = gp.pcrPolicies.policy[groupIndex];
124          return gp.pcrPolicies.hashAlg[groupIndex];

```



```

125     }
126     else
127     {
128         policy->t.size = 0;
129         return TPM_ALG_NULL;
130     }
131 }

```

### 8.6.3.7 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.

```

132 void
133 PCRSimStart(void)
134 {
135     UINT32 i;
136     for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
137     {
138         gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
139         gp.pcrPolicies.policy[i].t.size = 0;
140     }
141
142     for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
143     {
144         gc.pcrAuthValues.auth[i].t.size = 0;
145     }
146
147     // We need to give an initial configuration on allocated PCR before
148     // receiving any TPM2_PCR_Allocate command to change this configuration
149     // When the simulation environment starts, we allocate all the PCRs
150     for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
151         gp.pcrAllocated.count++)
152     {
153         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
154             = CryptGetHashAlgByIndex(gp.pcrAllocated.count);
155
156         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
157             = PCR_SELECT_MAX;
158         for(i = 0; i < PCR_SELECT_MAX; i++)
159             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
160                 = 0xFF;
161     }
162
163     // Store the initial configuration to NV
164     NvWriteReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
165     NvWriteReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
166
167     return;
168 }

```

### 8.6.3.8 GetSavedPcrPointer()

This function returns the address of an array of state saved PCR based on the hash algorithm.

Return Value	Meaning
NULL	no such algorithm
not NULL	pointer to the 0th byte of the 0th PCR

```

169 static BYTE *
170 GetSavedPcrPointer (

```

```

171     TPM_ALG_ID      alg,          // IN: algorithm for bank
172     UINT32          pcrIndex      // IN: PCR index in PCR_SAVE
173 )
174 {
175     switch(alg)
176     {
177     #ifdef TPM_ALG_SHA1
178     case TPM_ALG_SHA1:
179         return gc.pcrSave.sha1[pcrIndex];
180         break;
181     #endif
182     #ifdef TPM_ALG_SHA256
183     case TPM_ALG_SHA256:
184         return gc.pcrSave.sha256[pcrIndex];
185         break;
186     #endif
187     #ifdef TPM_ALG_SHA384
188     case TPM_ALG_SHA384:
189         return gc.pcrSave.sha384[pcrIndex];
190         break;
191     #endif
192
193     #ifdef TPM_ALG_SHA512
194     case TPM_ALG_SHA512:
195         return gc.pcrSave.sha512[pcrIndex];
196         break;
197     #endif
198     #ifdef TPM_ALG_SM3_256
199     case TPM_ALG_SM3_256:
200         return gc.pcrSave.sm3_256[pcrIndex];
201         break;
202     #endif
203     default:
204         pAssert(FALSE);
205         break;
206     }
207
208     return NULL;
209 }

```

### 8.6.3.9 IsPcrAllocated()

This function indicates if a PCR number for the particular hash algorithm is allocated.

Return Value	Meaning
FALSE	PCR is not allocated
TRUE	PCR is allocated

```

210 static BOOL
211 IsPcrAllocated (
212     UINT32          pcr,          // IN: The number of the PCR
213     TPMI_ALG_HASH   hashAlg      // IN: The PCR algorithm
214 )
215 {
216     UINT32          i;
217
218     if(pcr >= IMPLEMENTATION_PCR)
219         return FALSE;
220
221     for(i = 0; i < gp.pcrAllocated.count; i++)
222     {
223         if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)

```

```

224     {
225         if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr/8])
226             & (1 << (pcr % 8))) != 0)
227             return TRUE;
228         else
229             return FALSE;
230     }
231 }
232
233 return FALSE;
234 }

```

### 8.6.3.10 GetPcrPointer()

This function returns the address of an array of PCR based on the hash algorithm.

Return Value	Meaning
NULL	no such algorithm
not NULL	pointer to the 0th byte of the 0th PCR

```

235 static BYTE *
236 GetPcrPointer (
237     TPM_ALG_ID      alg,           // IN: algorithm for bank
238     UINT32          pcrNumber      // IN: PCR number
239 )
240 {
241     // PCR must be allocated
242     pAssert(IsPcrAllocated(pcrNumber, alg) == TRUE);
243
244     switch(alg)
245     {
246 #ifdef TPM_ALG_SHA1
247         case TPM_ALG_SHA1:
248             return s_pcrs[pcrNumber].sha1Pcr;
249             break;
250 #endif
251 #ifdef TPM_ALG_SHA256
252         case TPM_ALG_SHA256:
253             return s_pcrs[pcrNumber].sha256Pcr;
254             break;
255 #endif
256 #ifdef TPM_ALG_SHA384
257         case TPM_ALG_SHA384:
258             return s_pcrs[pcrNumber].sha384Pcr;
259             break;
260 #endif
261 #ifdef TPM_ALG_SHA512
262         case TPM_ALG_SHA512:
263             return s_pcrs[pcrNumber].sha512Pcr;
264             break;
265 #endif
266 #ifdef TPM_ALG_SM3_256
267         case TPM_ALG_SM3_256:
268             return s_pcrs[pcrNumber].sm3_256Pcr;
269             break;
270 #endif
271         default:
272             pAssert(FALSE);
273             break;
274     }
275
276     return NULL;

```

277 }

### 8.6.3.11 IsPcrSelected()

This function indicates if an indicated PCR number is selected by the bit map in *selection*.

Return Value	Meaning
FALSE	PCR is not selected
TRUE	PCR is selected

```

278 static BOOL
279 IsPcrSelected (
280     UINT32          pcr,           // IN: The number of the PCR
281     TPMS_PCR_SELECTION *selection // IN: The selection structure
282 )
283 {
284     if(pcr >= IMPLEMENTATION_PCR)
285         return FALSE;
286     if(((selection->pcrSelect[pcr/8]) & (1 << (pcr % 8))) != 0)
287         return TRUE;
288     else
289         return FALSE;
290 }

```

### 8.6.3.12 FilterPcr()

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

```

291 static void
292 FilterPcr(
293     TPMS_PCR_SELECTION *selection // IN: input PCR selection
294 )
295 {
296     UINT32 i;
297     TPMS_PCR_SELECTION *allocated = NULL;
298
299     // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
300     for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
301         selection->pcrSelect[i] = 0;
302
303     // Find the internal configuration for the bank
304     for(i = 0; i < gp.pcrAllocated.count; i++)
305     {
306         if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
307         {
308             allocated = &gp.pcrAllocated.pcrSelections[i];
309             break;
310         }
311     }
312
313     for (i = 0; i < selection->sizeofSelect; i++)
314     {
315         if(allocated == NULL)
316         {
317             // If the required bank does not exist, clear input selection
318             selection->pcrSelect[i] = 0;
319         }
320         else
321             selection->pcrSelect[i] &= allocated->pcrSelect[i];
322     }
323 }

```

```

324     return;
325 }

```

### 8.6.3.13 PCRStartup()

This function initializes the PCR subsystem at TPM2\_Startup().

```

326 void
327 PCRStartup(
328     STARTUP_TYPE          type          // IN: startup type
329 )
330 {
331     UINT32    pcr, j;
332     UINT32    saveIndex = 0;
333
334     g_pcrReConfig = FALSE;
335
336     if(type != SU_RESUME)
337     {
338         // PCR generation counter is cleared at TPM_RESET and TPM_RESTART
339         gr.pcrCounter = 0;
340     }
341
342     // Initialize/Restore PCR values
343     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
344     {
345         BOOL        incrSaveIndex = FALSE;
346
347         // If PCR[0] it was already initialized by H-CRTM, then don't re-initialize
348         if(pcr == 0 && g_DrtmPreStartup)
349             continue;
350         // Iterate each hash algorithm bank
351         for(j = 0; j < gp.pcrAllocated.count; j++)
352         {
353             BYTE    *pcrData;
354             UINT32  pcrSize;
355             pcrSize =
356                 CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
357
358             if(IsPcrAllocated(pcr, gp.pcrAllocated.pcrSelections[j].hash))
359             {
360                 pcrData =
361                     GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
362
363                 if(type == SU_RESUME && s_initAttributes[pcr].stateSave == SET)
364                 {
365                     // Restore saved PCR value
366                     BYTE    *pcrSavedData;
367                     pcrSavedData = GetSavedPcrPointer(
368                         gp.pcrAllocated.pcrSelections[j].hash,
369                         saveIndex);
370                     MemoryCopy(pcrData, pcrSavedData, pcrSize);
371                     incrSaveIndex = TRUE;
372                 }
373                 else
374                 {
375                     // PCR was not restored by state save
376
377                     // If the reset locality of the PCR is 4, then
378                     // the reset value is all one's, otherwise it is
379                     // all zero.
380                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
381                         MemorySet(pcrData, 0xFF, pcrSize);
382                     else
383                         {

```

```

383         // Don't reset PCR[0] if H-CRTM was done
384         if(pcr != 0 || !g_DrtmPreStartup)
385             MemorySet(pcrData, 0, pcrSize);
386     }
387 }
388 }
389 }
390 if(incrSaveIndex == TRUE)
391     saveIndex++;
392 }
393
394 // Reset authValues
395 if(type != SU_RESUME)
396 {
397     for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
398     {
399         gc.pcrAuthValues.auth[j].t.size = 0;
400     }
401 }
402
403 }

```

#### 8.6.3.14 PCRStateSave()

This function is used to save the PCR values that will be restored on TPM Resume.

```

404 void
405 PCRStateSave(
406     TPM_SU          type          // IN: startup type
407 )
408 {
409     UINT32          pcr, j;
410     UINT32          saveIndex = 0;
411
412     // if state save CLEAR, nothing to be done. Return here
413     if(type == TPM_SU_CLEAR) return;
414
415     // Copy PCR values to the structure that should be saved to NV
416     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
417     {
418         BOOL          incrSaveIndex = FALSE;
419
420         // Iterate each hash algorithm bank
421         for(j = 0; j < gp.pcrAllocated.count; j++)
422         {
423             BYTE      *pcrData;
424             UINT32    pcrSize;
425             pcrSize
426                 = CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
427
428             if(IsPcrAllocated(pcr, gp.pcrAllocated.pcrSelections[j].hash))
429             {
430                 pcrData
431                     = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
432                 if(s_initAttributes[pcr].stateSave == SET)
433                 {
434                     // Restore saved PCR value
435                     BYTE      *pcrSavedData;
436                     pcrSavedData
437                         = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
438                                             saveIndex);
439                     MemoryCopy(pcrSavedData, pcrData, pcrSize);
440                     incrSaveIndex = TRUE;
441                 }

```

```

442     }
443   }
444   if(incrSaveIndex == TRUE)
445     saveIndex++;
446 }
447
448 return;
449 }

```

### 8.6.3.15 PCRIsStateSaved()

This function indicates if the selected PCR is a PCR that is state saved on TPM2\_Shutdown(STATE). The return value is based on PCR attributes.

Return Value	Meaning
TRUE	PCR is state saved
FALSE	PCR is not state saved

```

450 BOOL
451 PCRIsStateSaved(
452     TPMI_DH_PCR      handle           // IN: PCR handle to be extended
453 )
454 {
455     UINT32            pcr = handle - PCR_FIRST;
456
457     if(s_initAttributes[pcr].stateSave == SET)
458         return TRUE;
459     else
460         return FALSE;
461 }

```

### 8.6.3.16 PCRIsResetAllowed()

This function indicates if a PCR may be reset by the current command locality. The return value is based on PCR attributes, and not the PCR allocation.

Return Value	Meaning
TRUE	extend is allowed
FALSE	extend is not allowed

```

462 BOOL
463 PCRIsResetAllowed(
464     TPMI_DH_PCR      handle           // IN: PCR handle to be extended
465 )
466 {
467     UINT8            commandLocality;
468     UINT8            localityBits = 1;
469     UINT32            pcr = handle - PCR_FIRST;
470
471     // Check for the locality
472     commandLocality = _plat_LocalityGet();
473     localityBits = localityBits << commandLocality;
474     if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
475         return FALSE;
476     else
477         return TRUE;
478 }
479 }

```

### 8.6.3.17 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.

```

480 void
481 PCRChanged(
482     TPM_HANDLE     pcrHandle           // IN: the handle of the PCR that changed.
483 )
484 {
485     // For the reference implementation, the only change that does not cause
486     // increment is a change to a PCR in the TCB group.
487     if(!PCRBelongsTCBGroup(pcrHandle))
488         gr.pcrCounter++;
489 }

```

### 8.6.3.18 PCRIsExtendAllowed()

This function indicates a PCR may be extended at the current command locality. The return value is based on PCR attributes, and not the PCR allocation.

Return Value	Meaning
TRUE	extend is allowed
FALSE	extend is not allowed

```

490 BOOL
491 PCRIsExtendAllowed(
492     TPMT_DH_PCR     handle           // IN: PCR handle to be extended
493 )
494 {
495     UINT8           commandLocality;
496     UINT8           localityBits = 1;
497     UINT32          pcr = handle - PCR_FIRST;
498
499     // Check for the locality
500     commandLocality = _plat_LocalityGet();
501     localityBits = localityBits << commandLocality;
502     if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
503         return FALSE;
504     else
505         return TRUE;
506 }
507 }

```

### 8.6.3.19 PCRExtend()

This function is used to extend a PCR in a specific bank.

```

508 void
509 PCRExtend(
510     TPMT_DH_PCR     handle,         // IN: PCR handle to be extended
511     TPMT_ALG_HASH   hash,         // IN: hash algorithm of PCR
512     UINT32          size,         // IN: size of data to be extended
513     BYTE            *data         // IN: data to be extended
514 )
515 {
516     UINT32          pcr = handle - PCR_FIRST;
517     BYTE            *pcrData;
518     HASH_STATE      hashState;
519     UINT16          pcrSize;

```



```

520
521 // Extend PCR if it is allocated
522 if(IsPcrAllocated(pcr, hash))
523 {
524     pcrSize = CryptGetHashDigestSize(hash);
525     pcrData = GetPcrPointer(hash, pcr);
526     CryptStartHash(hash, &hashState);
527     CryptUpdateDigest(&hashState, pcrSize, pcrData);
528     CryptUpdateDigest(&hashState, size, data);
529     CryptCompleteHash(&hashState, pcrSize, pcrData);
530
531     // If PCR does not belong to TCB group, increment PCR counter
532     if(!PCRBelongsTCBGroup(handle))
533         gr.pcrCounter++;
534 }
535
536 return;
537 }

```

### 8.6.3.20 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.

```

538 void
539 PCRComputeCurrentDigest(
540     TPMI_ALG_HASH      hashAlg,           // IN: hash algorithm to compute digest
541     TPML_PCR_SELECTION *selection,       // IN/OUT: PCR selection (filtered on
542                                           // output)
543     TPM2B_DIGEST       *digest           // OUT: digest
544 )
545 {
546     HASH_STATE          hashState;
547     TPMS_PCR_SELECTION *select;
548     BYTE                *pcrData;       // will point to a digest
549     UINT32              pcrSize;
550     UINT32              pcr;
551     UINT32              i;
552
553     // Initialize the hash
554     digest->t.size = CryptStartHash(hashAlg, &hashState);
555     pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
556
557     // Iterate through the list of PCR selection structures
558     for(i = 0; i < selection->count; i++)
559     {
560         // Point to the current selection
561         select = &selection->pcrSelections[i]; // Point to the current selection
562         FilterPcr(select); // Clear out the bits for unimplemented PCR
563
564         // Need the size of each digest
565         pcrSize = CryptGetHashDigestSize(selection->pcrSelections[i].hash);
566
567         // Iterate through the selection
568         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
569         {
570             if(IsPcrSelected(pcr, select)) // Is this PCR selected
571             {
572                 // Get pointer to the digest data for the bank
573                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
574                 CryptUpdateDigest(&hashState, pcrSize, pcrData); // add to digest
575             }
576         }
577     }

```

```

578     // Complete hash stack
579     CryptCompleteHash2B(&hashState, &digest->b);
580
581     return;
582 }

```

### 8.6.3.21 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.

```

583 void
584 PCRRead(
585     TPML_PCR_SELECTION    *selection,    // IN/OUT: PCR selection (filtered on
586                                     //      output)
587     TPML_DIGEST          *digest,       // OUT: digest
588     UINT32                *pcrCounter   // OUT: the current value of PCR
589                                     //      generation number
590 )
591 {
592     TPMS_PCR_SELECTION    *select;
593     BYTE                  *pcrData;     // will point to a digest
594     UINT32                pcr;
595     UINT32                i;
596
597     digest->count = 0;
598
599     // Iterate through the list of PCR selection structures
600     for(i = 0; i < selection->count; i++)
601     {
602         // Point to the current selection
603         select = &selection->pcrSelections[i]; // Point to the current selection
604         FilterPcr(select); // Clear out the bits for unimplemented PCR
605
606         // Iterate through the selection
607         for (pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
608         {
609             if(IsPcrSelected(pcr, select)) // Is this PCR selected
610             {
611                 // Check if number of digest exceed upper bound
612                 if(digest->count > 7)
613                 {
614                     // Clear rest of the current select bitmap
615                     while( pcr < IMPLEMENTATION_PCR
616                            // do not round up!
617                            && (pcr / 8) < select->sizeofSelect)
618                     {
619                         // do not round up!
620                         select->pcrSelect[pcr/8] &= (BYTE) ~(1 << (pcr % 8));
621                         pcr++;
622                     }
623                     // Exit inner loop
624                     break;;
625                 }
626                 // Need the size of each digest
627                 digest->digests[digest->count].t.size =
628                     CryptGetHashDigestSize(selection->pcrSelections[i].hash);
629
630                 // Get pointer to the digest data for the bank
631                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
632                 // Add to the data to digest
633                 MemoryCopy(digest->digests[digest->count].t.buffer,
634                             pcrData,
635                             digest->digests[digest->count].t.size);

```

```

636         digest->count++;
637     }
638 }
639 // If we exit inner loop because we have exceed the output upper bound
640 if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
641 {
642     // Clear rest of the selection
643     while(i < selection->count)
644     {
645         MemorySet(selection->pcrSelections[i].pcrSelect, 0,
646                 selection->pcrSelections[i].sizeofSelect);
647         i++;
648     }
649     // exit outer loop
650     break;
651 }
652 }
653
654 *pcrCounter = gr.pcrCounter;
655
656 return;
657 }

```

### 8.6.3.22 PCRAllocate()

This function is used to change the PCR allocation.

Return Value	Meaning
YES	allocate success
NO	allocate fail

```

658 TPMI_YES_NO
659 PCRAllocate(
660     TPML_PCR_SELECTION *allocate,           // IN: required allocation
661     UINT32 maxPCR,           // OUT: Maximum number of PCR
662     UINT32 *sizeNeeded,     // OUT: required space
663     UINT32 *sizeAvailable   // OUT: available space
664 )
665 {
666     UINT32 i, j, k;
667     TPML_PCR_SELECTION newAllocate;
668
669     // Create the expected new PCR allocation based on the existing allocation
670     // and the new input:
671     // 1. if a PCR bank does not appear in the new allocation, the existing
672     //    allocation of this PCR bank will be preserved.
673     // 2. if a PCR bank appears multiple times in the new allocation, only the
674     //    last one will be in effect.
675     newAllocate = gp.pcrAllocated;
676     for(i = 0; i < allocate->count; i++)
677     {
678         for(j = 0; j < newAllocate.count; j++)
679         {
680             // If hash matches, the new allocation covers the old allocation
681             // for this particular bank.
682             // The assumption is the initial PCR allocation (from manufacture)
683             // has all the supported hash algorithms allocated. So there must
684             // be a match for any new bank allocation from the input.
685             if(newAllocate.pcrSelections[j].hash ==
686                allocate->pcrSelections[i].hash)
687             {
688                 newAllocate.pcrSelections[j] = allocate->pcrSelections[i];

```

```

689         break;
690     }
691 }
692 // The j loop must exit with a match.
693 pAssert(j < newAllocate.count);
694 }
695
696 // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
697 *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
698 if(*maxPCR > IMPLEMENTATION_PCR)
699     *maxPCR = IMPLEMENTATION_PCR;
700
701 // Compute required size for allocation
702 *sizeNeeded = 0;
703 for(i = 0; i < newAllocate.count; i++)
704 {
705     UINT32    digestSize
706             = CryptGetHashDigestSize(newAllocate.pcrSelections[i].hash);
707     for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
708     {
709         BYTE    mask = 1;
710         for(k = 0; k < 8; k++)
711         {
712             if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
713                 *sizeNeeded += digestSize;
714             mask = mask << 1;
715         }
716     }
717 }
718
719 // In this particular implementation, we always have enough space to
720 // allocate PCR. Different implementation may return a sizeAvailable less
721 // than the sizeNeed.
722 *sizeAvailable = sizeof(s_pcrs);
723
724 // Save the required allocation to NV. Note that after NV is written, the
725 // PCR allocation in NV is no longer consistent with the RAM data
726 // gp.pcrAllocated. The NV version reflect the allocate after next
727 // TPM_RESET, while the RAM version reflects the current allocation
728 NvWriteReserved(NV_PCR_ALLOCATED, &newAllocate);
729
730 return YES;
731 }
732 }

```

### 8.6.3.23 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.

```

733 void
734 PCRSetValue(
735     TPM_HANDLE    handle,           // IN: the handle of the PCR to set
736     INT8          initialValue      // IN: the value to set
737 )
738 {
739     int          i;
740     UINT32      pcr = handle - PCR_FIRST;
741     TPMT_ALG_HASH hash;
742     UINT16      digestSize;
743     BYTE        *pcrData;
744
745     // Iterate supported PCR bank algorithms to reset
746     for(i = 0; i < HASH_COUNT; i++)

```

```

747     {
748         hash = CryptGetHashAlgByIndex(i);
749         // Prevent runaway
750         if(hash == TPM_ALG_NULL)
751             break;
752
753         // If the PCR is allocated
754         if(IsPcrAllocated(pcr, gp.pcrAllocated.pcrSelections[i].hash))
755         {
756             // Get a pointer to the data
757             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
758
759             // And the size of the digest
760             digestSize = CryptGetHashDigestSize(hash);
761
762             // Set the LSO to the input value
763             pcrData[digestSize - 1] = initialValue;
764
765             // Sign extend
766             if(initialValue >= 0)
767                 MemorySet(pcrData, 0, digestSize - 1);
768             else
769                 MemorySet(pcrData, -1, digestSize - 1);
770         }
771     }
772 }

```

#### 8.6.3.24 PCRResetDynamics

This function is used to reset a dynamic PCR to 0. This function is used in DRTM sequence.

```

773 void
774 PCRResetDynamics(void)
775 {
776     UINT32          pcr, i;
777
778     // Initialize PCR values
779     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
780     {
781         // Iterate each hash algorithm bank
782         for(i = 0; i < gp.pcrAllocated.count; i++)
783         {
784             BYTE      *pcrData;
785             UINT32    pcrSize;
786             pcrSize
787                 = CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
788
789             if(IsPcrAllocated(pcr, gp.pcrAllocated.pcrSelections[i].hash))
790             {
791                 pcrData
792                     = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
793
794                 // Reset PCR
795                 // Any PCR can be reset by locality 4 should be reset to 0
796                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
797                     MemorySet(pcrData, 0, pcrSize);
798             }
799         }
800     }
801     return;
802 }

```

### 8.6.3.25 PCRCapGetAllocation()

This function is used to get the current allocation of PCR banks.

Return Value	Meaning
YES:	if the return count is 0
NO:	if the return count is not 0

```

803  TPMI_YES_NO
804  PCRCapGetAllocation(
805      UINT32          count,           // IN: count of return
806      TPML_PCR_SELECTION *pcrSelection // OUT: PCR allocation list
807  )
808  {
809      if(count == 0)
810      {
811          pcrSelection->count = 0;
812          return YES;
813      }
814      else
815      {
816          *pcrSelection = gp.pcrAllocated;
817          return NO;
818      }
819  }

```

### 8.6.3.26 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```

820  static void
821  PCRSetSelectBit(
822      UINT32          pcr,           // IN: PCR number
823      BYTE            *bitmap       // OUT: bit map to be set
824  )
825  {
826      bitmap[pcr / 8] |= (1 << (pcr % 8));
827      return;
828  }

```

### 8.6.3.27 PCRGetProperty()

This function returns the selected PCR property.

Return Value	Meaning
TRUE	the property type is implemented
FALSE	the property type os not implemented

```

829  static BOOL
830  PCRGetProperty(
831      TPM_PT_PCR          property,
832      TPMS_TAGGED_PCR_SELECT *select
833  )
834  {
835      UINT32          pcr;
836      UINT32          groupIndex;
837
838      select->tag = property;

```

```

839 // Always set the bitmap to be the size of all PCR
840 select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
841
842 // Initialize bitmap
843 MemorySet(select->pcrSelect, 0, select->sizeofSelect);
844
845 // Collecting properties
846 for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
847 {
848     switch(property)
849     {
850     case TPM_PT_PCR_SAVE:
851         if(s_initAttributes[pcr].stateSave == SET)
852             PCRSetSelectBit(pcr, select->pcrSelect);
853         break;
854     case TPM_PT_PCR_EXTEND_L0:
855         if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
856             PCRSetSelectBit(pcr, select->pcrSelect);
857         break;
858     case TPM_PT_PCR_RESET_L0:
859         if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
860             PCRSetSelectBit(pcr, select->pcrSelect);
861         break;
862     case TPM_PT_PCR_EXTEND_L1:
863         if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
864             PCRSetSelectBit(pcr, select->pcrSelect);
865         break;
866     case TPM_PT_PCR_RESET_L1:
867         if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
868             PCRSetSelectBit(pcr, select->pcrSelect);
869         break;
870     case TPM_PT_PCR_EXTEND_L2:
871         if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
872             PCRSetSelectBit(pcr, select->pcrSelect);
873         break;
874     case TPM_PT_PCR_RESET_L2:
875         if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
876             PCRSetSelectBit(pcr, select->pcrSelect);
877         break;
878     case TPM_PT_PCR_EXTEND_L3:
879         if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
880             PCRSetSelectBit(pcr, select->pcrSelect);
881         break;
882     case TPM_PT_PCR_RESET_L3:
883         if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
884             PCRSetSelectBit(pcr, select->pcrSelect);
885         break;
886     case TPM_PT_PCR_EXTEND_L4:
887         if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
888             PCRSetSelectBit(pcr, select->pcrSelect);
889         break;
890     case TPM_PT_PCR_RESET_L4:
891         if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
892             PCRSetSelectBit(pcr, select->pcrSelect);
893         break;
894     case TPM_PT_PCR_DRTM_RESET:
895         // DRTM reset PCRs are the PCR reset by locality 4
896         if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
897             PCRSetSelectBit(pcr, select->pcrSelect);
898         break;
899 #if NUM_POLICY_PCR_GROUP > 0
900     case TPM_PT_PCR_POLICY:
901         if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
902             PCRSetSelectBit(pcr, select->pcrSelect);
903         break;
904 #endif

```

```

905 #if NUM_AUTHVALUE_PCR_GROUP > 0
906     case TPM_PT_PCR_AUTH:
907         if (PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
908             PCRSetSelectBit(pcr, select->pcrSelect);
909         break;
910 #endif
911 #if ENABLE_PCR_NO_INCREMENT == YES
912     case TPM_PT_PCR_NO_INCREMENT:
913         if (PCRBelongsTCBGroup(pcr + PCR_FIRST))
914             PCRSetSelectBit(pcr, select->pcrSelect);
915         break;
916 #endif
917     default:
918         // If property is not supported, stop scanning PCR attributes
919         // and return.
920         return FALSE;
921         break;
922 }
923 }
924 return TRUE;
925 }

```

### 8.6.3.28 PCRCapGetProperties()

This function returns a list of PCR properties starting at *property*.

Return Value	Meaning
YES:	if no more property is available
NO:	if there are more properties not reported

```

926 TPMI_YES_NO
927 PCRCapGetProperties(
928     TPM_PT_PCR          property,      // IN: the starting PCR property
929     UINT32              count,        // IN: count of returned
930     // properties
931     TPML_TAGGED_PCR_PROPERTY *select  // OUT: PCR select
932 )
933 {
934     TPMI_YES_NO    more = NO;
935     UINT32         i;
936
937     // Initialize output property list
938     select->count = 0;
939
940     // The maximum count of properties we may return is MAX_PCR_PROPERTIES
941     if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
942
943     // TPM_PT_PCR_FIRST is defined as 0 in spec. It ensures that property
944     // value would never be less than TPM_PT_PCR_FIRST
945     pAssert(TPM_PT_PCR_FIRST == 0);
946
947     // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
948     // implemented on the TPM.
949     for(i = property; i <= TPM_PT_PCR_LAST; i++)
950     {
951         if(select->count < count)
952         {
953             // If we have not filled up the return list, add more properties to it
954             if(PCRGetProperty(i, &select->pcrProperty[select->count]))
955                 // only increment if the property is implemented
956                 select->count++;
957         }
958     }
959 }

```



```

958     else
959     {
960         // If the return list is full but we still have properties
961         // available, report this and stop iterating.
962         more = YES;
963         break;
964     }
965 }
966 return more;
967 }

```

### 8.6.3.29 PCRCapGetHandles()

This function is used to get a list of handles of PCR, started from *handle*. If *handle* exceeds the maximum PCR handle range, an empty list will be returned and the return value will be NO.

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

```

968 TPMI_YES_NO
969 PCRCapGetHandles(
970     TPMI_DH_PCR          handle,          // IN: start handle
971     UINT32               count,          // IN: count of returned handles
972     TPML_HANDLE         *handleList     // OUT: list of handle
973 )
974 {
975     TPMI_YES_NO    more = NO;
976     UINT32        i;
977
978     pAssert(HandleGetType(handle) == TPM_HT_PCR);
979
980     // Initialize output handle list
981     handleList->count = 0;
982
983     // The maximum count of handles we may return is MAX_CAP_HANDLES
984     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
985
986     // Iterate PCR handle range
987     for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
988     {
989         if(handleList->count < count)
990         {
991             // If we have not filled up the return list, add this PCR
992             // handle to it
993             handleList->handle[handleList->count] = i + PCR_FIRST;
994             handleList->count++;
995         }
996         else
997         {
998             // If the return list is full but we still have PCR handle
999             // available, report this and stop iterating
1000             more = YES;
1001             break;
1002         }
1003     }
1004     return more;
1005 }

```

## 8.7 PP.c

### 8.7.1 Introduction

This file contains the functions that support the physical presence operations of the TPM.

### 8.7.2 Includes

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

### 8.7.3 Functions

#### 8.7.3.1 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(void)
4 {
5     // Clear all the PP commands
6     MemorySet(&gp.ppList, 0,
7             ((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7) / 8);
8
9     // TPM_CC_PP_Commands always requires PP
10    if(CommandIsImplemented(TPM_CC_PP_Commands))
11        PhysicalPresenceCommandSet(TPM_CC_PP_Commands);
12
13    // Write PP list to NV
14    NvWriteReserved(NV_PP_LIST, &gp.ppList);
15
16    return;
17 }
```

#### 8.7.3.2 PhysicalPresenceCommandSet()

This function is used to indicate a command that requires PP confirmation.

```
18 void
19 PhysicalPresenceCommandSet(
20     TPM_CC    commandCode    // IN: command code
21 )
22 {
23     UINT32    bitPos;
24
25     // Assume command is implemented. It should be checked before this
26     // function is called
27     pAssert(CommandIsImplemented(commandCode));
28
29     // If the command is not a PP command, ignore it
30     if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
31         return;
32
33     bitPos = commandCode - TPM_CC_PP_FIRST;
34
35     // Set bit
36     gp.ppList[bitPos/8] |= 1 << (bitPos % 8);
37 }
```

```

38     return;
39 }

```

### 8.7.3.3 PhysicalPresenceCommandClear()

This function is used to indicate a command that no longer requires PP confirmation.

```

40 void
41 PhysicalPresenceCommandClear(
42     TPM_CC      commandCode      // IN: command code
43 )
44 {
45     UINT32      bitPos;
46
47     // Assume command is implemented. It should be checked before this
48     // function is called
49     pAssert(CommandIsImplemented(commandCode));
50
51     // If the command is not a PP command, ignore it
52     if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
53         return;
54
55     // if the input code is TPM_CC_PP_Commands, it can not be cleared
56     if(commandCode == TPM_CC_PP_Commands)
57         return;
58
59     bitPos = commandCode - TPM_CC_PP_FIRST;
60
61     // Set bit
62     gp.ppList[bitPos/8] |= (1 << (bitPos % 8));
63     // Flip it to off
64     gp.ppList[bitPos/8] ^= (1 << (bitPos % 8));
65
66     return;
67 }

```

### 8.7.3.4 PhysicalPresencelsRequired()

This function indicates if PP confirmation is required for a command.

Return Value	Meaning
TRUE	if physical presence is required
FALSE	if physical presence is not required

```

68 BOOL
69 PhysicalPresenceIsRequired(
70     TPM_CC      commandCode      // IN: command code
71 )
72 {
73     UINT32      bitPos;
74
75     // if the input commandCode is not a PP command, return FALSE
76     if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
77         return FALSE;
78
79     bitPos = commandCode - TPM_CC_PP_FIRST;
80
81     // Check the bit map. If the bit is SET, PP authorization is required
82     return ((gp.ppList[bitPos/8] & (1 << (bitPos % 8))) != 0);
83
84 }

```

### 8.7.3.5 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*.

Return Value	Meaning
YES	if there are more command codes available
NO	all the available command codes have been returned

```

85  TPMI_YES_NO
86  PhysicalPresenceCapGetCCList(
87      TPM_CC          commandCode,          // IN: start command code
88      UINT32          count,                // IN: count of returned TPM_CC
89      TPML_CC         *commandList         // OUT: list of TPM_CC
90  )
91  {
92      TPMI_YES_NO     more = NO;
93      UINT32          i;
94
95      // Initialize output handle list
96      commandList->count = 0;
97
98      // The maximum count of command we may return is MAX_CAP_CC
99      if(count > MAX_CAP_CC) count = MAX_CAP_CC;
100
101     // Collect PP commands
102     for(i = commandCode; i <= TPM_CC_PP_LAST; i++)
103     {
104         if(PhysicalPresenceIsRequired(i))
105         {
106             if(commandList->count < count)
107             {
108                 // If we have not filled up the return list, add this command
109                 // code to it
110                 commandList->commandCodes[commandList->count] = i;
111                 commandList->count++;
112             }
113             else
114             {
115                 // If the return list is full but we still have PP command
116                 // available, report this and stop iterating
117                 more = YES;
118                 break;
119             }
120         }
121     }
122     return more;
123 }

```

## 8.8 Session.c

### 8.8.1 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 TPM), and save it. Since loading the oldest session also eliminates its *contextID* value from *contextArray*, there TPM will always be able to load and save the oldest existing context.

In the worst case, software may have to load and save several contexts in order to save an additional one. This should happen very infrequently.

When the TPM searches *contextArray* and finds that none of the *contextIDs* match the low-order 16-bits of *contextCount*, the TPM can copy the low bits to the *contextArray* associated with the session, and increment *contextCount*.

There is one entry in *contextArray* for each of the active sessions allowed by the TPM implementation. This array contains either a context count, an index, or a value indicating the slot is available (0).

The index into the *contextArray* is the handle for the session with the region selector byte of the session set to zero. If an entry in *contextArray* contains 0, then the corresponding handle may be assigned to a session. If the entry contains a value that is less than or equal to the number of loaded sessions for the TPM, then the array entry is the slot in which the context is loaded.

EXAMPLE: If the TPM allows 8 loaded sessions, then the slot numbers would be 1-8 and a *contextArray* value in that range would represent the loaded session.

NOTE: When the TPM firmware determines that the array entry is for a loaded session, it will subtract 1 to create the zero-based slot number.

There is one significant corner case in this scheme. When the *contextCount* is equal to a value in the *contextArray*, the oldest session needs to be recycled or flushed. In order to recycle the session, it must be loaded. To be loaded, there must be an available slot. Rather than require that a spare slot be available all the time, the TPM will check to see if the *contextCount* is equal to some value in the *contextArray* when a session is created. This prevents the last session slot from being used when it is likely that a session will need to be recycled.

If a TPM with both 1.2 and 2.0 functionality uses this scheme for both 1.2 and 2.0 sessions, and the list of active contexts is read with `TPM_GetCapability()`, the TPM will create 32-bit representations of the list that contains 16-bit values (the `TPM2_GetCapability()` returns a list of handles for active sessions rather than a list of *contextID*). The full *contextID* has high-order bits that are either the same as the current *contextCount* or one less. It is one less if the 16-bits of the *contextArray* has a value that is larger than the low-order 16 bits of *contextCount*.

## 8.8.2 Includes, Defines, and Local Variables

```

1  #define SESSION_C
2  #include "InternalRoutines.h"
3  #include "Platform.h"
4  #include "SessionProcess_fp.h"

```

### 8.8.3 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*.

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(void)
7  {
8      CONTEXT_SLOT    lowBits;
9      CONTEXT_SLOT    entry;
10     CONTEXT_SLOT    smallest = ((CONTEXT_SLOT) ~0);
11     UINT32    i;
12
13     // Set oldestSaveContext to a value indicating none assigned
14     s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
15
16     lowBits = (CONTEXT_SLOT)gr.contextCounter;
17     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
18     {
19         entry = gr.contextArray[i];
20
21         // only look at entries that are saved contexts
22         if(entry > MAX_LOADED_SESSIONS)
23         {
24             // Use a less than or equal in case the oldest
25             // is brand new (= lowBits-1) and equal to our initial
26             // value for smallest.
27             if(((CONTEXT_SLOT) (entry - lowBits)) <= smallest)
28             {
29                 smallest = (entry - lowBits);
30                 s_oldestSavedSession = i;
31             }
32         }
33     }
34     // When we finish, either the s_oldestSavedSession still has its initial
35     // value, or it has the index of the oldest saved context.
36 }

```

### 8.8.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2\_Startup().

```

37  void
38  SessionStartup(
39      STARTUP_TYPE    type
40  )
41  {
42      UINT32    i;
43
44      // Initialize session slots. At startup, all the in-memory session slots
45      // are cleared and marked as not occupied
46      for(i = 0; i < MAX_LOADED_SESSIONS; i++)

```

```

47     s_sessions[i].occupied = FALSE;    // session slot is not occupied
48
49     // The free session slots the number of maximum allowed loaded sessions
50     s_freeSessionSlots = MAX_LOADED_SESSIONS;
51
52     // Initialize context ID data. On a ST_SAVE or hibernate sequence, it will
53     // scan the saved array of session context counts, and clear any entry that
54     // references a session that was in memory during the state save since that
55     // memory was not preserved over the ST_SAVE.
56     if(type == SU_RESUME || type == SU_RESTART)
57     {
58         // On ST_SAVE we preserve the contexts that were saved but not the ones
59         // in memory
60         for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
61         {
62             // If the array value is unused or references a loaded session then
63             // that loaded session context is lost and the array entry is
64             // reclaimed.
65             if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
66                 gr.contextArray[i] = 0;
67         }
68         // Find the oldest session in context ID data and set it in
69         // s_oldestSavedSession
70         ContextIdSetOldest();
71     }
72     else
73     {
74         // For STARTUP_CLEAR, clear out the contextArray
75         for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
76             gr.contextArray[i] = 0;
77
78         // reset the context counter
79         gr.contextCounter = MAX_LOADED_SESSIONS + 1;
80
81         // Initialize oldest saved session
82         s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
83     }
84     return;
85 }

```

## 8.8.5 Access Functions

### 8.8.5.1 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.

Return Value	Meaning
TRUE	if session is loaded
FALSE	if it is not loaded

```

86     BOOL
87     SessionIsLoaded(
88         TPM_HANDLE    handle    // IN: session handle
89     )
90     {
91         pAssert( HandleGetType(handle) == TPM_HT_POLICY_SESSION
92                 || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
93     }

```

```

94     handle = handle & HR_HANDLE_MASK;
95
96     // if out of range of possible active session, or not assigned to a loaded
97     // session return false
98     if(   handle >= MAX_ACTIVE_SESSIONS
99         || gr.contextArray[handle] == 0
100        || gr.contextArray[handle] > MAX_LOADED_SESSIONS
101        )
102         return FALSE;
103
104     return TRUE;
105 }

```

### 8.8.5.2 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.

Return Value	Meaning
TRUE	if session is saved
FALSE	if it is not saved

```

106 BOOL
107 SessionIsSaved(
108     TPM_HANDLE     handle     // IN: session handle
109 )
110 {
111     pAssert(   HandleGetType(handle) == TPM_HT_POLICY_SESSION
112             || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
113
114     handle = handle & HR_HANDLE_MASK;
115     // if out of range of possible active session, or not assigned, or
116     // assigned to a loaded session, return false
117     if(   handle >= MAX_ACTIVE_SESSIONS
118         || gr.contextArray[handle] == 0
119         || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
120        )
121         return FALSE;
122
123     return TRUE;
124 }

```

### 8.8.5.3 SessionPCRValuesCurrent()

This function is used to check if PCR values have been updated since the last time they were checked in a policy session.

This function requires the session is loaded.

Return Value	Meaning
TRUE	if PCR value is current
FALSE	if PCR value is not current

```

125 BOOL

```



```

126 SessionPCRValueIsCurrent(
127     TPMI_SH_POLICY    handle           // IN: session handle
128 )
129 {
130     SESSION            *session;
131
132     pAssert(SessionIsLoaded(handle));
133
134     session = SessionGet(handle);
135     if( session->pcrCounter != 0
136         && session->pcrCounter != gr.pcrCounter
137     )
138         return FALSE;
139     else
140         return TRUE;
141 }

```

#### 8.8.5.4 SessionGet()

This function returns a pointer to the session object associated with a session handle.

The function requires that the session is loaded.

```

142 SESSION *
143 SessionGet(
144     TPM_HANDLE    handle           // IN: session handle
145 )
146 {
147     CONTEXT_SLOT    sessionIndex;
148
149     pAssert( HandleGetType(handle) == TPM_HT_POLICY_SESSION
150             || HandleGetType(handle) == TPM_HT_HMAC_SESSION
151         );
152
153     pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);
154
155     // get the contents of the session array. Because session is loaded, we
156     // should always get a valid sessionIndex
157     sessionIndex = gr.contextArray[handle & HR_HANDLE_MASK] - 1;
158
159     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
160
161     return &s_sessions[sessionIndex].session;
162 }

```

### 8.8.6 Utility Functions

#### 8.8.6.1 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.

return type	TPM_RC
TPM_RC_SUCCESS	context ID was assigned
TPM_RC_CONTEXT_GAP	can't assign a new <i>contextID</i> until the oldest saved session context is recycled
TPM_RC_SESSION_HANDLE	there is no slot available in the context array for tracking of this session context

```

163 static TPM_RC
164 ContextIdSessionCreate (
165     TPM_HANDLE     *handle,           // OUT: receives the assigned handle.
166                                     // This will be an index that must be
167                                     // adjusted by the caller according
168                                     // to the type of the session created
169     UINT32         sessionIndex      // IN: The session context array entry
170                                     // that will be occupied by the created
171                                     // session
172 )
173 {
174
175     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
176
177     // check to see if creating the context is safe
178     // Is this going to be an assignment for the last session context
179     // array entry? If so, then there will be no room to recycle the
180     // oldest context if needed. If the gap is not at maximum, then
181     // it will be possible to save a context if it becomes necessary.
182     if( s_oldestSavedSession < MAX_ACTIVE_SESSIONS
183         && s_freeSessionSlots == 1)
184     {
185         // See if the gap is at maximum
186         if( (CONTEXT_SLOT)gr.contextCounter
187            == gr.contextArray[s_oldestSavedSession])
188
189             // Note: if this is being used on a TPM.combined, this return
190             // code should be transformed to an appropriate 1.2 error
191             // code for this case.
192             return TPM_RC_CONTEXT_GAP;
193     }
194
195     // Find an unoccupied entry in the contextArray
196     for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
197     {
198         if(gr.contextArray[*handle] == 0)
199         {
200             // indicate that the session associated with this handle
201             // references a loaded session
202             gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex+1);
203             return TPM_RC_SUCCESS;
204         }
205     }
206     return TPM_RC_SESSION_HANDLES;
207 }

```

### 8.8.6.2 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.

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

```

208 TPM_RC
209 SessionCreate(
210     TPM_SE             sessionType,      // IN: the session type
211     TPMI_ALG_HASH     authHash,        // IN: the hash algorithm
212     TPM2B_NONCE       *nonceCaller,    // IN: initial nonceCaller
213     TPMT_SYM_DEF      *symmetric,      // IN: the symmetric algorithm
214     TPMI_DH_ENTITY    bind,           // IN: the bind object
215     TPM2B_DATA        *seed,          // IN: seed data
216     TPM_HANDLE        *sessionHandle   // OUT: the session handle
217 )
218 {
219     TPM_RC             result = TPM_RC_SUCCESS;
220     CONTEXT_SLOT      slotIndex;
221     SESSION           *session = NULL;
222
223     pAssert( sessionType == TPM_SE_HMAC
224             || sessionType == TPM_SE_POLICY
225             || sessionType == TPM_SE_TRIAL);
226
227     // If there are no open spots in the session array, then no point in searching
228     if(s_freeSessionSlots == 0)
229         return TPM_RC_SESSION_MEMORY;
230
231     // Find a space for loading a session
232     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
233     {
234         // Is this available?
235         if(s_sessions[slotIndex].occupied == FALSE)
236         {
237             session = &s_sessions[slotIndex].session;
238             break;
239         }
240     }
241     // if no spot found, then this is an internal error
242     pAssert (slotIndex < MAX_LOADED_SESSIONS);
243
244     // Call context ID function to get a handle. TPM_RC_SESSION_HANDLE may be
245     // returned from ContextIdHandleAssign()
246     result = ContextIdSessionCreate(sessionHandle, slotIndex);
247     if(result != TPM_RC_SUCCESS)
248         return result;
249
250     //*** Only return from this point on is TPM_RC_SUCCESS
251
252     // Can now indicate that the session array entry is occupied.
253     s_freeSessionSlots--;
254     s_sessions[slotIndex].occupied = TRUE;
255
256     // Initialize the session data
257     MemorySet(session, 0, sizeof(SESSION));
258
259     // Initialize internal session data
260     session->authHashAlg = authHash;
261     // Initialize session type
262     if(sessionType == TPM_SE_HMAC)
263     {
264         *sessionHandle += HMAC_SESSION_FIRST;
265     }

```

```

266     }
267     else
268     {
269         *sessionHandle += POLICY_SESSION_FIRST;
270
271         // For TPM_SE_POLICY or TPM_SE_TRIAL
272         session->attributes.isPolicy = SET;
273         if(sessionType == TPM_SE_TRIAL)
274             session->attributes.isTrialPolicy = SET;
275
276         // Initialize policy session data
277         SessionInitPolicyData(session);
278     }
279     // Create initial session nonce
280     session->nonceTPM.t.size = nonceCaller->t.size;
281     CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
282
283     // Set up session parameter encryption algorithm
284     session->symmetric = *symmetric;
285
286     // If there is a bind object or a session secret, then need to compute
287     // a sessionKey.
288     if(bind != TPM_RH_NULL || seed->t.size != 0)
289     {
290         // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
291         //                     nonceCaller, bits)
292         // The HMAC key for generating the sessionSecret can be the concatenation
293         // of an authorization value and a seed value
294         TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
295         TPM2B_KEY      key;
296
297         UINT16          hashSize;          // The size of the hash used by the
298                                         // session crated by this command
299         TPM2B_AUTH     entityAuth;        // The authValue of the entity
300                                         // associated with HMAC session
301
302         // Get hash size, which is also the length of sessionKey
303         hashSize = CryptGetHashDigestSize(session->authHashAlg);
304
305         // Get authValue of associated entity
306         entityAuth.t.size = EntityGetAuthValue(bind, &entityAuth.t.buffer[0]);
307
308         // Concatenate authValue and seed
309         MemoryCopy2B(&key.b, &entityAuth.b);
310         MemoryConcat2B(&key.b, &seed->b);
311
312         session->sessionKey.t.size = hashSize;
313
314         // Compute the session key
315         KDFa(session->authHashAlg, &key.b, "ATH", &session->nonceTPM.b,
316             &nonceCaller->b, hashSize * 8, session->sessionKey.t.buffer, NULL);
317     }
318
319     // Copy the name of the entity that the HMAC session is bound to
320     // Policy session is not bound to an entity
321     if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
322     {
323         session->attributes.isBound = SET;
324         SessionComputeBoundEntity(bind, &session->u1.boundEntity);
325     }
326     // If there is a bind object and it is subject to DA, then use of this session
327     // is subject to DA regardless of how it is used.
328     session->attributes.isDaBound = (bind != TPM_RH_NULL)
329                                     && (IsDAExempted(bind) == FALSE);
330
331     // If the session is bound, then check to see if it is bound to lockoutAuth

```

```

332     session->attributes.isLockoutBound = (session->attributes.isDaBound == SET)
333                                         && (bind == TPM_RH_LOCKOUT);
334     return TPM_RC_SUCCESS;
335 }
336

```

### 8.8.6.3 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.

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	a <i>contextID</i> could not be assigned.
TPM_RC_TOO_MANY_CONTEXTS	the counter maxed out

```

337 TPM_RC
338 SessionContextSave (
339     TPM_HANDLE      handle,          // IN: session handle
340     CONTEXT_COUNTER *contextID      // OUT: assigned contextID
341 )
342 {
343     UINT32          contextIndex;
344     CONTEXT_SLOT    slotIndex;
345
346     pAssert(SessionIsLoaded(handle));
347
348     // check to see if the gap is already maxed out
349     // Need to have a saved session
350     if( s_oldestSavedSession < MAX_ACTIVE_SESSIONS
351         // if the oldest saved session has the same value as the low bits
352         // of the contextCounter, then the GAP is maxed out.
353         && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
354         return TPM_RC_CONTEXT_GAP;
355
356     // if the caller wants the context counter, set it
357     if(contextID != NULL)
358         *contextID = gr.contextCounter;
359
360     pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);
361
362     contextIndex = handle & HR_HANDLE_MASK;
363
364     // Extract the session slot number referenced by the contextArray
365     // because we are going to overwrite this with the low order
366     // contextID value.
367     slotIndex = gr.contextArray[contextIndex] - 1;
368
369     // Set the contextID for the contextArray
370     gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
371
372     // Increment the counter
373     gr.contextCounter++;
374
375     // In the unlikely event that the 64-bit context counter rolls over...
376     if(gr.contextCounter == 0)
377     {
378         // back it up
379         gr.contextCounter--;

```

```

380     // return an error
381     return TPM_RC_TOO_MANY_CONTEXTS;
382 }
383 // if the low-order bits wrapped, need to advance the value to skip over
384 // the values used to indicate that a session is loaded
385 if(((CONTEXT_SLOT)gr.contextCounter) == 0)
386     gr.contextCounter += MAX_LOADED_SESSIONS + 1;
387
388 // If no other sessions are saved, this is now the oldest.
389 if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
390     s_oldestSavedSession = contextIndex;
391
392 // Mark the session slot as unoccupied
393 s_sessions[slotIndex].occupied = FALSE;
394
395 // and indicate that there is an additional open slot
396 s_freeSessionSlots++;
397
398 return TPM_RC_SUCCESS;
399 }

```

#### 8.8.6.4 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.

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

```

400 TPM_RC
401 SessionContextLoad(
402     SESSION          *session,    // IN: session structure from saved
403     // context
404     TPM_HANDLE      *handle      // IN/OUT: session handle
405 )
406 {
407     UINT32          contextIndex;
408     CONTEXT_SLOT    slotIndex;
409
410     pAssert( HandleGetType(*handle) == TPM_HT_POLICY_SESSION
411             || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
412
413     // Don't bother looking if no openings
414     if(s_freeSessionSlots == 0)
415         return TPM_RC_SESSION_MEMORY;
416
417     // Find a free session slot to load the session
418     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
419         if(s_sessions[slotIndex].occupied == FALSE) break;
420
421     // if no spot found, then this is an internal error
422     pAssert (slotIndex < MAX_LOADED_SESSIONS);
423
424     contextIndex = *handle & HR_HANDLE_MASK;    // extract the index
425
426     // If there is only one slot left, and the gap is at maximum, the only session
427     // context that we can safely load is the oldest one.

```

```

428     if( s_oldestSavedSession < MAX_ACTIVE_SESSIONS
429         && s_freeSessionSlots == 1
430         && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
431         && contextIndex != s_oldestSavedSession
432     )
433         return TPM_RC_CONTEXT_GAP;
434
435     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
436
437     // set the contextArray value to point to the session slot where
438     // the context is loaded
439     gr.contextArray[contextIndex] = slotIndex + 1;
440
441     // if this was the oldest context, find the new oldest
442     if(contextIndex == s_oldestSavedSession)
443         ContextIdSetOldest();
444
445     // Copy session data to session slot
446     s_sessions[slotIndex].session = *session;
447
448     // Set session slot as occupied
449     s_sessions[slotIndex].occupied = TRUE;
450
451     // Reduce the number of open spots
452     s_freeSessionSlots--;
453
454     return TPM_RC_SUCCESS;
455 }

```

#### 8.8.6.5 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.

```

456 void
457 SessionFlush(
458     TPM_HANDLE          handle          // IN: loaded or saved session handle
459 )
460 {
461     CONTEXT_SLOT        slotIndex;
462     UINT32              contextIndex; // Index into contextArray
463
464     pAssert( ( HandleGetType(handle) == TPM_HT_POLICY_SESSION
465              || HandleGetType(handle) == TPM_HT_HMAC_SESSION
466            )
467             && (SessionIsLoaded(handle) || SessionIsSaved(handle))
468            );
469
470     // Flush context ID of this session
471     // Convert handle to an index into the contextArray
472     contextIndex = handle & HR_HANDLE_MASK;
473
474     // Get the current contents of the array
475     slotIndex = gr.contextArray[contextIndex];
476
477     // Mark context array entry as available
478     gr.contextArray[contextIndex] = 0;
479
480     // Is this a saved session being flushed
481     if(slotIndex > MAX_LOADED_SESSIONS)
482     {
483         // Flushing the oldest session?

```

```

484     if(contextIndex == s_oldestSavedSession)
485         // If so, find a new value for oldest.
486         ContextIdSetOldest();
487     }
488     else
489     {
490         // Adjust slot index to point to session array index
491         slotIndex -= 1;
492
493         // Free session array index
494         s_sessions[slotIndex].occupied = FALSE;
495         s_freeSessionSlots++;
496     }
497
498     return;
499 }

```

### 8.8.6.6 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, they will be overlapped by *XORing()* bytes. If XOR is required, the bind value will be full.

```

500 void
501 SessionComputeBoundEntity(
502     TPMI_DH_ENTITY    entityHandle, // IN: handle of entity
503     TPM2B_NAME        *bind         // OUT: binding value
504 )
505 {
506     TPM2B_AUTH        auth;
507     INT16             overlap;
508
509     // Get name
510     bind->t.size = EntityGetName(entityHandle, bind->t.name);
511
512     // // The bound value of a reserved handle is the handle itself
513     // if(bind->t.size == sizeof(TPM_HANDLE)) return;
514
515     // For all the other entities, concatenate the auth value to the name.
516     // Get a local copy of the auth value because some overlapping
517     // may be necessary.
518     auth.t.size = EntityGetAuthValue(entityHandle, auth.t.buffer);
519     pAssert(auth.t.size <= <K>sizeof(TPMU_HA));
520
521     // Figure out if there will be any overlap
522     overlap = bind->t.size + auth.t.size - sizeof(bind->t.name);
523
524     // There is overlap if the combined sizes are greater than will fit
525     if(overlap > 0)
526     {
527         // The overlap area is at the end of the Name
528         BYTE    *result = &bind->t.name[bind->t.size - overlap];
529         int     i;
530
531         // XOR the auth value into the Name for the overlap area
532         for(i = 0; i < overlap; i++)
533             result[i] ^= auth.t.buffer[i];
534     }
535     else
536     {
537         // There is no overlap
538         overlap = 0;
539     }

```



```

540     //copy the remainder of the authData to the end of the name
541     MemoryCopy(&bind->t.name[bind->t.size], &auth.t.buffer[overlap],
542             auth.t.size - overlap);
543
544     // Increase the size of the bind data by the size of the auth - the overlap
545     bind->t.size += auth.t.size-overlap;
546
547     return;
548 }

```

### 8.8.6.7 SessionInitPolicyData()

This function initializes the portions of the session policy data that are not set by the allocation of a session.

```

549 void
550 SessionInitPolicyData(
551     SESSION      *session      // IN: session handle
552 )
553 {
554     // Initialize start time
555     session->startTime = go.clock;
556
557     // Initialize policyDigest.  policyDigest is initialized with a string of 0 of
558     // session algorithm digest size. Since the policy already contains all zeros
559     // it is only necessary to set the size
560     session->u2.policyDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
561     return;
562 }

```

### 8.8.6.8 SessionResetPolicyData()

This function is used to reset the policy data without changing the nonce or the start time of the session.

```

563 void
564 SessionResetPolicyData(
565     SESSION      *session      // IN: the session to reset
566 )
567 {
568     session->commandCode = 0;      // No command
569
570     // No locality selected
571     MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
572
573     // The cpHash size to zero
574     session->u1.cpHash.b.size = 0;
575
576     // Reset the pcrCounter
577     session->pcrCounter = 0;
578
579     // Reset the policy hash
580     MemorySet(&session->u2.policyDigest.t.buffer, 0, session->u2.policyDigest.t.size);
581
582     // Reset the session attributes
583     MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
584
585     // set the policy attribute
586     session->attributes.isPolicy = SET;
587 }

```

### 8.8.6.9 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.

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

```

588  TPMI_YES_NO
589  SessionCapGetLoaded(
590      TPMI_SH_POLICY      handle,          // IN: start handle
591      UINT32              count,          // IN: count of returned handles
592      TPML_HANDLE         *handleList     // OUT: list of handle
593  )
594  {
595      TPMI_YES_NO      more = NO;
596      UINT32           i;
597
598      pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
599
600      // Initialize output handle list
601      handleList->count = 0;
602
603      // The maximum count of handles we may return is MAX_CAP_HANDLES
604      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
605
606      // Iterate session context ID slots to get loaded session handles
607      for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
608      {
609          // If session is active
610          if(gr.contextArray[i] != 0)
611          {
612              // If session is loaded
613              if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
614              {
615                  if(handleList->count < count)
616                  {
617                      SESSION      *session;
618
619                      // If we have not filled up the return list, add this
620                      // session handle to it
621                      // assume that this is going to be an HMAC session
622                      handle = i + HMAC_SESSION_FIRST;
623                      session = SessionGet(handle);
624                      if(session->attributes.isPolicy)
625                          handle = i + POLICY_SESSION_FIRST;
626                      handleList->handle[handleList->count] = handle;
627                      handleList->count++;
628                  }
629                  else
630                  {
631                      // If the return list is full but we still have loaded object
632                      // available, report this and stop iterating
633                      more = YES;
634                      break;
635                  }
636              }
637          }
638      }
639
640      return more;

```

```
641
642 }
```

### 8.8.6.10 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

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

```
643 TPMI_YES_NO
644 SessionCapGetSaved(
645     TPMI_SH_HMAC          handle,          // IN: start handle
646     UINT32                count,          // IN: count of returned handles
647     TPML_HANDLE          *handleList     // OUT: list of handle
648 )
649 {
650     TPMI_YES_NO    more = NO;
651     UINT32        i;
652
653     pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);
654
655     // Initialize output handle list
656     handleList->count = 0;
657
658     // The maximum count of handles we may return is MAX_CAP_HANDLES
659     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
660
661     // Iterate session context ID slots to get loaded session handles
662     for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
663     {
664         // If session is active
665         if(gr.contextArray[i] != 0)
666         {
667             // If session is saved
668             if (gr.contextArray[i] > MAX_LOADED_SESSIONS)
669             {
670                 if(handleList->count < count)
671                 {
672                     // If we have not filled up the return list, add this
673                     // session handle to it
674                     handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
675                     handleList->count++;
676                 }
677             }
678             else
679             {
680                 // If the return list is full but we still have loaded object
681                 // available, report this and stop iterating
682                 more = YES;
683                 break;
684             }
685         }
686     }
687
688     return more;
689 }
690 }
```

**8.8.6.11 SessionCapGetLoadedNumber()**

This function return the number of authorization sessions currently loaded into TPM RAM.

```

691  UINT32
692  SessionCapGetLoadedNumber(void)
693  {
694      return MAX_LOADED_SESSIONS - s_freeSessionSlots;
695  }
```

**8.8.6.12 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 may just be an estimate. The only requirement for the estimate is, if it is one or more, then at least one session must be loadable.

```

696  UINT32
697  SessionCapGetLoadedAvail(void)
698  {
699      return s_freeSessionSlots;
700  }
```

**8.8.6.13 SessionCapGetActiveNumber()**

This function returns the number of active authorization sessions currently being tracked by the TPM.

```

701  UINT32
702  SessionCapGetActiveNumber(void)
703  {
704      UINT32          i;
705      UINT32          num = 0;
706
707      // Iterate the context array to find the number of non-zero slots
708      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
709      {
710          if(gr.contextArray[i] != 0) num++;
711      }
712
713      return num;
714  }
```

**8.8.6.14 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.

```

715  UINT32
716  SessionCapGetActiveAvail(void)
717  {
718      UINT32          i;
719      UINT32          num = 0;
720
721      // Iterate the context array to find the number of zero slots
722      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
723      {
724          if(gr.contextArray[i] == 0) num++;
725      }
```

```

726
727     return num;
728 }

```

## 8.9 Time.c

### 8.9.1 Introduction

This file contains the functions relating to the TPM's time functions including the interface to the implementation-specific time functions.

### 8.9.2 Includes

```

1 #include "InternalRoutines.h"
2 #include "Platform.h"

```

### 8.9.3 Functions

#### 8.9.3.1 TimePowerOn()

This function initialize time info at `_TPM_Init()`.

```

3 void
4 TimePowerOn(void)
5 {
6     TPM_SU          orderlyShutDown;
7
8     // Read time info from NV memory
9     NvReadReserved(NV_CLOCK, &go.clock);
10
11    // Read orderly shut down state
12    NvReadReserved(NV_ORDERLY, &orderlyShutDown);
13
14    // If the previous cycle is orderly shut down, the value of the safe bit
15    // the same as previously saved. Otherwise, it is not safe.
16    if(orderlyShutDown == SHUTDOWN_NONE)
17        go.clockSafe= NO;
18    else
19        go.clockSafe = YES;
20
21    // Clear time
22    g_time = 0;
23
24    return;
25 }

```

#### 8.9.3.2 TimeStartup()

This function updates the *resetCount* and *restartCount* components of `TPMS_CLOCK_INFO` structure at `TPM2_Startup()`.

```

26 void
27 TimeStartup(
28     STARTUP_TYPE      type          // IN: start up type
29 )
30 {
31     if(type == SU_RESUME)
32     {

```

```

33     // Resume sequence
34     gr.restartCount++;
35 }
36 else
37 {
38     if(type == SU_RESTART)
39     {
40         // Hibernate sequence
41         gr.clearCount++;
42         gr.restartCount++;
43     }
44     else
45     {
46         // Reset sequence
47         // Increase resetCount
48         gp.resetCount++;
49
50         // Write resetCount to NV
51         NvWriteReserved(NV_RESET_COUNT, &gp.resetCount);
52         gp.totalResetCount++;
53
54         // We do not expect the total reset counter overflow during the life
55         // time of TPM.  if it ever happens, TPM will be put to failure mode
56         // and there is no way to recover it.
57         // The reason that there is no recovery is that we don't increment
58         // the NV totalResetCount when incrementing would make it 0.  When the
59         // TPM starts up again, the old value of totalResetCount will be read
60         // and we will get right back to here with the increment failing.
61         if(gp.totalResetCount == 0)
62             FAIL(FATAL_ERROR_INTERNAL);
63
64
65         // Write total reset counter to NV
66         NvWriteReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
67
68         // Reset restartCount
69         gr.restartCount = 0;
70     }
71 }
72
73 return;
74 }

```

### 8.9.3.3 TimeUpdateToCurrent()

This function updates the *Time* and *Clock* in the global TPMS\_TIME\_INFO structure.

In this implementation, *Time* and *Clock* are updated at the beginning of each command and the values are unchanged for the duration of the command.

Because *Clock* updates may require a write to NV memory, *Time* and *Clock* are not allowed to advance if NV is not available. When clock is not advancing, any function that uses *Clock* will fail and return TPM\_RC\_NV\_UNAVAILABLE or TPM\_RC\_NV\_RATE.

This 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.

```

75 void
76 TimeUpdateToCurrent(void)
77 {
78     UINT64      oldClock;
79     UINT64      elapsed;
80     #define CLOCK_UPDATE_MASK ((1ULL << NV_CLOCK_UPDATE_INTERVAL) - 1)
81

```

```

82     // Can't update time during the dark interval or when rate limiting.
83     if(NvIsAvailable() != TPM_RC_SUCCESS)
84         return;
85
86     // Save the old clock value
87     oldClock = go.clock;
88
89     // Update the time info to current
90     elapsed = _plat__ClockTimeElapsed();
91     go.clock += elapsed;
92     g_time += elapsed;
93
94     // Check to see if the update has caused a need for an nvClock update
95     // CLOCK_UPDATE_MASK is measured by second, while the value in go.clock is
96     // recorded by millisecond. Align the clock value to second before the bit
97     // operations
98     if( ((go.clock/1000) | CLOCK_UPDATE_MASK)
99         > ((oldClock/1000) | CLOCK_UPDATE_MASK)
100     {
101         NvWriteReserved(NV_CLOCK, &go.clock);
102
103         // Now the time state is updated
104         go.clockSafe = YES;
105     }
106
107     // Call self healing logic for dictionary attack parameters
108     DASelfHeal();
109
110     return;
111 }

```

#### 8.9.3.4 TimeSetAdjustRate()

This function is used to perform rate adjustment on *Time* and *Clock*.

```

112 void
113 TimeSetAdjustRate(
114     TPM_CLOCK_ADJUST          adjust          // IN: adjust constant
115 )
116 {
117     switch(adjust)
118     {
119         case TPM_CLOCK_COARSE_SLOWER:
120             _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
121             break;
122         case TPM_CLOCK_COARSE_FASTER:
123             _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
124             break;
125         case TPM_CLOCK_MEDIUM_SLOWER:
126             _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
127             break;
128         case TPM_CLOCK_MEDIUM_FASTER:
129             _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
130             break;
131         case TPM_CLOCK_FINE_SLOWER:
132             _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
133             break;
134         case TPM_CLOCK_FINE_FASTER:
135             _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
136             break;
137         case TPM_CLOCK_NO_CHANGE:
138             break;
139         default:
140             pAssert(FALSE);

```

```

141         break;
142     }
143
144     return;
145 }

```

### 8.9.3.5 TimeGetRange()

This function is used to access TPMS\_TIME\_INFO. The TPMS\_TIME\_INFO structure is treated as an array of bytes, and a byte offset and length determine what bytes are returned.

Error Returns	Meaning
TPM_RC_RANGE	invalid data range

```

146 TPM_RC
147 TimeGetRange(
148     UINT16    offset,           // IN: offset in TPMS_TIME_INFO
149     UINT16    size,           // IN: size of data
150     BYTE      *dataBuffer     // OUT: result buffer
151 )
152 {
153     TPMS_TIME_INFO    timeInfo;
154     UINT16            infoSize;
155     BYTE              infoData[sizeof(TPMS_TIME_INFO)];
156     BYTE              *buffer;
157
158     // Fill TPMS_TIME_INFO structure
159     timeInfo.time = g_time;
160     TimeFillInfo(&timeInfo.clockInfo);
161
162     // Marshal TPMS_TIME_INFO to canonical form
163     buffer = infoData;
164     infoSize = TPMS_TIME_INFO_Marshal(&timeInfo, &buffer, NULL);
165
166     // Check if the input range is valid
167     if(offset + size > infoSize) return TPM_RC_RANGE;
168
169     // Copy info data to output buffer
170     MemoryCopy(dataBuffer, infoData + offset, size);
171
172     return TPM_RC_SUCCESS;
173 }

```

### 8.9.3.6 TimeFillInfo

This function gathers information to fill in a TPMS\_CLOCK\_INFO structure.

```

174 void
175 TimeFillInfo(
176     TPMS_CLOCK_INFO    *clockInfo
177 )
178 {
179     clockInfo->clock = go.clock;
180     clockInfo->resetCount = gp.resetCount;
181     clockInfo->restartCount = gr.restartCount;
182
183     // If NV is not available, clock stopped advancing and the value reported is
184     // not "safe".
185     if(NvIsAvailable() == TPM_RC_SUCCESS)
186         clockInfo->safe = go.clockSafe;
187     else

```



```
188         clockInfo->safe = NO;
189
190     return;
191 }
```

## 9 Support

### 9.1 AlgorithmCap.c

#### 9.1.1 Description

This file contains the algorithm property definitions for the algorithms and the code for the TPM2\_GetCapability() to return the algorithm properties.

#### 9.1.2 Includes and Defines

```

1  #include "InternalRoutines.h"
2  typedef struct
3  {
4      TPM_ALG_ID          algID;
5      TPMA_ALGORITHM     attributes;
6  } ALGORITHM;
7  static const ALGORITHM  s_algorithms[] =
8  {
9      #ifndef TPM_ALG_RSA
10     {TPM_ALG_RSA,          {1, 0, 0, 1, 0, 0, 0, 0, 0}},
11     #endif
12     #ifndef TPM_ALG_DES
13     {TPM_ALG_DES,         {0, 1, 0, 0, 0, 0, 0, 0, 0}},
14     #endif
15     #ifndef TPM_ALG_3DES
16     {TPM_ALG_3DES,        {0, 1, 0, 0, 0, 0, 0, 0, 0}},
17     #endif
18     #ifndef TPM_ALG_SHA1
19     {TPM_ALG_SHA1,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
20     #endif
21     #ifndef TPM_ALG_HMAC
22     {TPM_ALG_HMAC,        {0, 0, 1, 0, 0, 1, 0, 0, 0}},
23     #endif
24     #ifndef TPM_ALG_AES
25     {TPM_ALG_AES,         {0, 1, 0, 0, 0, 0, 0, 0, 0}},
26     #endif
27     #ifndef TPM_ALG_MGF1
28     {TPM_ALG_MGF1,        {0, 0, 1, 0, 0, 0, 0, 1, 0}},
29     #endif
30
31     {TPM_ALG_KEYEDHASH,   {0, 0, 1, 1, 0, 1, 1, 0, 0}},
32
33     #ifndef TPM_ALG_XOR
34     {TPM_ALG_XOR,         {0, 1, 1, 0, 0, 0, 0, 0, 0}},
35     #endif
36
37     #ifndef TPM_ALG_SHA256
38     {TPM_ALG_SHA256,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
39     #endif
40     #ifndef TPM_ALG_SHA384
41     {TPM_ALG_SHA384,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
42     #endif
43     #ifndef TPM_ALG_SHA512
44     {TPM_ALG_SHA512,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
45     #endif
46     #ifndef TPM_ALG_WHIRLPOOL512
47     {TPM_ALG_WHIRLPOOL512, {0, 0, 1, 0, 0, 0, 0, 0, 0}},
48     #endif
49     #ifndef TPM_ALG_SM3_256
50     {TPM_ALG_SM3_256,     {0, 0, 1, 0, 0, 0, 0, 0, 0}},
51     #endif

```

```

52 #ifdef TPM_ALG_SM4
53     {TPM_ALG_SM4,          {0, 1, 0, 0, 0, 0, 0, 0, 0}},
54 #endif
55 #ifdef TPM_ALG_RSASSA
56     {TPM_ALG_RSASSA,     {1, 0, 0, 0, 0, 1, 0, 0, 0}},
57 #endif
58 #ifdef TPM_ALG_RSAES
59     {TPM_ALG_RSAES,      {1, 0, 0, 0, 0, 0, 1, 0, 0}},
60 #endif
61 #ifdef TPM_ALG_RSAPSS
62     {TPM_ALG_RSAPSS,     {1, 0, 0, 0, 0, 1, 0, 0, 0}},
63 #endif
64 #ifdef TPM_ALG_OAEP
65     {TPM_ALG_OAEP,       {1, 0, 0, 0, 0, 0, 1, 0, 0}},
66 #endif
67 #ifdef TPM_ALG_ECDSA
68     {TPM_ALG_ECDSA,      {1, 0, 0, 0, 0, 1, 0, 1, 0}},
69 #endif
70 #ifdef TPM_ALG_ECDH
71     {TPM_ALG_ECDH,       {1, 0, 0, 0, 0, 0, 0, 1, 0}},
72 #endif
73 #ifdef TPM_ALG_ECDA
74     {TPM_ALG_ECDA,       {1, 0, 0, 0, 0, 1, 0, 0, 0}},
75 #endif
76 #ifdef TPM_ALG_ECSCHNORR
77     {TPM_ALG_ECSCHNORR, {1, 0, 0, 0, 0, 1, 0, 0, 0}},
78 #endif
79 #ifdef TPM_ALG_KDF1_SP800_56a
80     {TPM_ALG_KDF1_SP800_56a, {0, 0, 1, 0, 0, 0, 0, 1, 0}},
81 #endif
82 #ifdef TPM_ALG_KDF2
83     {TPM_ALG_KDF2,       {0, 0, 1, 0, 0, 0, 0, 1, 0}},
84 #endif
85 #ifdef TPM_ALG_KDF1_SP800_108
86     {TPM_ALG_KDF1_SP800_108, {0, 0, 1, 0, 0, 0, 0, 1, 0}},
87 #endif
88 #ifdef TPM_ALG_ECC
89     {TPM_ALG_ECC,        {1, 0, 0, 1, 0, 0, 0, 0, 0}},
90 #endif
91     {TPM_ALG_SYMCIPHER,   {0, 0, 0, 1, 0, 0, 0, 0, 0}},
92 #ifdef TPM_ALG_CTR
93     {TPM_ALG_CTR,        {0, 1, 0, 0, 0, 0, 1, 0, 0}},
94 #endif
95 #ifdef TPM_ALG_OFB
96     {TPM_ALG_OFB,        {0, 1, 0, 0, 0, 0, 1, 0, 0}},
97 #endif
98 #ifdef TPM_ALG_CBC
99     {TPM_ALG_CBC,        {0, 1, 0, 0, 0, 0, 1, 0, 0}},
100 #endif
101 #ifdef TPM_ALG_CFB
102     {TPM_ALG_CFB,        {0, 1, 0, 0, 0, 0, 1, 0, 0}},
103 #endif
104 #ifdef TPM_ALG_ECB
105     {TPM_ALG_ECB,        {0, 1, 0, 0, 0, 0, 1, 0, 0}},
106 #endif
107 };
108

```

### 9.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2\_GetCapability() to return a list of the implemented algorithms.

Return Value	Meaning
YES	more algorithms to report
NO	no more algorithms to report

```

110  TPMI_YES_NO
111  AlgorithmCapGetImplemented(
112      TPM_ALG_ID          algID,      // IN: the starting algorithm ID
113      UINT32              count,      // IN: count of returned algorithms
114      TPML_ALG_PROPERTY  *algList    // OUT: algorithm list
115  )
116  {
117      TPMI_YES_NO    more = NO;
118      UINT32         i;
119      UINT32         algNum;
120
121      // initialize output algorithm list
122      algList->count = 0;
123
124      // The maximum count of algorithms we may return is MAX_CAP_ALGS.
125      if(count > MAX_CAP_ALGS) count = MAX_CAP_ALGS;
126
127      // Compute how many algorithms are defined in s_algorithms array.
128      algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
129
130      // Scan the implemented algorithm list to see if there is a match to 'algID'.
131      for(i = 0; i < algNum; i++)
132      {
133          // If algID is less than the starting algorithm ID, skip it
134          if(s_algorithms[i].algID < algID)
135              continue;
136          if(algList->count < count)
137          {
138              // If we have not filled up the return list, add more algorithms
139              // to it
140              algList->algProperties[algList->count].alg = s_algorithms[i].algID;
141              algList->algProperties[algList->count].algProperties =
142                  s_algorithms[i].attributes;
143              algList->count++;
144          }
145          else
146          {
147              // If the return list is full but we still have algorithms
148              // available, report this and stop scanning.
149              more = YES;
150              break;
151          }
152      }
153  }
154
155  return more;
156
157  }

```

## 9.2 Bits.c

### 9.2.1 Introduction

This file contains bit manipulation routines. They operate on bit arrays.

The 0th bit in the array is the right-most bit in the 0th octet in the array.

NOTE: If *pAssert()* is defined, the functions will assert if the indicated bit number is outside of the range of *bArray*. How the assert is handled is implementation dependent.

## 9.2.2 Includes

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

## 9.2.3 BitIsSet()

This function is used to check the setting of a bit in an array of bits.

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 }
```

## 9.2.4 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'
17 )
18 {
19     pAssert(arraySize > bitNum/8);
20     bArray[bitNum >> 3] |= (1 << (bitNum & 7));
21 }
```

## 9.2.5 BitClear()

This function will clear the indicated bit in *bArray*.

```
22 void
23 BitClear(
24     unsigned int      bitNum,      // IN: number of the bit in 'bArray'.
25     BYTE             *bArray,     // IN: array containing the bits
26     unsigned int      arraySize   // IN: size in bytes of 'bArray'
27 )
28 {
29     pAssert(arraySize > bitNum/8);
30     bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
31 }
```

### 9.3 CommandCodeAttributes\_fp.h

```
[[CommandCodeAttributes_fp_h]]
```

### 9.4 Commands.c

#### 9.4.1 Description

This file contains the function used by TPM2\_GetCapability() to build the list of command code attributes.

#### 9.4.2 Includes

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

#### 9.4.3 CommandCapGetCCList()

This function returns a list of implemented commands and command attributes starting from the command in *commandCode*.

Return Value	Meaning
YES	more command attributes are available
NO	no more command attributes are available

```
2 TPMI_YES_NO
3 CommandCapGetCCList(
4     TPM_CC      commandCode,      // IN: start command code
5     UINT32      count,           // IN: maximum count for number of
6     // entries in 'commandList'
7     TPML_CCA   *commandList      // OUT: list of TPMA_CC
8 )
9 {
10     TPMI_YES_NO    more = NO;
11     UINT32         i;
12
13     // initialize output handle list count
14     commandList->count = 0;
15
16     // The maximum count of commands that may be return is MAX_CAP_CC.
17     if(count > MAX_CAP_CC) count = MAX_CAP_CC;
18
19     // If the command code is smaller than TPM_CC_FIRST, start from TPM_CC_FIRST
20     if(commandCode < TPM_CC_FIRST) commandCode = TPM_CC_FIRST;
21
22     // Collect command attributes
23     for(i = commandCode; i <= TPM_CC_LAST; i++)
24     {
25         if(CommandIsImplemented(i))
26         {
27             if(commandList->count < count)
28             {
29                 // If the list is not full, add the attributes for this command.
30                 commandList->commandAttributes[commandList->count]
31                     = CommandGetAttribute(i);
32                 commandList->count++;
33             }
34             else
35             {
36                 // If the list is full but there are more commands to report,
37                 // indicate this and return.
```

```

38         more = YES;
39         break;
40     }
41 }
42 }
43
44 return more;
45
46 }

```

## 9.5 DRTM.c

### 9.5.1 Description

This file contains functions that simulate the DRTM events.

### 9.5.2 Includes

```

1 #include "InternalRoutines.h"

```

#### 9.5.2.1 Signal\_Hash\_Start()

This function interfaces between the platform code and `_TPM_Hash_Start()`.

```

2 void Signal_Hash_Start(void)
3 {
4     _TPM_Hash_Start();
5     return;
6 }

```

#### 9.5.2.2 Signal\_Hash\_Data()

This function interfaces between the platform code and `_TPM_Hash_Data()`.

```

7 void Signal_Hash_Data(
8     unsigned int    size,
9     unsigned char   *buffer
10 )
11 {
12     _TPM_Hash_Data(size, buffer);
13     return;
14 }

```

#### 9.5.2.3 Signal\_Hash\_End()

This function interfaces between the platform code and `_TPM_Hash_End()`.

```

15 void Signal_Hash_End(void)
16 {
17     _TPM_Hash_End();
18     return;
19 }

```

## 9.6 Entity.c

### 9.6.1 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.

### 9.6.2 Includes

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

### 9.6.3 Functions

#### 9.6.3.1 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.

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

```
2 TPM_RC
3 EntityGetLoadStatus(
4     TPM_HANDLE *handle // IN/OUT: handle of the entity
5 )
6 {
7     switch(HandleGetType(*handle))
8     {
9         // For handles associated with hierarchies, the entity is present
10        // only if the associated enable is SET.
11        case TPM_HT_PERMANENT:
12            switch(*handle)
13            {
14                case TPM_RH_OWNER:
15                    if(!gc.shEnable)
16                        return TPM_RC_HIERARCHY;
17                    else
18                        return TPM_RC_SUCCESS;
19                    break;
20                case TPM_RH_ENDORSEMENT:
21                    if(!gc.ehEnable)
22                        return TPM_RC_HIERARCHY;
23                    else
24                        return TPM_RC_SUCCESS;
25                    break;
26                case TPM_RH_PLATFORM:
27                    if(!g_phEnable)
28                        return TPM_RC_HIERARCHY;
29                    else
30                        return TPM_RC_SUCCESS;
31                    break;
32                // null handle, PW session handle and lockout
33                // handle are always available
```



```

34         case TPM_RH_NULL:
35         case TPM_RS_PW:
36         case TPM_RH_LOCKOUT:
37             return TPM_RC_SUCCESS;
38             break;
39         default:
40             // should never see any other permanent handle here
41             pAssert(FALSE);
42             return TPM_RC_HANDLE;
43             break;
44     }
45     break;
46 case TPM_HT_TRANSIENT:
47     // For a transient object, check if the handle is associated
48     // with a loaded object.
49     if(ObjectIsPresent(*handle))
50         return TPM_RC_SUCCESS;
51     else
52         return TPM_RC_REFERENCE_H0;
53     break;
54 case TPM_HT_PERSISTENT:
55     // Persistent object
56     // Copy the persistent object to RAM and replace the handle with the
57     // handle of the assigned slot. A TPM_RC_OBJECT_MEMORY,
58     // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned at
59     // this point
60     return ObjectLoadEvict(handle);
61     break;
62 case TPM_HT_HMAC_SESSION:
63     // For an HMAC session, see if the session is loaded
64     // and if the session in the session slot is actually
65     // an HMAC session.
66     if(SessionIsLoaded(*handle))
67     {
68         SESSION          *session;
69         session = SessionGet(*handle);
70         // Check if the session is a HMAC session
71         if(session->attributes.isPolicy == CLEAR)
72             return TPM_RC_SUCCESS;
73         else
74             return TPM_RC_HANDLE;
75     }
76     return TPM_RC_REFERENCE_H0;
77     break;
78 case TPM_HT_POLICY_SESSION:
79     // For a policy session, see if the session is loaded
80     // and if the session in the session slot is actually
81     // a policy session.
82     if(SessionIsLoaded(*handle))
83     {
84         SESSION          *session;
85         session = SessionGet(*handle);
86         // Check if the session is a policy session
87         if(session->attributes.isPolicy == SET)
88             return TPM_RC_SUCCESS;
89         else
90             return TPM_RC_HANDLE;
91     }
92     return TPM_RC_REFERENCE_H0;
93     break;
94 case TPM_HT_NV_INDEX:
95     // For an NV Index, use the platform-specific routine
96     // to search the IN Index space.
97     return NvIndexIsAccessible(*handle);
98     break;
99 case TPM_HT_PCR:

```

```

100         // Any PCR handle that is unmarshaled successfully referenced
101         // a PCR that is defined.
102         return TPM_RC_SUCCESS;
103         break;
104     default:
105         // Any other handle type is a defect in the unmarshaling code.
106         pAssert(FALSE);
107         return TPM_RC_HANDLE;
108         break;
109     }
110 }

```

### 9.6.3.2 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     BYTE              *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                 default:
145                     // If any other permanent handle is present it is
146                     // a code defect.
147                     pAssert(FALSE);
148                     break;
149             }
150             break;
151         case TPM_HT_TRANSIENT:

```

```

152     // authValue for an object
153     // A persistent object would have been copied into RAM
154     // and would have an transient object handle here.
155     {
156         OBJECT        *object;
157         object = ObjectGet(handle);
158         // special handling if this is a sequence object
159         if(ObjectIsSequence(object))
160         {
161             authValue = ((HASH_OBJECT *)object)->auth;
162         }
163         else
164         {
165             // Auth value is available only when the private portion of
166             // the object is loaded. The check should be made before
167             // this function is called
168             pAssert(object->attributes.publicOnly == CLEAR);
169             authValue = object->sensitive.authValue;
170         }
171     }
172     break;
173 case TPM_HT_NV_INDEX:
174     // authValue for an NV index
175     {
176         NV_INDEX        nvIndex;
177         NvGetIndexInfo(handle, &nvIndex);
178         authValue = nvIndex.authValue;
179     }
180     break;
181 case TPM_HT_PCR:
182     // authValue for PCR
183     PCRGetAuthValue(handle, &authValue);
184     break;
185 default:
186     // If any other handle type is present here, then there is a defect
187     // in the unmarshaling code.
188     pAssert(FALSE);
189     break;
190 }
191
192 // Copy the authValue
193 pAssert(authValue.t.size <= <K>sizeof(authValue.t.buffer));
194 MemoryCopy(auth, authValue.t.buffer, authValue.t.size);
195
196 return authValue.t.size;
197 }

```

### 9.6.3.3 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.

```

198 TPMI_ALG_HASH
199 EntityGetAuthPolicy(
200     TPMI_DH_ENTITY handle,           // IN: handle of entity
201     TPM2B_DIGEST *authPolicy        // OUT: authPolicy of the entity
202 )
203 {

```

```

204     TPMI_ALG_HASH      hashAlg = TPM_ALG_NULL;
205
206     switch(HandleGetType(handle))
207     {
208         case TPM_HT_PERMANENT:
209             switch(handle)
210             {
211                 case TPM_RH_OWNER:
212                     // ownerPolicy for TPM_RH_OWNER
213                     *authPolicy = gp.ownerPolicy;
214                     hashAlg = gp.ownerAlg;
215                     break;
216                 case TPM_RH_ENDORSEMENT:
217                     // endorsementPolicy for TPM_RH_ENDORSEMENT
218                     *authPolicy = gp.endorsementPolicy;
219                     hashAlg = gp.endorsementAlg;
220                     break;
221                 case TPM_RH_PLATFORM:
222                     // platformPolicy for TPM_RH_PLATFORM
223                     *authPolicy = gc.platformPolicy;
224                     hashAlg = gc.platformAlg;
225                     break;
226                 default:
227                     // If any other permanent handle is present it is
228                     // a code defect.
229                     pAssert(FALSE);
230                     break;
231             }
232             break;
233         case TPM_HT_TRANSIENT:
234             // authPolicy for an object
235             {
236                 OBJECT *object = ObjectGet(handle);
237                 *authPolicy = object->publicArea.authPolicy;
238                 hashAlg = object->publicArea.nameAlg;
239             }
240             break;
241         case TPM_HT_NV_INDEX:
242             // authPolicy for a NV index
243             {
244                 NV_INDEX      nvIndex;
245                 NvGetIndexInfo(handle, &nvIndex);
246                 *authPolicy = nvIndex.publicArea.authPolicy;
247                 hashAlg = nvIndex.publicArea.nameAlg;
248             }
249             break;
250         case TPM_HT_PCR:
251             // authPolicy for a PCR
252             hashAlg = PCRGetAuthPolicy(handle, authPolicy);
253             break;
254         default:
255             // If any other handle type is present it is a code defect.
256             pAssert(FALSE);
257             break;
258     }
259     return hashAlg;
260 }

```

#### 9.6.3.4 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.

261 **UINT16**

```

262 EntityGetName(
263     TPMI_DH_ENTITY handle, // IN: handle of entity
264     BYTE *name // OUT: name of entity
265 )
266 {
267     switch(HandleGetType(handle))
268     {
269         case TPM_HT_TRANSIENT:
270             // Name for an object
271             return ObjectGetName(handle, name);
272             break;
273         case TPM_HT_NV_INDEX:
274             // Name for a NV index
275             return NvGetName(handle, name);
276             break;
277         default:
278             // For all other types, the handle is the Name
279             return TPM_HANDLE_Marshal(&handle, &name, NULL);
280             break;
281     }
282 }

```

### 9.6.3.5 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.

```

283 TPMI_RH_HIERARCHY
284 EntityGetHierarchy(
285     TPMI_DH_ENTITY handle // IN :handle of entity
286 )
287 {
288     switch(HandleGetType(handle))
289     {
290         case TPM_HT_PERMANENT:
291             // hierarchy for a permanent handle
292             switch(handle)
293             {
294                 case TPM_RH_PLATFORM:
295                 case TPM_RH_ENDORSEMENT:
296                 case TPM_RH_NULL:
297                     return handle;
298                     break;
299                 // all other permanent handles are associated with the owner
300                 // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
301                 default:
302                     return TPM_RH_OWNER;
303                     break;
304             }
305             break;
306         case TPM_HT_NV_INDEX:
307             // hierarchy for NV index
308             {
309                 NV_INDEX nvIndex;
310                 NvGetIndexInfo(handle, &nvIndex);
311                 // If only the platform can delete the index, then it is
312                 // considered to be in the platform hierarchy, otherwise it
313                 // is in the owner hierarchy.

```

```

314         if (nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
315             return TPM_RH_PLATFORM;
316         else
317             return TPM_RH_OWNER;
318     }
319     break;
320 case TPM_HT_TRANSIENT:
321     // hierarchy for an object
322     {
323         OBJECT *object;
324         object = ObjectGet(handle);
325         if (object->attributes.ppsHierarchy)
326         {
327             return TPM_RH_PLATFORM;
328         }
329         else if (object->attributes.epsHierarchy)
330         {
331             return TPM_RH_ENDORSEMENT;
332         }
333         else if (object->attributes.spsHierarchy)
334         {
335             return TPM_RH_OWNER;
336         }
337         else
338         {
339             return TPM_RH_NULL;
340         }
341     }
342     break;
343 case TPM_HT_PCR:
344     return TPM_RH_OWNER;
345     break;
346 default:
347     pAssert(0);
348     break;
349 }
350 // this is unreachable but it provides a return value for the default
351 // case which makes the compiler happy
352 return TPM_RH_NULL;
353 }

```

## 9.7 Global.c

### 9.7.1 Description

This file will instance the TPM variables that are not stack allocated. The descriptions for these variables is in Global.h.

### 9.7.2 Includes and Defines

```

1 #define GLOBAL_C
2 #include "InternalRoutines.h"

```

### 9.7.3 Global Data Values

These values are visible across multiple modules.

```

3 BOOL g_phEnable;
4 const UINT16 g_rcIndex[15] = {TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,
5 TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,
6 TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,

```

```

7             TPM_RC_D, TPM_RC_E, TPM_RC_F
8             };
9 TPM_HANDLE   g_exclusiveAuditSession;
10 UINT64      g_time;
11 BOOL        g_pcrReConfig;
12 TPMI_DH_OBJECT g_DRTMHandle;
13 BOOL        g_DrtmPreStartup;
14 BOOL        g_clearOrderly;
15 TPM_SU      g_prevOrderlyState;
16 BOOL        g_updateNV;
17 STATE_CLEAR_DATA gc;
18 STATE_RESET_DATA gr;
19 PERSISTENT_DATA gp;
20 ORDERLY_DATA go;

```

## 9.7.4 Private Values

### 9.7.4.1 SessionProcess.c

```

21 TPM_HANDLE   s_sessionHandles[MAX_SESSION_NUM];
22 TPMA_SESSION s_attributes[MAX_SESSION_NUM];
23 TPM_HANDLE   s_associatedHandles[MAX_SESSION_NUM];
24 TPM2B_NONCE  s_nonceCaller[MAX_SESSION_NUM];
25 TPM2B_AUTH   s_inputAuthValues[MAX_SESSION_NUM];
26 UINT32       s_encryptSessionIndex = UNDEFINED_INDEX;
27 UINT32       s_decryptSessionIndex = UNDEFINED_INDEX;
28 UINT32       s_auditSessionIndex = UNDEFINED_INDEX;
29 TPM2B_DIGEST s_cpHashForAudit;
30 UINT32       s_sessionNum;
31 BOOL         s_DAPendingOnNV = FALSE;
32 #ifndef TPM_CC_GetCommandAuditDigest
33 TPM2B_DIGEST s_cpHashForCommandAudit;
34 #endif

```

### 9.7.4.2 DA.c

```

35 UINT64      s_selfHealTimer = 0;
36 UINT64      s_lockoutTimer = 0;

```

### 9.7.4.3 NV.c

```

37 UINT32      s_reservedAddr[NV_RESERVE_LAST];
38 UINT32      s_reservedSize[NV_RESERVE_LAST];
39 UINT32      s_ramIndexSize;
40 BYTE        s_ramIndex[RAM_INDEX_SPACE];
41 UINT32      s_ramIndexSizeAddr;
42 UINT32      s_ramIndexAddr;
43 UINT32      s_maxCountAddr;
44 UINT32      s_evictNvStart;
45 UINT32      s_evictNvEnd;
46 TPM_RC      s_NvIsAvailable;

```

### 9.7.4.4 Object.c

```

47 OBJECT_SLOT s_objects[MAX_LOADED_OBJECTS];

```

### 9.7.4.5 PCR.c

```

48 PCR         s_pcrs[IMPLEMENTATION_PCR];

```

### 9.7.4.6 Session.c

```

49 SESSION_SLOT      s_sessions[MAX_LOADED_SESSIONS];
50 UINT32            s_oldestSavedSession;
51 int               s_freeSessionSlots;

```

### 9.7.4.7 Manufacture.c

```

52 BOOL              s_manufactured = FALSE;

```

### 9.7.4.8 Power.c

```

53 BOOL              s_initialized = FALSE;

```

## 9.8 Handle.c

### 9.8.1 Description

This file contains the functions that return the type of a handle.

### 9.8.2 Includes

```

1 #include "Tpm.h"
2 #include "InternalRoutines.h"

```

### 9.8.3 Functions

#### 9.8.3.1 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 }

```

#### 9.8.3.2 PermanentCapGetHandles()

This function returns a list of the permanent handles of PCR, started from *handle*. If *handle* is larger than the largest permanent handle, an empty list will be returned with *more* set to NO.

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

```

11 TPML_YES_NO
12 PermanentCapGetHandles(
13     TPM_HANDLE      handle,      // IN: start handle
14     UINT32          count,      // IN: count of returned handles
15     TPML_HANDLE     *handleList // OUT: list of handle

```



```

16 )
17 {
18     TPMI_YES_NO    more = NO;
19     UINT32         i;
20
21     pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
22
23     // Initialize output handle list
24     handleList->count = 0;
25
26     // The maximum count of handles we may return is MAX_CAP_HANDLES
27     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
28
29     // Iterate permanent handle range
30     for(i = handle; i <= PERMANENT_LAST; i++)
31     {
32         if(handleList->count < count)
33         {
34             // If we have not filled up the return list, add this permanent
35             // handle to it
36             handleList->handle[handleList->count] = i;
37             handleList->count++;
38         }
39         else
40         {
41             // If the return list is full but we still have permanent handle
42             // available, report this and stop iterating
43             more = YES;
44             break;
45         }
46     }
47     return more;
48 }

```

## 9.9 Locality.c

### 9.9.1 Includes

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

### 9.9.2 LocalityGetAttributes()

This function will convert a locality expressed as an integer into TPMA\_LOCALITY form.

The function returns the locality attribute.

```

2 TPMA_LOCALITY
3 LocalityGetAttributes(
4     UINT8         locality           // IN: locality value
5 )
6 {
7     TPMA_LOCALITY    locality_attributes;
8     BYTE            *localityAsByte = (BYTE *)&locality_attributes;
9
10    MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
11    switch(locality)
12    {
13        case 0:
14            locality_attributes.TPM_LOC_ZERO = SET;
15            break;
16        case 1:
17            locality_attributes.TPM_LOC_ONE = SET;
18            break;

```

```

19     case 2:
20         locality_attributes.TPM_LOC_TWO = SET;
21         break;
22     case 3:
23         locality_attributes.TPM_LOC_THREE = SET;
24         break;
25     case 4:
26         locality_attributes.TPM_LOC_FOUR = SET;
27         break;
28     default:
29         pAssert(locality < 256 && locality > 31);
30         *localityAsByte = locality;
31         break;
32     }
33     return locality_attributes;
34 }

```

## 9.10 Manufacture.c

### 9.10.1 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.

### 9.10.2 Includes and Data Definitions

```

1 #define MANUFACTURE_C
2 #include "InternalRoutines.h"

```

### 9.10.3 Functions

#### 9.10.3.1 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 remanufactured by calling *TPM\_Teardown()* first and then calling this function again.

Return Value	Meaning
0	success
1	manufacturing process previously performed

```

3 int
4 TPM_Manufacture(void)
5 {
6     TPM_SU          orderlyShutdown;
7     UINT64          totalResetCount = 0;
8
9     // If TPM has been manufactured, return indication.
10    if(s_manufactured)
11        return 1;
12
13    // initialize crypto units
14    CryptInitUnits();
15
16    // initialize NV
17    NvInit();
18
19    // default configuration for PCR

```

```

20     PCRSimStart();
21
22     // initialize pre-installed hierarchy data
23     // This should happen after NV is initialized because hierarchy data is
24     // stored in NV.
25     HierarchyPreInstall_Init();
26
27     // initialize dictionary attack parameters
28     DAPreInstall_Init();
29
30     // initialize PP list
31     PhysicalPresencePreInstall_Init();
32
33     // initialize command audit list
34     CommandAuditPreInstall_Init();
35
36     // first start up is required to be Startup(CLEAR)
37     orderlyShutdown = TPM_SU_CLEAR;
38     NvWriteReserved(NV_ORDERLY, &orderlyShutdown);
39
40     // initialize the firmware version
41     gp.firmwareV1 = FIRMWARE_V1;
42 #ifdef FIRMWARE_V2
43     gp.firmwareV2 = FIRMWARE_V2;
44 #else
45     gp.firmwareV2 = 0;
46 #endif
47     NvWriteReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
48     NvWriteReserved(NV_FIRMWARE_V2, &gp.firmwareV2);
49
50     // initialize the total reset counter to 0
51     NvWriteReserved(NV_TOTAL_RESET_COUNT, &totalResetCount);
52
53     // Commit NV writes. Manufacture process is an artificial process existing
54     // only in simulator environment and it is not defined in the specification
55     // that what should be the expected behavior if the NV write fails at this
56     // point. Therefore, it is assumed the NV write here is always success and
57     // no return code of this function is checked.
58     NvCommit();
59
60     s_manufactured = TRUE;
61
62     return 0;
63 }

```

### 9.10.3.2 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.

Return Value	Meaning
0	success
1	TPM not previously manufactured

```

64 int
65 TPM_TearDown(void)
66 {
67     // if TPM has not been manufactured, return indication

```

```

68     if(!s_manufactured)
69         return 1;
70
71     // stop crypt units
72     CryptStopUnits();
73
74     s_manufactured = FALSE;
75     return 0;
76 }

```

## 9.11 Marshal.c

### 9.11.1 Introduction

This file contains the marshaling and unmarshaling code of the simulator.

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 the a few unmarshaling routines are provided. Most of the others are similar.

NOTE A machine readable version of Marshal.c, and Marsha\_fp.h are available from the TCG.

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine. The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets ("<>") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

### 9.11.2 Unmarshal and Marshal a Value

In part 2, a TPMI\_DI\_OBJECT is defined by this table:

**Table xxx — Definition of (TPM\_HANDLE) TPMI\_DH\_OBJECT Type**

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```

1  TPM_RC
2  TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
3                          bool flag)
4  {
5      TPM_RC    result;
6      result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
7      if(result != TPM_RC_SUCCESS)
8          return result;
9      if (*target == TPM_RH_NULL) {
10         if(flag)
11             return TPM_RC_SUCCESS;
12         else
13             return TPM_RC_VALUE;
14     }
15     if((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
16         if((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST))
17             return TPM_RC_VALUE;

```

```

18     return TPM_RC_SUCCESS;
19 }

```

and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```

1  UINT16
2  TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
3  {
4      return UINT32_Marshal((UINT32 *)source, buffer, size);
5  }

```

### 9.11.3 Unmarshal and Marshal a Union

In part 2, a TPMU\_PUBLIC\_PARMS union is defined by:

Table xxx — Definition of TPMU\_PUBLIC\_PARMS Union <IN/OUT, S>

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign   encrypt   neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys

NOTE The Description column indicates which of TPMA\_OBJECT.decrypt or TPMA\_OBJECT.sign may be set. "+" indicates that both may be set but one shall be set. "[" indicates the optional settings.

From this table, the following unmarshaling code is generated.

```

1  TPM_RC
2  TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
3                              UINT32 selector)
4  {
5      switch(selector) {
6  #ifdef TPM_ALG_KEYEDHASH
7          case TPM_ALG_KEYEDHASH:
8              return TPMS_KEYEDHASH_PARMS_Unmarshal(
9                  (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
10 #endif
11 #ifdef TPM_ALG_SYMCIPHER
12          case TPM_ALG_SYMCIPHER:
13              return TPMT_SYM_DEF_OBJECT_Unmarshal(
14                  (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
15 #endif
16 #ifdef TPM_ALG_RSA
17          case TPM_ALG_RSA:
18              return TPMS_RSA_PARMS_Unmarshal(
19                  (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
20 #endif
21 #ifdef TPM_ALG_ECC
22          case TPM_ALG_ECC:
23              return TPMS_ECC_PARMS_Unmarshal(
24                  (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
25 #endif
26      }
27      return TPM_RC_SELECTOR;
28 }

```

NOTE The `#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:

```

1  UINT16
2  TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
3                             UINT32 selector)
4  {
5      switch(selector) {
6  #ifdef TPM_ALG_KEYEDHASH
7          case TPM_ALG_KEYEDHASH:
8              return TPMS_KEYEDHASH_PARMS_Marshal(
9                  (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
10 #endif
11 #ifdef TPM_ALG_SYMCIPHER
12          case TPM_ALG_SYMCIPHER:
13              return TPMT_SYM_DEF_OBJECT_Marshal(
14                  (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
15 #endif
16 #ifdef TPM_ALG_RSA
17          case TPM_ALG_RSA:
18              return TPMS_RSA_PARMS_Marshal(
19                  (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
20 #endif
21 #ifdef TPM_ALG_ECC
22          case TPM_ALG_ECC:
23              return TPMS_ECC_PARMS_Marshal(
24                  (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
25 #endif
26     }
27     assert(1);
28     return 0;
29 }

```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

### 9.11.4 Unmarshal and Marshal a Structure

In part 2, the TPMT\_PUBLiC structure is defined by:

**Table xxx — Definition of TPMT\_PUBLIC Structure**

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	“algorithm” associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with <i>type</i> , determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the <i>nameAlg</i> of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```

1  TPM_RC
2  TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, bool flag)
3  {
4      TPM_RC    result;
5      result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                          buffer, size);
7      if(result != TPM_RC_SUCCESS)
8          return result;
9      result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                     buffer, size, flag);
11     if(result != TPM_RC_SUCCESS)
12         return result;
13     result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                   buffer, size);
15     if(result != TPM_RC_SUCCESS)
16         return result;
17     result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                    buffer, size);
19     if(result != TPM_RC_SUCCESS)
20         return result;
21
22     result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                         buffer, size, (UINT32)target->type);
24     if(result != TPM_RC_SUCCESS)
25         return result;
26
27     result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                      buffer, size, (UINT32)target->type);
29     if(result != TPM_RC_SUCCESS)
30         return result;
31
32     return TPM_RC_SUCCESS;
33 }

```

The marshaling code for the TPMT\_PUBLIC structure is:

```
1  UINT16
2  TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
3  {
4      UINT16    result = 0;
5      result = (UINT16) (result + TPMI_ALG_PUBLIC_Marshal(
6          (TPMI_ALG_PUBLIC *) &(source->type), buffer, size));
7      result = (UINT16) (result + TPMI_ALG_HASH_Marshal(
8          (TPMI_ALG_HASH *) &(source->nameAlg), buffer, size)
9      ;
10     result = (UINT16) (result + TPMA_OBJECT_Marshal(
11         (TPMA_OBJECT *) &(source->objectAttributes), buffer, size));
12
13     result = (UINT16) (result + TPM2B_DIGEST_Marshal(
14         (TPM2B_DIGEST *) &(source->authPolicy), buffer, size));
15
16     result = (UINT16) (result + TPMU_PUBLIC_PARMS_Marshal(
17         (TPMU_PUBLIC_PARMS *) &(source->parameters), buffer, size,
18         (UINT32) source->type));
19
20     result = (UINT16) (result + TPMU_PUBLIC_ID_Marshal(
21         (TPMU_PUBLIC_ID *) &(source->unique), buffer, size,
22         (UINT32) source->type));
23
24     return result;
25 }
```



### 9.11.5 Unmarshal and Marshal an Array

In part 2, the TPML\_DIGEST is defined by:

**Table xxx — Definition of TPML\_DIGEST Structure**

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B\_DIGESTS). The auto-generated code to Unmarshal this structure is:

```

1  TPM_RC
2  TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
3  {
4      TPM_RC    result;
5      result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
6      if(result != TPM_RC_SUCCESS)
7          return result;
8
9      if( (target->count < 2))          // This check is triggered by the {2:} notation
10                                         // on 'count'
11          return TPM_RC_SIZE;
12
13      if((target->count) > 8)          // This check is triggered by the {:8} notation
14                                         // on 'digests'.
15          return TPM_RC_SIZE;
16
17      result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *) (target->digests),
18                                             buffer, size, (INT32) (target->count));
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      return TPM_RC_SUCCESS;
23  }

```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B\_DIGEST values. The unmarshaling code for the array is:

```

1  TPM_RC
2  TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                               INT32 count)
4  {
5      TPM_RC    result;
6      INT32 i;
7      for(i = 0; i < count; i++) {
8          result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9          if(result != TPM_RC_SUCCESS)
10             return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
14

```

Marshaling of the TPML\_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
1  UINT16
2  TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
3  {
4      UINT16    result = 0;
5      result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
6                                               size));
7      result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
8                  (TPM2B_DIGEST *) (source->digests), buffer, size,
9                  (INT32)(source->count)));
10
11     return result;
12 }
```

The marshaling code for the array is:

```
1  TPM_RC
2  TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                               INT32 count)
4  {
5      TPM_RC    result;
6      INT32    i;
7      for(i = 0; i < count; i++) {
8          result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9          if(result != TPM_RC_SUCCESS)
10             return result;
11     }
12     return TPM_RC_SUCCESS;
13 }
```

### 9.11.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 9.11.5 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.

**Table xxx — Definition of TPM2B\_EVENT Structure**

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

```

1  TPM_RC
2  TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3  {
4      TPM_RC    result;
5      result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6      if(result != TPM_RC_SUCCESS)
7          return result;
8
9      // if size equal to 0, the rest of the structure is a zero buffer. Stop
10     processing
11     if(target->t.size == 0)
12         return TPM_RC_SUCCESS;
13
14     if((target->t.size) > 1024)    // This check is triggered by the {:1024} notation
15         return TPM_RC_SIZE;    // on 'buffer'
16
17     result = BYTE_Array_Unmarshal((BYTE *) (target->t.buffer), buffer, size,
18                                  (INT32) (target->t.size));
19     if(result != TPM_RC_SUCCESS)
20         return result;
21
22     return TPM_RC_SUCCESS;
23 }

```

Which use these structure definitions:

```

1  typedef struct {
2      UINT16    size;
3      BYTE      buffer[1];
4  } TPM2B;
5
6  typedef struct {
7      UINT16    size;
8      BYTE      buffer[1024];
9  } EVENT_2B;
10
11 typedef union {
12     EVENT_2B    t;    // The type-specific union member
13     TPM2B      b;    // The generic union member
14 } TPM2B_EVENT;

```

## 9.12 MemoryLib.c

### 9.12.1 Description

This file contains a set of miscellaneous memory manipulation routines. Many of the functions have the same semantics as functions defined in string.h. Those functions are not used in the TPM in order to avoid namespace contamination.

### 9.12.2 Includes and Data Definitions

```
1 #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.

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.

```
2 static UINT32 s_actionInputBuffer[1024]; // action input buffer
3 static UINT32 s_actionOutputBuffer[1024]; // action output buffer
4 static BYTE s_responseBuffer[MAX_RESPONSE_SIZE]; // response buffer
```

### 9.12.3 Functions

#### 9.12.3.1 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.

```
5 void
6 MemoryCopy(
7     void *destination, // OUT: copy destination
8     void *source, // IN: copy source
9     UINT32 size // IN: number of octets being copied
10 )
11 {
12     BYTE *p = (BYTE *)source;
13     BYTE *q = (BYTE *)destination;
14     while (size--)
15         *q++ = *p++;
16     return;
17 }
```

#### 9.12.3.2 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.

```
18 void
19 MemoryMove(
20     void *destination, // OUT: move destination
21     const void *source, // IN: move source
22     UINT32 size // IN: number of octets to moved
```

```

23 )
24 {
25     const BYTE *p = (BYTE *)source;
26     BYTE *q = (BYTE *)destination;
27     // if the destination buffer has a lower address than the
28     // source, then moving bytes in ascending order is safe.
29     if (p>q || (p+size <= q))
30     {
31         while (size--)
32             *q++ = *p++;
33     }
34     // If the destination buffer has a higher address than the
35     // source, then move bytes from the end to the beginning.
36     else if (p<q)
37     {
38         p += size;
39         q += size;
40         while (size--)
41             *--q = *--p;
42     }
43     // If the source and destination address are the same, nothing to move.
44     return;
45 }

```

### 9.12.3.3 MemoryEqual()

This function indicates if two buffers have the same values in the indicated number of bytes.

Return Value	Meaning
TRUE	all octets are the same
FALSE	all octets are not the same

```

46 BOOL
47 MemoryEqual(
48     const void    *buffer1,           // IN: compare buffer1
49     const void    *buffer2,           // IN: compare buffer2
50     UINT32        size                // IN: size of bytes being compared
51 )
52 {
53     BOOL         equal = TRUE;
54     const BYTE  *b1, *b2;
55
56     b1 = (BYTE *)buffer1;
57     b2 = (BYTE *)buffer2;
58
59     // Compare all bytes so that there is no leakage of information
60     // due to timing differences.
61     for(; size > 0; size--)
62         equal = (*b1++ == *b2++) && equal;
63
64     return equal;
65 }

```

### 9.12.3.4 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.

```

66  INT16
67  MemoryCopy2B(
68      TPM2B      *dest,          // OUT: receiving TPM2B
69      const TPM2B *source       // IN: source TPM2B
70  )
71  {
72      dest->size = source->size;
73      MemoryMove(dest->buffer, source->buffer, dest->size);
74      return dest->size;
75  }

```

### 9.12.3.5 MemoryConcat2B()

This function will concatenate the buffer contents of a TPM2B to an the buffer contents of another TPM2B and adjust the size accordingly ( $a := (a | b)$ ).

```

76  void
77  MemoryConcat2B(
78      TPM2B      *aInOut,       // IN/OUT: destination 2B
79      TPM2B      *bIn          // IN: second 2B
80  )
81  {
82      MemoryCopy(&aInOut->buffer[aInOut->size], bIn->buffer, bIn->size);
83      aInOut->size = aInOut->size + bIn->size;
84      return;
85  }

```

### 9.12.3.6 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.

Return Value	Meaning
TRUE	size and buffer contents are the same
FALSE	size or buffer contents are not the same

```

86  BOOL
87  Memory2BEqual(
88      const TPM2B *aIn,        // IN: compare value
89      const TPM2B *bIn        // IN: compare value
90  )
91  {
92      if(aIn->size != bIn->size)
93          return FALSE;
94
95      return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
96  }

```

### 9.12.3.7 MemorySet()

This function will set all the octets in the specified memory range to the specified octet value.

```

97  void
98  MemorySet(
99      void      *destination,  // OUT: memory destination
100     char      value,         // IN: fill value
101     UINT32    size          // IN: number of octets to fill
102  )
103  {

```

```

104     char *p = destination;
105     while (size--)
106         *p++ = value;
107     return;
108 }

```

### 9.12.3.8 MemoryGetActionInputBuffer()

This function returns the address of the buffer into which the command parameters will be unmarshaled in preparation for calling the command actions.

```

109 BYTE *
110 MemoryGetActionInputBuffer(
111     UINT32     size           // Size, in bytes, required for the input unmarshaling
112 )
113 {
114     // In this implementation, a static buffer is set aside for action output.
115     // Other implementations may apply additional optimization based on command
116     // code or other factors.
117     UINT32     *p = s_actionInputBuffer;
118     if(size == 0)
119         return NULL;
120     pAssert(size < <K>sizeof(s_actionInputBuffer));
121     #define SZ     sizeof(s_actionInputBuffer[0])
122
123     for(size = (size + SZ - 1) / SZ; size > 0; size--)
124         *p++ = 0;
125     #undef SZ
126
127     return (BYTE *)s_actionInputBuffer;
128 }

```

### 9.12.3.9 MemoryGetActionOutputBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```

129 void *
130 MemoryGetActionOutputBuffer(
131     TPM_CC     command       // Command that requires the buffer
132 )
133 {
134     // In this implementation, a static buffer is set aside for action output.
135     // Other implementations may apply additional optimization based on the command
136     // code or other factors.
137     command = 0;           // Unreferenced parameter
138     return s_actionOutputBuffer;
139 }

```

### 9.12.3.10 MemoryGetResponseBuffer()

This function returns the address into which the command response is marshaled from values in the action output buffer.

```

140 BYTE *
141 MemoryGetResponseBuffer(
142     TPM_CC     command       // Command that requires the buffer
143 )
144 {
145     // In this implementation, a static buffer is set aside for responses.
146     // Other implementation may apply additional optimization based on the command

```

```

147     // code or other factors.
148     command = 0;           // Unreferenced parameter
149     return s_responseBuffer;
150 }

```

### 9.12.3.11 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.

```

151 UINT16
152 MemoryRemoveTrailingZeros (
153     TPM2B_AUTH      *auth           // IN/OUT: value to adjust
154 )
155 {
156     BYTE            *a = &auth->t.buffer[auth->t.size-1];
157     for(; auth->t.size > 0; auth->t.size--)
158     {
159         if(*a--)
160             break;
161     }
162     return auth->t.size;
163 }

```

## 9.13 Power.c

### 9.13.1 Description

This file contains functions that receive the simulated power state transitions of the TPM.

### 9.13.2 Includes and Data Definitions

```

1  #define POWER_C
2  #include "InternalRoutines.h"

```

### 9.13.3 Functions

#### 9.13.3.1 TPMInit()

This function is used to process a power on event.

```

3  void
4  TPMInit(void)
5  {
6      // Set state as not initialized
7      s_initialized = FALSE;
8
9      return;
10 }

```

#### 9.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2\_Startup() has completed successfully).

```

11 void

```



```

12  TPMRegisterStartup(void)
13  {
14      s_initialized = TRUE;
15
16      return;
17  }

```

### 9.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2\_Startup() has completed successfully after a \_TPM\_Init()).

Return Value	Meaning
TRUE	TPM has been initialized
FALSE	TPM has not been initialized

```

18  BOOL
19  TPMIsStarted(void)
20  {
21      return s_initialized;
22  }

```

## 9.14 PropertyCap.c

### 9.14.1 Description

This file contains the functions that are used for accessing the TPM\_CAP\_TPM\_PROPERTY values.

### 9.14.2 Includes

```

1  #include "InternalRoutines.h"

```

### 9.14.3 Functions

#### 9.14.3.1 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.

Return Value	Meaning
TRUE	referenced property exists and <i>value</i> 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 the specification

```

```

12         // For this specification, the value is "2.0".
13         *value = TPM_SPEC_FAMILY;
14         break;
15     case TPM_PT_LEVEL:
16         // from the title page of the specification
17         *value = TPM_SPEC_LEVEL;
18         break;
19     case TPM_PT_REVISION:
20         // from the title page of the specification
21         *value = TPM_SPEC_VERSION;
22         break;
23     case TPM_PT_DAY_OF_YEAR:
24         // computed from the date value on the title page of the specification
25         *value = TPM_SPEC_DAY_OF_YEAR;
26         break;
27     case TPM_PT_YEAR:
28         // from the title page of the specification
29         *value = TPM_SPEC_YEAR;
30         break;
31     case TPM_PT_MANUFACTURER:
32         // vendor ID unique to each TPM manufacturer
33         *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
34         break;
35     case TPM_PT_VENDOR_STRING_1:
36         // first four characters of the vendor ID string
37         *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
38         break;
39     case TPM_PT_VENDOR_STRING_2:
40         // second four characters of the vendor ID string
41 #ifndef 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 #ifndef 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 #ifndef VENDOR_STRING_4
58         *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
59 #else
60         *value = 0;
61 #endif
62         break;
63     case TPM_PT_VENDOR_TPM_TYPE:
64         // vendor-defined value indicating the TPM model
65         *value = 1;
66         break;
67     case TPM_PT_FIRMWARE_VERSION_1:
68         // more significant 32-bits of a vendor-specific value
69         *value = gp.firmwareV1;
70         break;
71     case TPM_PT_FIRMWARE_VERSION_2:
72         // less significant 32-bits of a vendor-specific value
73         *value = gp.firmwareV2;
74         break;
75     case TPM_PT_INPUT_BUFFER:
76         // maximum size of TPM2B_MAX_BUFFER
77         *value = MAX_DIGEST_BUFFER;

```

```

78         break;
79     case TPM_PT_HR_TRANSIENT_MIN:
80         // minimum number of transient objects that can be held in TPM
81         // RAM
82         *value = MAX_LOADED_OBJECTS;
83         break;
84     case TPM_PT_HR_PERSISTENT_MIN:
85         // minimum number of persistent objects that can be held in
86         // TPM NV memory
87         // In this implementation, there is no minimum number of
88         // persistent objects.
89         *value = MIN_EVICT_OBJECTS;
90         break;
91     case TPM_PT_HR_LOADED_MIN:
92         // minimum number of authorization sessions that can be held in
93         // TPM RAM
94         *value = MAX_LOADED_SESSIONS;
95         break;
96     case TPM_PT_ACTIVE_SESSIONS_MAX:
97         // number of authorization sessions that may be active at a time
98         *value = MAX_ACTIVE_SESSIONS;
99         break;
100    case TPM_PT_PCR_COUNT:
101        // number of PCR implemented
102        *value = IMPLEMENTATION_PCR;
103        break;
104    case TPM_PT_PCR_SELECT_MIN:
105        // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
106        *value = PCR_SELECT_MIN;
107        break;
108    case TPM_PT_CONTEXT_GAP_MAX:
109        // maximum allowed difference (unsigned) between the contextID
110        // values of two saved session contexts
111        *value = (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:
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 this reference implementation, all of these values are 0.
216         *value = 0;
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_ECDSA
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;
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(g_prevOrderlyState != SHUTDOWN_NONE)
308                 flags.orderly = SET;
309
310             // Note: Different compilers may require a different method to cast
311             // a bit field structure to a UINT32.
312             *value = * (UINT32 *) &flags;
313             break;
314         }
315     case TPM_PT_HR_NV_INDEX:
316         // number of NV indexes currently defined
317         *value = NvCapGetIndexNumber();
318         break;
319     case TPM_PT_HR_LOADED:
320         // number of authorization sessions currently loaded into TPM
321         // RAM
322         *value = SessionCapGetLoadedNumber();
323         break;
324     case TPM_PT_HR_LOADED_AVAIL:
325         // number of additional authorization sessions, of any type,
326         // that could be loaded into TPM RAM
327         *value = SessionCapGetLoadedAvail();
328         break;
329     case TPM_PT_HR_ACTIVE:
330         // number of active authorization sessions currently being
331         // tracked by the TPM
332         *value = SessionCapGetActiveNumber();
333         break;
334     case TPM_PT_HR_ACTIVE_AVAIL:
335         // number of additional authorization sessions, of any type,
336         // that could be created
337         *value = SessionCapGetActiveAvail();
338         break;
339     case TPM_PT_HR_TRANSIENT_AVAIL:
340         // estimate of the number of additional transient objects that
341         // could be loaded into TPM RAM

```

```

342         *value = ObjectCapGetTransientAvail();
343         break;
344     case TPM_PT_HR_PERSISTENT:
345         // number of persistent objects currently loaded into TPM
346         // NV memory
347         *value = NvCapGetPersistentNumber();
348         break;
349     case TPM_PT_HR_PERSISTENT_AVAIL:
350         // number of additional persistent objects that could be loaded
351         // into NV memory
352         *value = NvCapGetPersistentAvail();
353         break;
354     case TPM_PT_NV_COUNTERS:
355         // number of defined NV indexes that have NV TPMA_NV_COUNTER
356         // attribute SET
357         *value = NvCapGetCounterNumber();
358         break;
359     case TPM_PT_NV_COUNTERS_AVAIL:
360         // number of additional NV indexes that can be defined with their
361         // TPMA_NV_COUNTER attribute SET
362         *value = NvCapGetCounterAvail();
363         break;
364     case TPM_PT_ALGORITHM_SET:
365         // region code for the TPM
366         *value = gp.algorithmSet;
367         break;
368
369     case TPM_PT_LOADED_CURVES:
370     #ifdef TPM_ALG_ECC
371         // number of loaded ECC curves
372         *value = CryptCapGetEccCurveNumber();
373     #else // TPM_ALG_ECC
374         *value = 0;
375     #endif // TPM_ALG_ECC
376         break;
377
378     case TPM_PT_LOCKOUT_COUNTER:
379         // current value of the lockout counter
380         *value = gp.failedTries;
381         break;
382     case TPM_PT_MAX_AUTH_FAIL:
383         // number of authorization failures before DA lockout is invoked
384         *value = gp.maxTries;
385         break;
386     case TPM_PT_LOCKOUT_INTERVAL:
387         // number of seconds before the value reported by
388         // TPM_PT_LOCKOUT_COUNTER is decremented
389         *value = gp.recoveryTime;
390         break;
391     case TPM_PT_LOCKOUT_RECOVERY:
392         // number of seconds after a lockoutAuth failure before use of
393         // lockoutAuth may be attempted again
394         *value = gp.lockoutRecovery;
395         break;
396     case TPM_PT_AUDIT_COUNTER_0:
397         // high-order 32 bits of the command audit counter
398         *value = (UINT32) (gp.auditCounter >> 32);
399         break;
400     case TPM_PT_AUDIT_COUNTER_1:
401         // low-order 32 bits of the command audit counter
402         *value = (UINT32) (gp.auditCounter);
403         break;
404     default:
405         // property is not defined
406         return FALSE;
407         break;

```

```

408     }
409
410     return TRUE;
411 }

```

### 9.14.3.2 TPMCapGetProperties()

This function is used to get the TPM\_PT values. The search of properties will start at *property* and continue until *propertyList* has as many values as will fit, or the last property has been reported, or the list has as many values as requested in *count*.

Return Value	Meaning
YES	more properties are available
NO	no more properties to be reported

```

412     TPMI_YES_NO
413     TPMCapGetProperties(
414         TPM_PT                property,        // IN: the starting TPM property
415         UINT32                count,          // IN: maximum number of returned
416                                 //          properties
417         TPML_TAGGED_TPM_PROPERTY *propertyList // OUT: property list
418     )
419     {
420         TPMI_YES_NO    more = NO;
421         UINT32         i;
422
423         // initialize output property list
424         propertyList->count = 0;
425
426         // maximum count of properties we may return is MAX_PCR_PROPERTIES
427         if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;
428
429         // If property is less than PT_FIXED, start from PT_FIXED.
430         if(property < PT_FIXED) property = PT_FIXED;
431
432         // Scan through the TPM properties of the requested group.
433         // The size of TPM property group is PT_GROUP * 2 for fix and
434         // variable groups.
435         for(i = property; i <= PT_FIXED + PT_GROUP * 2; i++)
436         {
437             UINT32    value;
438             if(TPMPropertyIsDefined((TPM_PT) i, &value))
439             {
440                 if(propertyList->count < count)
441                 {
442
443                     // If the list is not full, add this property
444                     propertyList->tpmProperty[propertyList->count].property =
445                         (TPM_PT) i;
446                     propertyList->tpmProperty[propertyList->count].value = value;
447                     propertyList->count++;
448                 }
449                 else
450                 {
451                     // If the return list is full but there are more properties
452                     // available, set the indication and exit the loop.
453                     more = YES;
454                     break;
455                 }
456             }
457         }
458         return more;

```



459 }

## Cryptographic Functions

### 9.15 Introduction

The files in this section provide cryptographic support and interface to the CryptoEngine.

### 9.16 CryptUtil.c

#### 9.16.1 Introduction

This module contains the interfaces to the *CryptoEngine()* and provides miscellaneous cryptographic functions in support of the TPM.

#### 9.16.2 Includes

```

1  #include    "TPM_Types.h"
2  #include    "CryptPri.h"           // types shared by CryptUtil and CryptoEngine.
3                                           // Includes the function prototypes for the
4                                           // CryptoEngine functions.
5  #include    "Global.h"
6  #include    "CryptUtil_fp.h"      // Declared here.
7  #include    "InternalRoutines.h"
8  #include    "MemoryLib_fp.h"

```

#### 9.16.3 TranslateCryptErrors()

This function converts errors from the cryptographic library into TPM\_RC\_VALUES.

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

```

9  static TPM_RC
10 TranslateCryptErrors (
11     CRYPT_RESULT    retVal           // IN: crypt error to evaluate
12 )
13 {
14     switch (retVal)
15     {
16     case CRYPT_SUCCESS:
17         return TPM_RC_SUCCESS;
18     case CRYPT_FAIL:
19         return TPM_RC_VALUE;
20     case CRYPT_NO_RESULT:
21         return TPM_RC_NO_RESULT;
22     case CRYPT_SCHEME:
23         return TPM_RC_SCHEME;
24     case CRYPT_PARAMETER:

```

```

25     return TPM_RC_VALUE;
26 case CRYPT_UNDERFLOW:
27     return TPM_RC_SIZE;
28 case CRYPT_POINT:
29     return TPM_RC_ECC_POINT;
30 case CRYPT_CANCEL:
31     return TPM_RC_CANCELED;
32 default: // Other unknown warnings
33     return TPM_RC_FAILURE;
34 }
35 }

```

## 9.16.4 Random Number Generation Functions

### 9.16.4.1 CryptStirRandom()

Stir random entropy

```

36 #ifdef TPM_ALG_NULL //%
37 void
38 CryptStirRandom(
39     UINT32          entropySize,      // IN: size of entropy buffer
40     BYTE           *buffer           // IN: entropy buffer
41 )
42 {
43     // RNG self testing code may be inserted here
44
45     // Call crypto engine random number stirring function
46     _cpri__StirRandom(entropySize, buffer);
47
48     return;
49 }

```

### 9.16.4.2 CryptGenerateRandom()

This is the interface to `_cpri__GenerateRandom()`.

```

50 UINT16
51 CryptGenerateRandom(
52     UINT16          randomSize,      // IN: size of random number
53     BYTE           *buffer           // OUT: buffer of random number
54 )
55 {
56     // Call crypto engine random number generation
57     return _cpri__GenerateRandom(randomSize, buffer);
58 }
59 #endif //TPM_ALG_NULL //%

```

## 9.16.5 Hash/HMAC Functions

### 9.16.5.1 CryptGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```

60 #ifdef TPM_ALG_KEYEDHASH //% 1
61 TPM_ALG_ID
62 CryptGetContextAlg(
63     void           *state           // IN: the context to check
64 )

```

```

65 {
66     HASH_STATE *context = (HASH_STATE *)state;
67     return _cpri__GetContextAlg(&context->state);
68 }

```

### 9.16.5.2 CryptStartHash()

This function starts a hash and return the size, in bytes, of the digest.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the <i>hashAlg</i> was TPM_ALG_NULL

```

69     UINT16
70     CryptStartHash(
71         TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
72         HASH_STATE      *hashState        // OUT: the state of hash stack. It
73                                           // will be used in hash update
74                                           // and completion
75     )
76 {
77     CRYPT_RESULT    retVal;
78     pAssert(hashState != NULL);
79
80     // Set the state type
81     hashState->type = HASH_STATE_HASH;
82
83     // Call crypto engine start hash function
84     if((retVal = _cpri__StartHash(hashAlg, FALSE, &hashState->state)) == 0)
85         hashState->type = HASH_STATE_EMPTY;
86
87     return retVal;
88 }

```

### 9.16.5.3 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.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the <i>hashAlg</i> was TPM_ALG_NULL

```

89     UINT16
90     CryptStartHashSequence(
91         TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
92         HASH_STATE      *hashState        // OUT: the state of hash stack. It
93                                           // will be used in hash update
94                                           // and completion
95     )
96 {
97     CRYPT_RESULT    retVal;
98
99     pAssert(hashState != NULL);
100
101     // Set the state type
102     hashState->type = HASH_STATE_HASH;
103
104     // Call crypto engine start hash function

```

```

105     if((retVal = _cpri_StartHash(hashAlg, TRUE, &hashState->state)) == 0)
106         hashState->type = HASH_STATE_EMPTY;
107
108     return retVal;
109 }

```

#### 9.16.5.4 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.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the <i>hashAlg</i> was TPM_ALG_NULL

```

110     UINT16
111     CryptStartHMAC(
112         TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
113         UINT16           keySize,          // IN: the size of HMAC key in bytes
114         BYTE             *key,            // IN: HMAC key
115         HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It
116                                         // will be used in HMAC update
117                                         // and completion
118     )
119 {
120     HASH_STATE    *hashState = (HASH_STATE *)hmacState;
121     CRYPT_RESULT  retVal;
122
123     if((retVal = _cpri_StartHMAC(hashAlg, FALSE, &hashState->state, keySize, key,
124                                 &hmacState->hmacKey.b)) > 0)
125         hashState->type = HASH_STATE_HMAC;
126     else
127         hashState->type = HASH_STATE_EMPTY;
128
129     return retVal;
130 }

```

#### 9.16.5.5 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.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the <i>hashAlg</i> was TPM_ALG_NULL

```

131     UINT16
132     CryptStartHMACSequence(
133         TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
134         UINT16           keySize,          // IN: the size of HMAC key in bytes
135         BYTE             *key,            // IN: HMAC key
136         HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It
137                                         // will be used in HMAC update

```

```

138                                     //      and completion
139 )
140 {
141     HASH_STATE      *hashState = (HASH_STATE *)hmacState;
142     CRYPT_RESULT    retVal;
143
144     if((retVal = _cpri__StartHMAC(hashAlg, TRUE, &hashState->state,
145                                 keySize, key, &hmacState->hmacKey.b)) > 0)
146         hashState->type = HASH_STATE_HMAC;
147     else
148         hashState->type = HASH_STATE_EMPTY;
149
150     return retVal;
151 }

```

### 9.16.5.6 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.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the <i>hashAlg</i> was TPM_ALG_NULL

```

152  UINT16
153  CryptStartHMAC2B(
154      TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
155      TPM2B            *key,             // IN: HMAC key
156      HMAC_STATE      *hmacState        // OUT: the state of HMAC stack. It
157                                         // will be used in HMAC update
158                                         // and completion
159  )
160  {
161      return CryptStartHMAC(hashAlg, key->size, key->buffer, hmacState);
162  }

```

### 9.16.5.7 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.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the <i>hashAlg</i> was TPM_ALG_NULL

```

163  UINT16
164  CryptStartHMACSequence2B(
165      TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
166      TPM2B            *key,             // IN: HMAC key
167      HMAC_STATE      *hmacState        // OUT: the state of HMAC stack. It
168                                         // will be used in HMAC update
169                                         // and completion

```

```

170 )
171 {
172     return CryptStartHMACSequence(hashAlg, key->size, key->buffer, hmacState);
173 }

```

### 9.16.5.8 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.

```

174 void
175 CryptUpdateDigest(
176     void                *digestState,        // IN: the state of hash stack
177     UINT32              dataSize,           // IN: the size of data
178     BYTE                *data               // IN: data to be hashed
179 )
180 {
181     HASH_STATE          *hashState = (HASH_STATE *)digestState;
182
183     pAssert(digestState != NULL);
184     if(hashState->type == HASH_STATE_EMPTY)
185         return;
186
187     // If no data, nothing to do (this is not an error)
188     if(data == NULL || dataSize == 0)
189         return;
190
191     // Call crypto engine update hash function
192     _cpri_UpdateHash(&hashState->state, dataSize, data);
193     return;
194 }

```

### 9.16.5.9 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.

```

195 void
196 CryptUpdateDigest2B(
197     void                *digestState,        // IN: the digest state
198     TPM2B               *bIn               // IN: 2B containing the data
199 )
200 {
201     // Only compute the digest if a pointer to the 2B is provided.
202     // In CryptUpdateDigest(), if size is zero or buffer is NULL, then no change
203     // to the digest occurs. This function should not provide a buffer if bIn is
204     // not provided.
205     if(bIn != NULL)
206         CryptUpdateDigest(digestState, bIn->size, bIn->buffer);
207     return;
208 }

```

### 9.16.5.10 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()*.

```

209 void
210 CryptUpdateDigestInt(
211     void          *state,          // IN: the state of hash stack
212     UINT32        intSize,        // IN: the size of 'intValue' in bytes
213     void          *intValue       // IN: integer value to be hashed
214 )
215 {
216
217 #if BIG_ENDIAN_TPM == YES
218     CryptUpdateHash(state, intSize, (BYTE *)intValue);
219 #else
220
221     BYTE        marshalBuffer[8];
222     // Point to the big end of a little-endian value
223     BYTE        *p = &((BYTE *)intValue)[intSize - 1];
224     // Point to the big end of a big-endian value
225     BYTE        *q = marshalBuffer;
226
227     pAssert(intSize <= 8 && intSize != 0 && intValue != NULL);
228     switch (intSize)
229     {
230     case 8:
231         *q++ = *p--;
232         *q++ = *p--;
233         *q++ = *p--;
234         *q++ = *p--;
235     case 4:
236         *q++ = *p--;
237         *q++ = *p--;
238     case 2:
239         *q++ = *p--;
240     case 1:
241         *q = *p;
242         break;
243     default:
244         pAssert(TRUE);
245         return;
246     }
247     // Call update the hash
248     CryptUpdateDigest(state, intSize, marshalBuffer);
249 #endif
250
251     return;
252 }

```

#### 9.16.5.11 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

Return Value	Meaning
>=0	the number of bytes placed in <i>digest</i>

```

253 UINT16
254 CryptCompleteHash(
255     void          *state,          // IN: the state of hash stack
256     UINT16        digestSize,     // IN: size of digest buffer
257     BYTE          *digest         // OUT: hash digest

```



```

258 )
259 {
260     HASH_STATE      *hashState = (HASH_STATE *)state;      // local value
261
262     // If this is a HMAC state, forward the call
263     if(hashState->type == HASH_STATE_HMAC)
264         return(CryptCompleteHMAC((HMAC_STATE *)state, digestSize, digest));
265
266     // If this is not has hash state, return nothing
267     if(hashState->type != HASH_STATE_HASH)
268         return 0;
269
270     // Set the state to empty so that it doesn't get used again
271     hashState->type = HASH_STATE_EMPTY;
272
273     // Call crypto engine complete hash function
274     return      _cpri__CompleteHash(&hashState->state, digestSize, digest);
275 }

```

### 9.16.5.12 CryptCompleteHash2B()

This function is the same as *CryptCompleteHash()* but the digest is placed in a TPM2B. This is the most common use and this is provided for specification clarity. 'digest. size' should be set to indicate the number of bytes to place in the buffer

Return Value	Meaning
>=0	the number of bytes placed in 'digest. buffer'

```

276     UINT16
277     CryptCompleteHash2B(
278         void          *state,          // IN: the state of hash stack
279         TPM2B         *digest         // IN: the size of the buffer
280                                     // Out: requested number of bytes
281     )
282     {
283         if(digest == NULL)
284             return 0;
285
286         return CryptCompleteHash(state, digest->size, digest->buffer);
287     }

```

### 9.16.5.13 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.

Return Value	Meaning
>=0	the number of bytes placed in <i>ret</i>

```

288     UINT16
289     CryptHashBlock(
290         TPM_ALG_ID    algId,          // IN: the hash algorithm to use
291         UINT16        blockSize,     // IN: size of the data block
292         BYTE          *block,        // IN: address of the block to hash
293         UINT16        retSize,       // IN: size of the return buffer
294         BYTE          *ret           // OUT: address of the buffer
295     )
296     {
297         return _cpri__HashBlock(algId, blockSize, block, retSize, ret);
298     }

```

### 9.16.5.14 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.

Return Value	Meaning
>=0	the number of bytes placed in <i>digest</i>

```

299  UINT16
300  CryptCompleteHMAC(
301      HMAC_STATE      *hmacState,          // IN: the state of HMAC stack
302      UINT32          digestSize,         // IN: size of digest buffer
303      BYTE            *digest            // OUT: HMAC digest
304  )
305  {
306      HASH_STATE      *hashState;
307
308      pAssert(hmacState != NULL);
309      hashState = &hmacState->hashState;
310      if(hashState->type == HASH_STATE_EMPTY)
311          return 0;
312
313      pAssert(hashState->type == HASH_STATE_HMAC);
314      hashState->type = HASH_STATE_EMPTY;
315
316      return _cpri__CompleteHMAC(&hashState->state, &hmacState->hmacKey.b,
317                                digestSize, digest);
318  }
319  
```

### 9.16.5.15 CryptCompleteHMAC2B()

This function is the same as *CryptCompleteHMAC()* but the HMAC result is returned in a TPM2B which is the most common use.

Return Value	Meaning
>=0	the number of bytes placed in <i>digest</i>

```

320  UINT16
321  CryptCompleteHMAC2B(
322      HMAC_STATE      *hmacState,          // IN: the state of HMAC stack
323      TPM2B           *digest            // OUT: HMAC
324  )
325  {
326      if(digest == NULL)
327          return 0;
328
329      return CryptCompleteHMAC(hmacState, digest->size, digest->buffer);
330  }
  
```

### 9.16.5.16 CryptGetHashDigestSize()

This function returns the digest size in bytes for a hash algorithm.

Return Value	Meaning
0	digest size for TPM_ALG_NULL
> 0	digest size

```

331  UINT16
332  CryptGetHashDigestSize(
333      TPM_ALG_ID      hashAlg          // IN: hash algorithm
334  )
335  {
336      return _cpri__GetDigestSize(hashAlg);
337  }

```

#### 9.16.5.17 CryptGetHashBlockSize()

Get the digest size in byte of a hash algorithm.

Return Value	Meaning
0	block size for TPM_ALG_NULL
> 0	block size

```

338  UINT16
339  CryptGetHashBlockSize(
340      TPM_ALG_ID      hash              // IN: hash algorithm to look up
341  )
342  {
343      return _cpri__GetHashBlockSize(hash);
344  }

```

#### 9.16.5.18 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.

Return Value	Meaning
TPM_ALG_XXX ()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```

345  TPM_ALG_ID
346  CryptGetHashAlgByIndex(
347      UINT32      index          // IN: the index
348  )
349  {
350      return _cpri__GetHashAlgByIndex(index);
351  }

```

#### 9.16.5.19 CryptSignHMAC()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

Error Returns	Meaning
---------------	---------

```

352  static TPM_RC

```

```

353 CryptSignHMAC(
354     OBJECT          *signKey,           // IN: HMAC key sign the hash
355     TPMT_SIG_SCHEME *scheme,           // IN: signing scheme
356     TPM2B_DIGEST    *hashData,        // IN: hash to be signed
357     TPMT_SIGNATURE  *signature        // OUT: signature
358 )
359 {
360     HMAC_STATE      hmacState;
361     UINT32          digestSize;
362
363     // HMAC algorithm self testing code may be inserted here
364
365     digestSize = CryptStartHMAC2B(scheme->details.hmac.hashAlg,
366                                  &signKey->sensitive.sensitive.bits.b,
367                                  &hmacState);
368
369     // The hash algorithm must be a valid one.
370     pAssert(digestSize > 0);
371
372     CryptUpdateDigest2B(&hmacState, &hashData->b);
373
374     CryptCompleteHMAC(&hmacState, digestSize,
375                      (BYTE *) &signature->signature.hmac.digest);
376
377     // Set HMAC algorithm
378     signature->signature.hmac.hashAlg = scheme->details.hmac.hashAlg;
379
380     return TPM_RC_SUCCESS;
381 }

```

#### 9.16.5.20 CryptHMACVerifySignature()

This function will verify a signature signed by a HMAC key.

Error Returns	Meaning
TPM_RC_SIGNATURE	if invalid input or signature is not genuine

```

382 static TPM_RC
383 CryptHMACVerifySignature(
384     OBJECT          *signKey,           // IN: HMAC key signed the hash
385     TPM2B_DIGEST    *hashData,        // IN: digest being verified
386     TPMT_SIGNATURE  *signature        // IN: signature to be verified
387 )
388 {
389     HMAC_STATE      hmacState;
390     TPM2B_DIGEST    digestToCompare;
391
392     // HMAC algorithm self testing code may be inserted here
393
394     digestToCompare.t.size = CryptStartHMAC2B(signature->signature.hmac.hashAlg,
395                                               &signKey->sensitive.sensitive.bits.b, &hmacState);
396
397     CryptUpdateDigest2B(&hmacState, &hashData->b);
398
399     CryptCompleteHMAC2B(&hmacState, &digestToCompare.b);
400
401     // Compare digest
402     if(MemoryEqual(digestToCompare.t.buffer,
403                  (BYTE *) &signature->signature.hmac.digest,
404                  digestToCompare.t.size))
405         return TPM_RC_SUCCESS;
406     else
407         return TPM_RC_SIGNATURE;

```

```
408
409 }
```

### 9.16.5.21 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Error Returns	Meaning
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```
410 static TPM_RC
411 CryptGenerateKeyedHash(
412     TPMT_PUBLIC *publicArea, // IN/OUT: the public area template
413                               // for the new key.
414     TPMS_SENSITIVE_CREATE *sensitiveCreate, // IN: sensitive creation data
415     TPMT_SENSITIVE *sensitive, // OUT: sensitive area
416     TPM_ALG_ID kdfHashAlg, // IN: algorithm for the KDF
417     TPM2B_SEED *seed, // IN: the seed
418     TPM2B_NAME *name // IN: name of the object
419 )
420 {
421     TPMT_KEYEDHASH_SCHEME *scheme;
422     TPM_ALG_ID hashAlg;
423     UINT16 hashBlockSize;
424
425     scheme = &publicArea->parameters.keyedHashDetail.scheme;
426
427     pAssert(publicArea->type == TPM_ALG_KEYEDHASH);
428
429     // Pick the limiting hash algorithm
430     if(scheme->scheme == TPM_ALG_NULL)
431         hashAlg = publicArea->nameAlg;
432     else if(scheme->scheme == TPM_ALG_XOR)
433         hashAlg = scheme->details.xor.hashAlg;
434     else
435         hashAlg = scheme->details.hmac.hashAlg;
436     hashBlockSize = CryptGetHashBlockSize(hashAlg);
437
438     // if this is a signing or a decryption key, then then the limit
439     // for the data size is the block size of the hash. This limit
440     // is set because larger values have lower entropy because of the
441     // HMAC function.
442     if( publicArea->objectAttributes.sensitiveDataOrigin == CLEAR
443        && ( publicArea->objectAttributes.decrypt
444            || publicArea->objectAttributes.sign)
445        && sensitiveCreate->data.t.size > hashBlockSize)
446
447         return TPM_RC_SIZE;
448
449     if(publicArea->objectAttributes.sensitiveDataOrigin == SET)
450     {
451         // Created block cannot be larger than the structure allows.
452         if(hashBlockSize > MAX_SYM_DATA)
453             hashBlockSize = MAX_SYM_DATA;
454
455         // Create new keyedHash object
456         sensitive->sensitive.bits.t.size = hashBlockSize;
457
458         CryptKDFa(kdfHashAlg,
459                 &seed->b,
460                 "sensitive", //This string is a vendor-
461                 //specific information
462                 &name->b, // computed from the public template
```

```

463         NULL,           // 32-bit ENDIAN counter.
464         sensitive->sensitive.bits.t.size * 8,
465         sensitive->sensitive.bits.t.buffer, NULL);
466     }
467     else
468     {
469         // Copy input data to sensitive area
470         MemoryCopy2B(&sensitive->sensitive.any.b, &sensitiveCreate->data.b);
471     }
472
473     // Compute obfuscation. Parent handle is not available and not needed for
474     // symmetric object at this point. TPM_RH_UNASSIGNED is passed at the
475     // place of parent handle
476     CryptComputeSymValue(TPM_RH_UNASSIGNED, publicArea, sensitive, seed,
477                         kdfHashAlg, name);
478
479     CryptComputeSymmetricUnique(publicArea->nameAlg,
480                                sensitive,
481                                &publicArea->unique.keyedHash);
482     return TPM_RC_SUCCESS;
483 }

```

### 9.16.5.22 CryptKDFa()

This function generates a key using the *KDFa()* formulation in Part 1 of the TPM specification. 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.

```

484 // #define CryptKDFa(hashAlg, key, label, contextU, contextV, \
485 //                 sizeInBits, keyStream, counterInOut) \
486 //     _cpri__KDFa( \
487 //         ((TPM_ALG_ID)hashAlg), \
488 //         ((TPM2B *)key), \
489 //         ((const char *)label), \
490 //         ((TPM2B *)contextU), \
491 //         ((TPM2B *)contextV), \
492 //         ((UINT32)sizeInBits), \
493 //         ((BYTE *)keyStream), \
494 //         ((UINT32 *)counterInOut), \
495 //         ((BOOL) FALSE) \
496 //     )
497 //

```

### 9.16.5.23 CryptKDFaOnce()

This function generates a key using the *KDFa()* formulation in Part 1 of the TPM specification. 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*.

```

498 // #define CryptKDFaOnce(hashAlg, key, label, contextU, contextV, \
499 //                      sizeInBits, keyStream, counterInOut) \
500 //     _cpri__KDFa( \
501 //         ((TPM_ALG_ID)hashAlg), \
502 //         ((TPM2B *)key), \
503 //         ((const char *)label), \
504 //         ((TPM2B *)contextU), \
505 //         ((TPM2B *)contextV), \
506 //         ((UINT32)sizeInBits), \
507 //         ((BYTE *)keyStream), \
508 //         ((UINT32 *)counterInOut), \

```

```

509 //%                ((BOOL) TRUE)                \
510 //%                )
511 //%

```

### 9.16.5.24 KDFa()

This function is used by functions outside of *CryptUtil()* to access *\_cpri\_KDFa()*.

```

512 void
513 KDFa(
514     TPM_ALG_ID      hash,           // IN: hash algorithm used in HMAC
515     TPM2B           *key,           // IN: HMAC key
516     const char      *label,         // IN: a null-terminated label for KDF
517     TPM2B           *contextU,      // IN: context U
518     TPM2B           *contextV,      // IN: context V
519     UINT32          sizeInBits,     // IN: size of generated key in bits
520     BYTE            *keyStream,      // OUT: key buffer
521     UINT32          *counterInOut    // IN/OUT: caller may provide the
522                                     // iteration counter for
523                                     // incremental operations to
524                                     // avoid large intermediate
525                                     // buffers.
526 )
527 {
528     CryptKDFa(hash, key, label, contextU, contextV, sizeInBits,
529              keyStream, counterInOut);
530 }

```

### 9.16.5.25 CryptKDFe()

This function generates a key using the *KDFa()* formulation in Part 1 of the TPM specification. In this implementation, this is a macro invocation of *\_cpri\_\_KDFe()* in the hash module of the *CryptoEngine()*.

```

531 //##define CryptKDFe(hashAlg, Z, label, partyUInfo, partyVInfo,           \
532 //%                sizeInBits, keyStream)                                  \
533 //% _cpri__KDFe(                                                           \
534 //%     ((TPM_ALG_ID)hashAlg),                                           \
535 //%     ((TPM2B *)Z),                                                     \
536 //%     ((const char *)label),                                           \
537 //%     ((TPM2B *)partyUInfo),                                           \
538 //%     ((TPM2B *)partyVInfo),                                           \
539 //%     ((UINT32)sizeInBits),                                             \
540 //%     ((BYTE *)keyStream)                                              \
541 //%     )
542 //%
543 #endif //TPM_ALG_KEYEDHASH // 1

```

## 9.16.6 RSA Functions

### 9.16.6.1 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.

```

544 #ifndef TPM_ALG_RSA // 2
545 static void
546 BuildRSA(
547     OBJECT      *rsaKey,
548     RSA_KEY     *key
549 )

```

```

550 {
551     key->exponent = rsaKey->publicArea.parameters.rsaDetail.exponent;
552     if(key->exponent == 0)
553         key->exponent = RSA_DEFAULT_PUBLIC_EXPONENT;
554     key->publicKey = &rsaKey->publicArea.unique.rsa.b;
555
556     if(rsaKey->attributes.publicOnly || rsaKey->privateExponent.t.size == 0)
557         key->privateKey = NULL;
558     else
559         key->privateKey = &(rsaKey->privateExponent.b);
560 }

```

### 9.16.6.2 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  $n \bmod p \neq 0$ , `TPM_RC_BINDING` is returned.

The key is validated by checking that a  $d$  can be found such that  $e d \bmod ((p-1)*(q-1)) = 1$ . If  $d$  is found that satisfies this requirement, it will be placed in  $d$ .

Error Returns	Meaning
TPM_RC_BINDING	the public and private portions of the key are not matched

```

561 TPM_RC
562 CryptTestKeyRSA(
563     TPM2B          *d,           // OUT: receives the private exponent
564     UINT32         e,           // IN: public exponent
565     TPM2B          *n,           // IN/OUT: public modulus
566     TPM2B          *p,           // IN: a first prime
567     TPM2B          *q,           // IN: an optional second prime
568 )
569 {
570     CRYPT_RESULT   retVal;
571
572     pAssert(d != NULL && n != NULL && p != NULL);
573     // Set the exponent
574     if(e == 0)
575         e = RSA_DEFAULT_PUBLIC_EXPONENT;
576     // CRYPT_PARAMETER
577     retVal = _cpri__TestKeyRSA(d, e, n, p, q);
578     if(retVal == CRYPT_SUCCESS)
579         return TPM_RC_SUCCESS;
580     else
581         return TPM_RC_BINDING; // convert CRYPT_PARAMETER
582 }

```

### 9.16.6.3 CryptGenerateKeyRSA()

This function is called to generate an RSA key from a provided seed. It calls `_cpri__GenerateKeyRSA()` to perform the computations.

Error Returns	Meaning
TPM_RC_CANCELLED	key generation has been cancelled
TPM_RC_VALUE	exponent is not prime or is less than 3; or could not find a prime using the provided parameters

```

583 static TPM_RC
584 CryptGenerateKeyRSA(

```



```

585     TPMT_PUBLIC      *publicArea,      // IN/OUT: The public area template for
586                                     // the new key. The public key
587                                     // area will be replaced by the
588                                     // product of two primes found by
589                                     // this function
590     TPMT_SENSITIVE   *sensitive,       // OUT: the sensitive area will be
591                                     // updated to contain the first
592                                     // prime and the symmetric
593                                     // encryption key
594     TPM_ALG_ID        hashAlg,         // IN: the hash algorithm for the KDF
595     TPM2B_SEED        *seed,          // IN: Seed for the creation
596     TPM2B_NAME        *name,          // IN: Object name
597     UINT32            *counter         // OUT: last iteration of the counter
598 )
599 {
600     CRYPT_RESULT      retVal;
601
602     *counter = 0;
603
604     // _cpri_GenerateKeyRSA can return CRYPT_CANCEL or CRYPT_FAIL
605     retVal = _cpri__GenerateKeyRSA(&publicArea->unique.rsa.b,
606                                   &sensitive->sensitive.rsa.b,
607                                   publicArea->parameters.rsaDetail.keyBits,
608                                   publicArea->parameters.rsaDetail.exponent,
609                                   hashAlg,
610                                   &seed->b,
611                                   "RSA key by vendor",
612                                   &name->b,
613                                   counter);
614     // CRYPT_CANCEL -> TPM_RC_CANCELLED; CRYPT_FAIL -> TPM_RC_VALUE
615     return TranslateCryptErrors(retVal);
616
617 }

```

#### 9.16.6.4 CryptLoadPrivateRSA()

This function is called to generate the private exponent of an RSA key. It uses *CryptTestKeyRSA()*.

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of <i>rsaKey</i> are not matched

```

618     TPM_RC
619     CryptLoadPrivateRSA(
620         OBJECT      *rsaKey      // IN: the RSA key object
621     )
622     {
623         TPM_RC      result;
624         TPMT_PUBLIC *publicArea = &rsaKey->publicArea;
625         TPMT_SENSITIVE *sensitive = &rsaKey->sensitive;
626
627         // Load key by computing the private exponent
628         // TPM_RC_BINDING
629         result = CryptTestKeyRSA(&(rsaKey->privateExponent.b),
630                                 publicArea->parameters.rsaDetail.exponent,
631                                 &(publicArea->unique.rsa.b),
632                                 &(sensitive->sensitive.rsa.b),
633                                 NULL);
634         if(result != TPM_RC_SUCCESS)
635             return result;
636         rsaKey->attributes.privateExp = SET;
637         return TPM_RC_SUCCESS;
638     }

```

### 9.16.6.5 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.

```

639 TPMT_RSA_DECRYPT*
640 CryptSelectRSAScheme(
641     TPMI_DH_OBJECT      rsaHandle,           // IN: handle of sign key
642     TPMT_RSA_DECRYPT    *scheme             // IN: a sign or decrypt scheme
643 )
644 {
645     OBJECT              *rsaObject;
646     TPMT_ASYM_SCHEME    *keyScheme;
647
648     // Get sign object pointer
649     rsaObject = ObjectGet(rsaHandle);
650     keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
651
652     // if the default scheme of the object is TPM_ALG_NULL, then select the
653     // input scheme
654     if(keyScheme->scheme == TPM_ALG_NULL)
655     {
656         return scheme;
657     }
658     // if the object scheme is not TPM_ALG_NULL and the input scheme is
659     // TPM_ALG_NULL, then select the default scheme of the object.
660     else if(scheme->scheme == TPM_ALG_NULL)
661         // if input scheme is NULL
662         return
663         (TPMT_RSA_DECRYPT *)keyScheme;
664     // get here if both the object scheme and the input scheme are
665     // not TPM_ALG_NULL. Need to insure that they are the same.
666     // IMPLEMENTATION NOTE: This could cause problems if future versions have
667     // schemes that have more values than just a hash algorithm. A new function
668     // (IsSchemeSame()) might be needed then.
669     else if( keyScheme->scheme == scheme->scheme
670             && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
671         return scheme;
672     else
673         // two different, incompatible schemes specified
674         return NULL;
675 }

```

### 9.16.6.6 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.

Error Returns	Meaning
TPM_RC_ATTRIBUTES	The key is not a decryption key.
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.

```

676 TPM_RC
677 CryptDecryptRSA(
678     UINT16          *dataOutSize,      // OUT: size of plain text in byte
679     BYTE            *dataOut,         // OUT: plain text
680     OBJECT          *rsaKey,         // IN: internal RSA key
681     TPMT_RSA_DECRYPT *scheme,        // IN: selects the padding scheme
682     UINT16          cipherInSize,    // IN: size of cipher text in byte
683     BYTE            *cipherIn,      // IN: cipher text
684     const char      *label          // IN: a label, when needed
685 )
686 {
687     RSA_KEY          key;
688     CRYPT_RESULT     retVal = CRYPT_SUCCESS;
689     UINT32           dSize;          // Place to put temporary value for the
690                                     // returned data size
691     TPMI_ALG_HASH    hashAlg = TPM_ALG_NULL; // hash algorithm in the selected
692                                     // padding scheme
693
694     // pointer checks
695     pAssert( (dataOutSize != NULL) && (dataOut != NULL)
696             && (rsaKey != NULL) && (cipherIn != NULL));
697
698     // The public type is a RSA object
699     pAssert(rsaKey->publicArea.type == TPM_ALG_RSA);
700
701     // Must have the private portion loaded. This check is made before this
702     // function is called.
703     pAssert(rsaKey->attributes.publicOnly == CLEAR);
704
705     if(rsaKey->publicArea.objectAttributes.decrypt != SET)
706         return TPM_RC_ATTRIBUTES;
707
708     // decryption requires that the private modulus be present
709     if(rsaKey->attributes.privateExp == CLEAR)
710     {
711         TPM_RC result;
712         // Load key by computing the private exponent
713         // CryptLoadPrivateRSA may return TPM_RC_BINDING
714         result = CryptLoadPrivateRSA(rsaKey);
715         if(result != TPM_RC_SUCCESS)
716             return result;
717     }
718
719     // the input buffer must be the size of the key
720     if(cipherInSize != rsaKey->publicArea.unique.rsa.t.size)
721         return TPM_RC_SIZE;
722
723     BuildRSA(rsaKey, &key);
724
725     // Initialize the dOutSize parameter
726     dSize = *dataOutSize;
727
728     // For OAEP scheme, initialize the hash algorithm for padding
729     if(scheme->scheme == TPM_ALG_OAEP)
730         hashAlg = scheme->details.oaep.hashAlg;

```

```

731 // _cpri__DecryptRSA may return CRYPT_PARAMETER CRYPT_FAIL CRYPT_SCHEME
732 retVal = _cpri__DecryptRSA(&dSize, dataOut, &key, scheme->scheme,
733                           cipherInSize, cipherIn, hashAlg, label);
734
735 // Scheme must have been validated when the key was loaded/imported
736 if(retVal == CRYPT_SCHEME) //++ this needs to be an assert
737     return TPM_RC_FAILURE;
738
739 // Set the return size
740 pAssert(dSize <= UINT16_MAX);
741 *dataOutSize = (UINT16)dSize;
742
743 // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_FAIL -> TPM_RC_VALUE
744 return TranslateCryptErrors(retVal);
745 }

```

### 9.16.6.7 CryptEncryptRSA()

This function provides the interface to `_cpri__EncryptRSA()`.

TPM_RC_ATTRIBUTES	<i>rsaKey</i> is not a valid decryption key
TPM_RC_SCHEME	<i>scheme</i> is not supported
TPM_RC_VALUE	numeric value of <i>dataIn</i> is greater than the key modulus

```

746 TPM_RC
747 CryptEncryptRSA(
748     UINT16          *cipherOutSize, // OUT: size of cipher text in byte
749     BYTE            *cipherOut,     // OUT: cipher text
750     OBJECT          *rsaKey,        // IN: internal RSA key
751     TPMT_RSA_DECRYPT *scheme,        // IN: selects the padding scheme
752     UINT16          dataInSize,     // IN: size of plain text in byte
753     BYTE            *dataIn,        // IN: plain text
754     const char      *label          // IN: an optional label
755 )
756 {
757     RSA_KEY          key;
758     CRYPT_RESULT     retVal;
759     UINT32           cOutSize; // Conversion variable
760     TPMI_ALG_HASH    hashAlg = TPM_ALG_NULL; // hash algorithm in selected
761                                     // padding scheme
762
763     // must have a pointer to a key and some data to encrypt
764     pAssert(rsaKey != NULL && dataIn != NULL);
765
766     // The public type is a RSA object
767     pAssert(rsaKey->publicArea.type == TPM_ALG_RSA);
768
769     // If the cipher buffer must be provided and it must be large enough
770     // for the result
771     pAssert( cipherOut != NULL
772             && cipherOutSize != NULL
773             && *cipherOutSize >= rsaKey->publicArea.unique.rsa.t.size);
774
775     if(rsaKey->publicArea.objectAttributes.decrypt != SET)
776         return TPM_RC_ATTRIBUTES;
777
778     // Only need the public key and exponent for encryption
779     BuildRSA(rsaKey, &key);
780
781     // Copy the size to the conversion buffer
782     cOutSize = *cipherOutSize;
783

```

```

784 // For OAEP scheme, initialize the hash algorithm for padding
785 if(scheme->scheme == TPM_ALG_OAEP)
786     hashAlg = scheme->details.oaep.hashAlg;
787
788 // Encrypt the data
789 // _cpri_EncryptRSA may return CRYPT_PARAMETER or CRYPT_SCHEME
790 retVal = _cpri_EncryptRSA(&cOutSize,cipherOut, &key, scheme->scheme,
791                          dataInSize, dataIn, hashAlg, label);
792
793 pAssert (cOutSize <= UINT16_MAX);
794 *cipherOutSize = (UINT16)cOutSize;
795 // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_SCHEME -> TPM_RC_SCHEME
796 return TranslateCryptErrors(retVal);
797 }

```

### 9.16.6.8 CryptSignRSA()

This function is used to sign a digest with an RSA signing key.

Error Returns	Meaning
TPM_RC_BINDING	public and private part of <i>signKey</i> are not properly bound
TPM_RC_SCHEME	<i>scheme</i> is not supported
TPM_RC_VALUE	<i>hashData</i> is larger than the modulus of <i>signKey</i> , or the size of <i>hashData</i> does not match hash algorithm in <i>scheme</i>

```

798 static TPM_RC
799 CryptSignRSA(
800     OBJECT                *signKey,           // IN: RSA key signs the hash
801     TPMT_SIG_SCHEME      *scheme,           // IN: sign scheme
802     TPM2B_DIGEST         *hashData,        // IN: hash to be signed
803     TPMT_SIGNATURE       *sig              // OUT: signature
804 )
805 {
806     UINT32                signSize;
807     RSA_KEY               key;
808     CRYPT_RESULT          retVal;
809
810     pAssert( (signKey != NULL) && (scheme != NULL)
811             && (hashData != NULL) && (sig != NULL));
812
813
814     // assume that the key has private part loaded and that it is a signing key.
815     pAssert( (signKey->attributes.publicOnly == CLEAR)
816             && (signKey->publicArea.objectAttributes.sign == SET));
817
818     // check if the private exponent has been computed
819     if(signKey->attributes.privateExp == CLEAR)
820     {
821         // need to compute the private exponent
822         TPM_RC result;
823         // May return TPM_RC_BINDING
824         result = CryptLoadPrivateRSA(signKey);
825         if(result != TPM_RC_SUCCESS)
826             return result;
827     }
828     BuildRSA(signKey, &key);
829
830     // initialize the common signature values
831     sig->sigAlg = scheme->scheme;
832     sig->signature.any.hashAlg = scheme->details.any.hashAlg;
833
834     // _crypti_SignRSA can return CRYPT_SCHEME and CRYPT_PARAMETER

```

```

835     retVal = _cpri__SignRSA(&signSize,
836                           sig->signature.rsassa.sig.t.buffer,
837                           &key,
838                           sig->sigAlg,
839                           sig->signature.any.hashAlg,
840                           hashData->t.size, hashData->t.buffer);
841     pAssert(signSize <= UINT16_MAX);
842     sig->signature.rsassa.sig.t.size = (UINT16)signSize;
843
844     // CRYPT_SCHEME -> TPM_RC_SCHEME; CRYPT_PARAMTER -> TPM_RC_VALUE
845     return TranslateCryptErrors(retVal);
846 }

```

### 9.16.6.9 CryptRSAVerifySignature()

This function is used to verify signature signed by a RSA key.

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not genuine
TPM_RC_SCHEME	signature scheme not supported

```

847 static TPM_RC
848 CryptRSAVerifySignature(
849     OBJECT                *signKey,           // IN: RSA key signed the hash
850     TPM2B_DIGEST          *hashData,         // IN: hash being signed
851     TPMT_SIGNATURE        *sig              // IN: signature to be verified
852 )
853 {
854     RSA_KEY                key;
855     CRYPT_RESULT           retVal;
856
857     // Validate parameter assumptions
858     pAssert((signKey != NULL) && (hashData != NULL) && (sig != NULL));
859
860     // This is a public-key-only operation
861     BuildRSA(signKey, &key);
862
863     // Call crypto engine to verify signature
864     // _cpri__ValidateSignaturRSA may return CRYPT_FAIL or CRYPT_SCHEME
865     retVal = _cpri__ValidateSignatureRSA(&key, sig->sigAlg,
866                                         sig->signature.any.hashAlg,
867                                         hashData->t.size,
868                                         hashData->t.buffer,
869                                         sig->signature.rsassa.sig.t.size,
870                                         sig->signature.rsassa.sig.t.buffer,
871                                         0);
872     // _cpri__ValidateSignatureRSA can return CRYPT_SUCCESS, CRYPT_FAIL, or
873     // CRYPT_SCHEME. Translate CRYPT_FAIL to TPM_RC_SIGNATURE
874     if(retVal == CRYPT_FAIL)
875         return TPM_RC_SIGNATURE;
876     //
877     // CRYPT_SCHEME -> TPM_RC_SCHEME
878     return TranslateCryptErrors(retVal);
879 }
880 #endif //TPM_ALG_RSA    //% 2

```

## 9.16.7 ECC Functions

### 9.16.7.1 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.

```

881 #ifdef TPM_ALG_ECC /*% 3
882 static const ECC_CURVE *
883 CryptEccGetCurveDataPointer(
884     TPM_ECC_CURVE    curveID           // IN: id of the curve
885 )
886 {
887     return _cpri__EccGetParametersByCurveId(curveID);
888 }

```

### 9.16.7.2 CryptEccGetKeySizeInBits()

This function returns the size in bits of the key associated with a curve.

```

889 UINT16
890 CryptEccGetKeySizeInBits(
891     TPM_ECC_CURVE    curveID           // IN: id of the curve
892 )
893 {
894     const ECC_CURVE    *curve = CryptEccGetCurveDataPointer(curveID);
895
896     if(curve == NULL)
897         return 0;
898     return curve->keySizeBits;
899 }

```

### 9.16.7.3 CryptEccGetKeySizeBytes()

This macro returns the size of the ECC key in bytes. It uses *CryptEccGetKeySizeInBits()*. The next lines will be placed in *CryptUtil\_fp.h* with the */\*%* removed

```

900 /*% #define CryptEccGetKeySizeInBytes(curve)          \
901 /*%          ((CryptEccGetKeySizeInBits(curve)+7)/8)

```

### 9.16.7.4 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}.

```

902 const TPM2B *
903 CryptEccGetParameter(
904     char                p,                // IN: the parameter selector
905     TPM_ECC_CURVE    curve                // IN: the curve id
906 )
907 {
908     const ECC_CURVE    *curveData = _cpri__EccGetParametersByCurveId(curve);
909
910     if(curveData == NULL)
911         return NULL;
912     switch (p)
913     {
914     case 'p':
915         return curveData->curveData->p;

```

```

916     case 'n':
917         return curveData->curveData->n;
918     case 'a':
919         return curveData->curveData->a;
920     case 'b':
921         return curveData->curveData->b;
922     case 'x':
923         return curveData->curveData->x;
924     case 'y':
925         return curveData->curveData->y;
926     case 'h':
927         return curveData->curveData->h;
928     default:
929         return NULL;
930     }
931 }

```

### 9.16.7.5 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```

932 const TPMT_ECC_SCHEME *
933 CryptGetCurveSignScheme(
934     TPM_ECC_CURVE    curveId           // IN: The curve selector
935 )
936 {
937     const ECC_CURVE    *curveData = _cpri__EccGetParametersByCurveId(curveId);
938
939     if(curveData == NULL)
940         return NULL;
941     return &(curveData->sign);
942 }

```

### 9.16.7.6 CryptEccIsPointOnCurve()

This function will validate that an ECC point is on the curve of given *curveID*.

Return Value	Meaning
TRUE	if the point is on curve
FALSE	if the point is not on curve

```

943 BOOL
944 CryptEccIsPointOnCurve(
945     TPM_ECC_CURVE    curveID,           // IN: ECC curve ID
946     TPMS_ECC_POINT    *Q                // IN: ECC point
947 )
948 {
949     // ECC algorithm self testing code may be inserted here
950
951     // Call crypto engine function to check if a ECC public point is on the
952     // given curve
953     if(_cpri__EccIsPointOnCurve(curveID, Q))
954         return TRUE;
955     else
956         return FALSE;
957 }

```

### 9.16.7.7 CryptNewEccKey()

This function creates a random ECC key that is not derived from other parameters as is a Primary Key.



```

958 TPM_RC
959 CryptNewEccKey(
960     TPM_ECC_CURVE      curveID,           // IN: ECC curve
961     TPMS_ECC_POINT     *publicPoint,     // OUT: public point
962     TPM2B_ECC_PARAMETER *sensitive       // OUT: private area
963 )
964 {
965     // _cpri_GetEphemeralECC may return CRYPT_PARAMETER
966     if(_cpri_GetEphemeralEcc(publicPoint, sensitive, curveID) != CRYPT_SUCCESS)
967         // Something is wrong with the key.
968         return TPM_RC_KEY;
969     else
970         return TPM_RC_SUCCESS;
971 }

```

### 9.16.7.8 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.

Error Returns	Meaning
TPM_RC_ECC_POINT	invalid optional ECC point $pIn$
TPM_RC_NO_RESULT	multiplication resulted in a point at infinity

```

972 TPM_RC
973 CryptEccPointMultiply(
974     TPMS_ECC_POINT     *pOut,           // OUT: output point
975     TPM_ECC_CURVE      curveId,        // IN: curve selector
976     TPM2B_ECC_PARAMETER *dIn,         // IN: public scalar
977     TPMS_ECC_POINT     *pIn           // IN: optional point
978 )
979 {
980     TPM2B_ECC_PARAMETER *n = NULL;
981     CRYPT_RESULT        retVal;
982
983     pAssert(pOut != NULL && dIn != NULL);
984
985     if(pIn != NULL)
986     {
987         n = dIn;
988         dIn = NULL;
989     }
990
991     // _cpri_EccPointMultiply may return CRYPT_POINT or CRYPT_NO_RESULT
992     retVal = _cpri_EccPointMultiply(pOut, curveId, dIn, pIn, n);
993
994     // CRYPT_POINT->TPM_RC_ECC_POINT and CRYPT_NO_RESULT->TPM_RC_NO_RESULT
995     return TranslateCryptErrors(retVal);
996 }

```

### 9.16.7.9 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.

Error Returns	Meaning
TPM_RC_VALUE	hash algorithm is not supported

```

997 static TPM_RC
998 CryptGenerateKeyECC(
999     TPMT_PUBLIC      *publicArea,          // IN/OUT: The public area template
1000                                     //         for the new key.
1001     TPMT_SENSITIVE   *sensitive,          // IN/OUT: the sensitive area
1002     TPM_ALG_ID       hashAlg,             // IN: algorithm for the KDF
1003     TPM2B_SEED       *seed,              // IN: the seed value
1004     TPM2B_NAME       *name,              // IN: the name of the object
1005     UINT32           *counter             // OUT: the iteration counter
1006 )
1007 {
1008     CRYPT_RESULT      retVal;
1009
1010     *counter = 0;
1011
1012     // _cpri_GenerateKeyEcc only has one error return (CRYPT_PARAMETER) which means
1013     // that the hash algorithm is not supported. This should not be possible
1014     retVal = _cpri_GenerateKeyEcc(&publicArea->unique.ecc,
1015                                 &sensitive->sensitive.ecc,
1016                                 publicArea->parameters.eccDetail.curveID,
1017                                 hashAlg, &seed->b, "ECC key by vendor",
1018                                 &name->b, counter);
1019
1020     // This will only be useful if _cpri_GenerateKeyEcc return CRYPT_CANCEL
1021     return TranslateCryptErrors(retVal);
1022 }

```

#### 9.16.7.10 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.

Error Returns	Meaning
TPM_RC_SCHEME	unsupported <i>scheme</i>
TPM_RC_VALUE	invalid commit status (in case of a split scheme) or failed to generate r value.

```

1022 static TPM_RC
1023 CryptSignECC(
1024     OBJECT           *signKey,           // IN: ECC key to sign the hash
1025     TPMT_SIG_SCHEME *scheme,           // IN: sign scheme
1026     TPM2B_DIGEST     *hashData,        // IN: hash to be signed
1027     TPMT_SIGNATURE   *signature        // OUT: signature
1028 )
1029 {
1030     TPM2B_ECC_PARAMETER r;
1031     TPM2B_ECC_PARAMETER *pr = NULL;
1032     CRYPT_RESULT        retVal;
1033
1034     if(CryptIsSplitSign(scheme->scheme))
1035     {
1036         // When this code was written, the only split scheme was ECDAA
1037         // (which can also be used for U-Prove).
1038         if(!CryptGenerateR(&r,
1039                             &scheme->details.ecdaa.count,
1040                             signKey->publicArea.parameters.eccDetail.curveID,
1041                             &signKey->name))
1042             return TPM_RC_VALUE;
1043         pr = &r;

```

```

1044     }
1045     // Call crypto engine function to sign
1046     // _cpri_SignEcc may return CRYPT_SCHEME
1047     retVal = _cpri_SignEcc(&signature->signature.ecdsa.signatureR,
1048                          &signature->signature.ecdsa.signatureS,
1049                          scheme->scheme,
1050                          scheme->details.any.hashAlg,
1051                          signKey->publicArea.parameters.eccDetail.curveID,
1052                          &signKey->sensitive.sensitive.ecc,
1053                          &hashData->b,
1054                          pr
1055                          );
1056     if(CryptIsSplitSign(scheme->scheme) && retVal == CRYPT_SUCCESS)
1057         CryptEndCommit(scheme->details.ecdaa.count);
1058     // CRYPT_SCHEME->TPM_RC_SCHEME
1059     return TranslateCryptErrors(retVal);
1060 }

```

### 9.16.7.11 CryptECCVerifySignature()

This function is used to verify a signature created with an ECC key.

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not valid
TPM_RC_SCHEME	the signing scheme or <i>hashAlg</i> is not supported

```

1061 static TPM_RC
1062 CryptECCVerifySignature(
1063     OBJECT          *signKey,           // IN: ECC key signed the hash
1064     TPM2B_DIGEST    *hashData,         // IN: hash being signed
1065     TPMT_SIGNATURE  *signature         // IN: signature to be verified
1066 )
1067 {
1068     CRYPT_RESULT     retVal;
1069     // This implementation uses the fact that all the defined ECC signing
1070     // schemes have the hash as the first parameter.
1071     // _cpriValidateSignatureEcc may return CRYPT_FAIL or CRYPT_SCHEME
1072     retVal = _cpri_ValidateSignatureEcc(&signature->signature.ecdsa.signatureR,
1073                                       &signature->signature.ecdsa.signaturesS,
1074                                       signature->sigAlg,
1075                                       signature->signature.any.hashAlg,
1076                                       signKey->publicArea.parameters.eccDetail.curveID,
1077                                       &signKey->publicArea.unique.ecc,
1078                                       &hashData->b);
1079     if(retVal == CRYPT_FAIL)
1080         return TPM_RC_SIGNATURE;
1081     // CRYPT_SCHEME->TPM_RC_SCHEME
1082     return TranslateCryptErrors(retVal);
1083 }

```

### 9.16.7.12 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.

Return Value	Meaning
TRUE	r value computed
FALSE	no r value computed

```

1084 BOOL
1085 CryptGenerateR(
1086     TPM2B_ECC_PARAMETER *r,           // OUT: the generated random value
1087     UINT16               *c,           // IN/OUT: count value.
1088     TPMI_ECC_CURVE       curveID,     // IN: the curve for the value
1089     TPM2B_NAME            *name        // IN: optional name of a key to
1090                                         // associate with 'r'
1091 )
1092 {
1093     // This holds the marshaled g_commitCounter.
1094     TPM2B_TYPE(8B, 8);
1095     TPM2B_8B          cntr = {8,{0}};
1096
1097     UINT32             iterations;
1098     const TPM2B        *n;
1099     UINT64             currentCount = gr.commitCounter;
1100
1101     n = CryptEccGetParameter('n', curveID);
1102     pAssert(r != NULL && n != NULL);
1103
1104     // If this is the commit phase, use the current value of the commit counter
1105     if(c != NULL)
1106     {
1107
1108         UINT16         t1;
1109         // if the array bit is not set, can't use the value.
1110         if(!BitIsSet((*c & COMMIT_INDEX_MASK), gr.commitArray,
1111                     sizeof(gr.commitArray)))
1112             return FALSE;
1113
1114         // If it is the sign phase, figure out what the counter value was
1115         // when the commitment was made.
1116         //
1117         // When gr.commitArray has less than 64K bits, the extra
1118         // bits of 'c' are used as a check to make sure that the
1119         // signing operation is not using an out of range count value
1120         t1 = (UINT16)currentCount;
1121
1122         // If the lower bits of c are greater or equal to the lower bits of t1
1123         // then the upper bits of t1 must be one more than the upper bits
1124         // of c
1125         if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
1126             // Since the counter is behind, reduce the current count
1127             currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
1128
1129         t1 = (UINT16)currentCount;
1130         if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
1131             return FALSE;
1132         // set the counter to the value that was
1133         // present when the commitment was made
1134         currentCount = (currentCount & 0xffffffff0000) | *c;
1135
1136     }
1137     // Marshal the count value to a TPM2B buffer for the KDF
1138     cntr.t.size = sizeof(currentCount);
1139     UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
1140
1141     // Now can do the KDF to create the random value for the signing operation
1142     // During the creation process, we may generate an r that does not meet the

```

```

1143     // requirements of the random value.
1144     // want to generate a new r.
1145
1146     r->t.size = n->size;
1147
1148     // Arbitrary upper limit on the number of times that we can look for
1149     // a suitable random value. The normally number of tries will be 1.
1150     for(iterations = 1; iterations < 1000000;)
1151     {
1152         BYTE    *pr = &r->b.buffer[0];
1153         int      i;
1154         CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, "ECDAA Commit",
1155                 name, &cntr.b, n->size * 8, r->t.buffer, &iterations);
1156
1157         // random value must be less than the prime
1158         if(CryptCompare(r->b.size, r->b.buffer, n->size, n->buffer) >= 0)
1159             continue;
1160
1161         // in this implementation it is required that at least bit
1162         // in the upper half of the number be set
1163         for(i = n->size/2; i > 0; i--)
1164             if(*pr++ != 0)
1165                 return TRUE;
1166     }
1167     return FALSE;
1168 }

```

#### 9.16.7.13 CryptCommit()

This function is called when the count value is committed. The *gr.commitArray* value associated with the current count value is SET and *g\_commitCounter* is incremented. The low-order 16 bits of old value of the counter is returned.

```

1169 UUINT16
1170 CryptCommit(
1171     void
1172 )
1173 {
1174     UUINT16    oldCount = (UUINT16)gr.commitCounter;
1175     gr.commitCounter++;
1176     BitSet(oldCount & COMMIT_INDEX_MASK, gr.commitArray, sizeof(gr.commitArray));
1177     return oldCount;
1178 }

```

#### 9.16.7.14 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the *gr.commitArray* bit associated with the count value so that it can't be used again.

```

1179 void
1180 CryptEndCommit(
1181     UUINT16    c           // IN: the counter value of the commitment
1182 )
1183 {
1184     BitClear((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
1185 }

```

#### 9.16.7.15 CryptCommitCompute()

This function performs the computations for the TPM2\_Commit() command. This could be a macro.

Error Returns	Meaning
TPM_RC_NO_RESULT	K, L, or E is the point at infinity
TPM_RC_CANCELLED	command was cancelled

```

1186 TPM_RC
1187 CryptCommitCompute (
1188     TPMS_ECC_POINT      *K,           // OUT: [d]B
1189     TPMS_ECC_POINT      *L,           // OUT: [r]B
1190     TPMS_ECC_POINT      *E,           // OUT: [r]M
1191     TPM_ECC_CURVE        curveID,     // IN: The curve for the computations
1192     TPMS_ECC_POINT      *M,           // IN: M (P1)
1193     TPMS_ECC_POINT      *B,           // IN: B (x2, y2)
1194     TPM2B_ECC_PARAMETER *d,           // IN: the private scalar
1195     TPM2B_ECC_PARAMETER *r           // IN: the computed r value
1196 )
1197 {
1198     // CRYPT_NO_RESULT->TPM_RC_NO_RESULT CRYPT_CANCEL->TPM_RC_CANCELLED
1199     return TranslateCryptErrors(
1200         _cpri__EccCommitCompute(K, L, E, curveID, M, B, d, r));
1201 }

```

#### 9.16.7.16 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve

Return Value	Meaning
TRUE	Get parameters success
FALSE	Unsupported ECC curve ID

```

1202 BOOL
1203 CryptEccGetParameters (
1204     TPM_ECC_CURVE        curveId,     // IN: ECC curve ID
1205     TPMS_ALGORITHM_DETAIL_ECC *parameters // OUT: ECC parameters
1206 )
1207 {
1208     const ECC_CURVE        *curve = _cpri__EccGetParametersByCurveId(curveId);
1209     const ECC_CURVE_DATA    *data;
1210
1211     if(curve == NULL)
1212         return FALSE;
1213
1214     data = curve->curveData;
1215
1216     parameters->curveID = curve->curveId;
1217
1218     // Key size in bit
1219     parameters->keySize = curve->keySizeBits;
1220
1221     // KDF
1222     parameters->kdf = curve->kdf;
1223
1224     // Sign
1225     parameters->sign = curve->sign;
1226
1227     // Copy p value
1228     MemoryCopy2B(&parameters->p.b, data->p);
1229
1230     // Copy a value
1231     MemoryCopy2B(&parameters->a.b, data->a);
1232 }

```

```

1233     // Copy b value
1234     MemoryCopy2B(&parameters->b.b, data->b);
1235
1236     // Copy Gx value
1237     MemoryCopy2B(&parameters->gX.b, data->x);
1238
1239     // Copy Gy value
1240     MemoryCopy2B(&parameters->gY.b, data->y);
1241
1242     // Copy n value
1243     MemoryCopy2B(&parameters->n.b, data->n);
1244
1245     // Copy h value
1246     MemoryCopy2B(&parameters->h.b, data->h);
1247
1248     return TRUE;
1249 }
1250 #if CC_ZGen_2Phase == YES

```

*CryptEcc2PhaseKeyExchange()* This is the interface to the key exchange function.

```

1251 TPM_RC
1252 CryptEcc2PhaseKeyExchange(
1253     TPMS_ECC_POINT      *outZ1,           // OUT: the computed point
1254     TPMS_ECC_POINT      *outZ2,           // OUT: optional second point
1255     TPM_ALG_ID           scheme,          // IN: the key exchange scheme
1256     TPM_ECC_CURVE        curveId,        // IN: the curve for the computations
1257     TPM2B_ECC_PARAMETER  *dsA,           // IN: static private TPM key
1258     TPM2B_ECC_PARAMETER  *deA,           // IN: ephemeral private TPM key
1259     TPMS_ECC_POINT      *QsB,           // IN: static public party B key
1260     TPMS_ECC_POINT      *QeB,           // IN: ephemeral public party B key
1261 )
1262 {
1263     return (TranslateCryptErrors(_cpri__C_2_2_KeyExchange(outZ1,
1264                                                            outZ2,
1265                                                            scheme,
1266                                                            curveId,
1267                                                            dsA,
1268                                                            deA,
1269                                                            QsB,
1270                                                            QeB)));
1271 }
1272 #endif // CC_ZGen_2Phase
1273 #endif //TPM_ALG_ECC  %% 3

```

#### 9.16.7.17 CryptIsSchemeAnonymous()

This function is used to test a scheme to see if it is an anonymous scheme. The only anonymous scheme is ECDSA. ECDSA can be used to do things like U-Prove.

```

1274 BOOL
1275 CryptIsSchemeAnonymous(
1276     TPM_ALG_ID           scheme           // IN: the scheme algorithm to test
1277 )
1278 {
1279     return ( 0
1280 #ifdef TPM_ALG_ECDSA
1281             || scheme == TPM_ALG_ECDSA
1282 #endif
1283             );
1284 }

```

## 9.16.8 Symmetric Functions

### 9.16.8.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

Error Returns	Meaning
TPM_RC_SYMMETRIC	unsupported symmetric algorithm

```

1285 static TPM_RC
1286 ParmDecryptSym(
1287     TPM_ALG_ID      symAlg,           // IN: the symmetric algorithm
1288     TPM_ALG_ID      hash,            // IN: hash algorithm for KDFa
1289     UINT16           keySizeInBits,   // IN: key key size in bits
1290     TPM2B            *key,           // IN: KDF HMAC key
1291     TPM2B            *nonceCaller,   // IN: nonce caller
1292     TPM2B            *nonceTpm,     // IN: nonce TPM
1293     UINT32           dataSize,       // IN: size of parameter buffer
1294     BYTE             *data           // OUT: buffer to be decrypted
1295 )
1296 {
1297     // KDF output buffer
1298     // It contains parameters for the CFB encryption
1299     // From MSB to LSB, they are the key and iv
1300     BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
1301     // Symmetric key size in byte
1302     UINT16           keySize = (keySizeInBits + 7) / 8;
1303     TPM2B_IV         iv;
1304
1305     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
1306     if(iv.t.size == 0)
1307         return TPM_RC_SYMMETRIC;
1308     // Generate key and iv
1309     CryptKDFa(hash, key, "CFB", nonceCaller, nonceTpm,
1310             keySizeInBits + (iv.t.size * 8), symParmString, NULL);
1311     MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
1312
1313     CryptSymmetricDecrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB, symParmString,
1314             &iv, dataSize, data);
1315     return TPM_RC_SUCCESS;
1316 }

```

### 9.16.8.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

Error Returns	Meaning
TPM_RC_SYMMETRIC	unsupported symmetric algorithm

```

1317 static TPM_RC
1318 ParmEncryptSym(
1319     TPM_ALG_ID      symAlg,           // IN: symmetric algorithm
1320     TPM_ALG_ID      hash,            // IN: hash algorithm for KDFa
1321     UINT16           keySizeInBits,   // IN: AES key size in bits
1322     TPM2B            *key,           // IN: KDF HMAC key
1323     TPM2B            *nonceCaller,   // IN: nonce caller
1324     TPM2B            *nonceTpm,     // IN: nonce TPM
1325     UINT32           dataSize,       // IN: size of parameter buffer
1326     BYTE             *data           // OUT: buffer to be encrypted
1327 )

```



```

1328 {
1329     // KDF output buffer
1330     // It contains parameters for the CFB encryption
1331     BYTE          symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
1332
1333     // Symmetric key size in bytes
1334     UINT16       keySize = (keySizeInBits + 7) / 8;
1335
1336     TPM2B_IV     iv;
1337
1338     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
1339     if(iv.t.size == 0)
1340         return TPM_RC_SYMMETRIC;
1341
1342     // Generate key and iv
1343     CryptKDFa(hash, key, "CFB", nonceTpm, nonceCaller,
1344             keySizeInBits + (iv.t.size * 8), symParmString, NULL);
1345
1346     MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
1347
1348     CryptSymmetricEncrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB, symParmString,
1349             &iv, dataSize, data);
1350     return TPM_RC_SUCCESS;
1351 }

```

### 9.16.8.3 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

```

1352 static TPM_RC
1353 CryptGenerateKeySymmetric(
1354     TPMT_PUBLIC          *publicArea,           // IN/OUT: The public area template
1355                                     //         for the new key.
1356     TPMS_SENSITIVE_CREATE *sensitiveCreate,    // IN: sensitive creation data
1357     TPMT_SENSITIVE        *sensitive,         // OUT: sensitive area
1358     TPM_ALG_ID            hashAlg,            // IN: hash algorithm for the KDF
1359     TPM2B_SEED            *seed,             // IN: seed used in creation
1360     TPM2B_NAME            *name              // IN: name of the object
1361 )
1362 {
1363     // If this is not a new key, then the provided key data must be the right size
1364     if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR
1365         && (sensitiveCreate->data.t.size * 8) !=
1366         publicArea->parameters.symDetail.keyBits.sym)
1367         return TPM_RC_KEY_SIZE;
1368
1369     // Make sure that the key size is OK.
1370     // This implementation only supports symmetric key sizes that are
1371     // multiples of 8
1372     if(publicArea->parameters.symDetail.keyBits.sym % 8 != 0)
1373         return TPM_RC_KEY_SIZE;
1374
1375     if(publicArea->objectAttributes.sensitiveDataOrigin == SET)
1376     {
1377         // Create new symmetric key
1378         sensitive->sensitive.sym.t.size =
1379             (publicArea->parameters.symDetail.keyBits.sym + 7) / 8;
1380
1381         CryptKDFa(hashAlg, &seed->b, "sensitive", &name->b,
1382             NULL, publicArea->parameters.symDetail.keyBits.sym,

```

```

1383         sensitive->sensitive.sym.t.buffer, NULL);
1384     }
1385     else
1386     {
1387         // Copy input symmetric key to sensitive area if the size is right
1388         MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b);
1389     }
1390
1391     // Compute obfuscation. Parent handle is not available and not needed for
1392     // symmetric object at this point. TPM_RH_UNASSIGNED is passed at the
1393     // place of parent handle
1394     CryptComputeSymValue(TPM_RH_UNASSIGNED, publicArea, sensitive, seed,
1395                         hashAlg, name);
1396
1397     // Create unique area in public
1398     CryptComputeSymmetricUnique(publicArea->nameAlg,
1399                               sensitive, &publicArea->unique.sym);
1400
1401     return TPM_RC_SUCCESS;
1402 }

```

#### 9.16.8.4 CryptXORObfuscation()

This function implements XOR obfuscation. It should not be called if the hash algorithm is not implemented. The only return value from this function is TPM\_RC\_SUCCESS.

```

1403 #ifdef TPM_ALG_KEYEDHASH /*% 5
1404 static TPM_RC
1405 CryptXORObfuscation(
1406     TPM_ALG_ID      hash,           // IN: hash algorithm for KDF
1407     TPM2B           *key,           // IN: KDF key
1408     TPM2B           *contextU,      // IN: contextU
1409     TPM2B           *contextV,      // IN: contextV
1410     UINT32          dataSize,       // IN: size of data buffer
1411     BYTE            *data           // IN/OUT: data to be XORed in place
1412 )
1413 {
1414     BYTE            mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
1415     BYTE            *pm;
1416     UINT32          i;
1417     UINT32          counter = 0;
1418     UINT16          hLen = CryptGetHashDigestSize(hash);
1419     UINT32          requestSize = dataSize * 8;
1420     INT32           remainBytes = (INT32) dataSize;
1421
1422     pAssert((key != NULL) && (data != NULL) && (hLen != 0));
1423
1424     // Call KDFa to generate XOR mask
1425     for(; remainBytes > 0; remainBytes -= hLen)
1426     {
1427         // Make a call to KDFa to get next iteration
1428         CryptKDFaOnce(hash, key, "XOR", contextU, contextV,
1429                     requestSize, mask, &counter);
1430
1431         // XOR next piece of the data
1432         pm = mask;
1433         for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
1434             *data++ ^= *pm++;
1435     }
1436     return TPM_RC_SUCCESS;
1437 }
1438 #endif //TPM_ALG_KEYED_HASH /*%5

```

## 9.16.9 Initialization and shut down

### 9.16.9.1 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.

```

1439 void
1440 CryptInitUnits(void)
1441 {
1442     // Call crypto engine unit initialization
1443     // We assume crypt engine initialization should always succeed. Otherwise,
1444     // TPM should go to failure mode.
1445
1446     // This is used to make sure that the correct version of CryptoEngine
1447     // has been linked
1448     _cpri__InitCryptoUnits();
1449     return;
1450 }

```

### 9.16.9.2 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.

```

1451 void
1452 CryptStopUnits(void)
1453 {
1454     // Call crypto engine unit stopping
1455     _cpri__StopCryptoUnits();
1456
1457     return;
1458 }

```

### 9.16.9.3 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.

```

1459 BOOL
1460 CryptUtilStartup(
1461     STARTUP_TYPE          type          // IN: the startup type
1462 )
1463 {
1464     // Make sure that the crypto library functions are ready
1465     if( !_cpri__Startup())
1466         return FALSE;
1467
1468     if(type == SU_RESET)
1469     {
1470 #ifdef TPM_ALG_ECDSA

```

```

1471
1472     // Get a new random commit nonce
1473     gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
1474     _cpri_GenerateRandom(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
1475     // Reset the counter and commit array
1476     gr.commitCounter = 0;
1477     MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
1478 #endif // TPM_ALG_ECDSA
1479     }
1480
1481     // If the shutdown was orderly, then the values recovered from NV will
1482     // be OK to use. If the shutdown was not orderly, then a TPM Reset was required
1483     // and we would have initialized in the code above.
1484
1485     return TRUE;
1486 }

```

## 9.16.10 Algorithm-Independent Functions

### 9.16.10.1 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.

### 9.16.10.2 CryptIsAsymAlgorithm()

This function indicates if an algorithm is an asymmetric algorithm.

Return Value	Meaning
TRUE	if it is an asymmetric algorithm
FALSE	if it is not an asymmetric algorithm

```

1487 BOOL
1488 CryptIsAsymAlgorithm(
1489     TPM_ALG_ID      algID           // IN: algorithm ID
1490 )
1491 {
1492     return ( 0
1493 #ifdef TPM_ALG_RSA
1494     || algID == TPM_ALG_RSA
1495 #endif
1496 #ifdef TPM_ALG_ECC
1497     || algID == TPM_ALG_ECC
1498 #endif
1499     );
1500 }

```

### 9.16.10.3 CryptGetSymmetricBlockSize()

This function returns the size in octets of the symmetric encryption block used by an algorithm and key size combination.

```

1501 INT16
1502 CryptGetSymmetricBlockSize(
1503     TPMI_ALG_SYM      algorithm,    // IN: symmetric algorithm
1504     UINT16            keySize       // IN: key size in bit
1505 )
1506 {

```

```

1507     return _cpri_GetSymmetricBlockSize(algorithm, keySize);
1508 }

```

#### 9.16.10.4 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.

```

1509 void
1510 CryptSymmetricEncrypt(
1511     BYTE                *encrypted,           // OUT: the encrypted data
1512     TPM_ALG_ID          algorithm,           // IN: algorithm for encryption
1513     UINT16              keySizeInBits,       // IN: key size in bits
1514     TPMI_ALG_SYM_MODE   mode,               // IN: symmetric encryption mode
1515     BYTE                *key,                // IN: encryption key
1516     TPM2B_IV            *ivIn,              // IN/OUT: Input IV and output chaining
1517                                     // value for the next block
1518     UINT32              dataSize,           // IN: data size in byte
1519     BYTE*               data                 // IN/OUT: data buffer
1520 )
1521 {
1522     BYTE                *iv = NULL;
1523     BYTE                defaultIV[sizeof(TPMT_HA)];
1524
1525     pAssert( ((mode == TPM_ALG_ECB) && (ivIn->t.size == 0))
1526             || (mode != TPM_ALG_ECB));
1527
1528     if(0
1529 #ifdef TPM_ALG_AES
1530         || algorithm == TPM_ALG_AES
1531 #endif
1532 #ifdef TPM_ALG_SM4
1533         || algorithm == TPM_ALG_SM4
1534 #endif
1535     )
1536     {
1537         // Both SM4 and AES have block size of 128 bits
1538         // If the iv is not provided, create a default of 0
1539         if(ivIn == NULL)
1540         {
1541             // Initialize the default IV
1542             iv = defaultIV;
1543             MemorySet(defaultIV, 0, 16);
1544         }
1545         else
1546         {
1547             // A provided IV has to be the right size
1548             pAssert(mode == TPM_ALG_ECB || ivIn->t.size == 16);
1549             iv = &(ivIn->t.buffer[0]);
1550         }
1551     }
1552     switch(algorithm)
1553     {
1554 #ifdef TPM_ALG_AES
1555     case TPM_ALG_AES:
1556     {
1557         switch (mode)
1558         {
1559             case TPM_ALG_CTR:
1560                 _cpri_AESEncryptCTR(encrypted, keySizeInBits, key, iv,
1561                                     dataSize, data);
1562                 break;
1563             case TPM_ALG_OFB:
1564                 _cpri_AESEncryptOFB(encrypted, keySizeInBits, key, iv,

```

```
1565                                     dataSize, data);
1566                                     break;
1567     case TPM_ALG_CBC:
1568         _cpri__AESEncryptCBC(encrypted, keySizeInBits, key, iv,
1569                             dataSize, data);
1570                                     break;
1571     case TPM_ALG_CFB:
1572         _cpri__AESEncryptCFB(encrypted, keySizeInBits, key, iv,
1573                             dataSize, data);
1574                                     break;
1575     case TPM_ALG_ECB:
1576         _cpri__AESEncryptECB(encrypted, keySizeInBits, key,
1577                             dataSize, data);
1578                                     break;
1579     default:
1580         pAssert(0);
1581     }
1582 }
1583 break;
1584 #endif
1585 #ifdef TPM_ALG_SM4
1586     case TPM_ALG_SM4:
1587     {
1588         switch (mode)
1589         {
1590             case TPM_ALG_CTR:
1591                 _cpri__SM4EncryptCTR(encrypted, keySizeInBits, key, iv,
1592                                     dataSize, data);
1593                 break;
1594             case TPM_ALG_OFB:
1595                 _cpri__SM4EncryptOFB(encrypted, keySizeInBits, key, iv,
1596                                     dataSize, data);
1597                 break;
1598             case TPM_ALG_CBC:
1599                 _cpri__SM4EncryptCBC(encrypted, keySizeInBits, key, iv,
1600                                     dataSize, data);
1601                 break;
1602
1603             case TPM_ALG_CFB:
1604                 _cpri__SM4EncryptCFB(encrypted, keySizeInBits, key, iv,
1605                                     dataSize, data);
1606                 break;
1607             case TPM_ALG_ECB:
1608                 _cpri__SM4EncryptECB(encrypted, keySizeInBits, key,
1609                                     dataSize, data);
1610                 break;
1611             default:
1612                 pAssert(0);
1613         }
1614     }
1615 #endif
1616     default:
1617         pAssert(FALSE);
1618         break;
1619 }
1620
1621 return;
1622
1623 }
1624 }
```

### 9.16.10.5 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.

```

1625 void
1626 CryptSymmetricDecrypt(
1627     BYTE          *decrypted,
1628     TPM_ALG_ID    algorithm,           // IN: algorithm for encryption
1629     UINT16        keySizeInBits,      // IN: key size in bits
1630     TPMI_ALG_SYM_MODE mode,           // IN: symmetric encryption mode
1631     BYTE          *key,               // IN: encryption key
1632     TPM2B_IV      *ivIn,              // IN/OUT: IV for next block
1633     UINT32        dataSize,           // IN: data size in byte
1634     BYTE*         data                 // IN/OUT: data buffer
1635 )
1636 {
1637     BYTE          *iv = NULL;
1638     BYTE          defaultIV[sizeof(TPMT_HA)];
1639
1640     if(0
1641 #ifdef TPM_ALG_AES
1642         || algorithm == TPM_ALG_AES
1643 #endif
1644 #ifdef TPM_ALG_SM4
1645         || algorithm == TPM_ALG_SM4
1646 #endif
1647     )
1648     {
1649         // Both SM4 and AES have block size of 128 bits
1650         // If the iv is not provided, create a default of 0
1651         if(ivIn == NULL)
1652         {
1653             // Initialize the default IV
1654             iv = defaultIV;
1655             MemorySet(defaultIV, 0, 16);
1656         }
1657         else
1658         {
1659             // A provided IV has to be the right size
1660             pAssert(mode == TPM_ALG_ECB || ivIn->t.size == 16);
1661             iv = &(ivIn->t.buffer[0]);
1662         }
1663     }
1664
1665     switch(algorithm)
1666     {
1667 #ifdef TPM_ALG_AES
1668     case TPM_ALG_AES:
1669     {
1670         switch (mode)
1671         {
1672             case TPM_ALG_CTR:
1673                 _cpri__AESDecryptCTR(decrypted, keySizeInBits, key, iv,
1674                                     dataSize, data);
1675                 break;
1676             case TPM_ALG_OFB:
1677                 _cpri__AESDecryptOFB(decrypted, keySizeInBits, key, iv,
1678                                     dataSize, data);
1679                 break;
1680             case TPM_ALG_CBC:
1681                 _cpri__AESDecryptCBC(decrypted, keySizeInBits, key, iv,
1682                                     dataSize, data);
1683                 break;
1684 
```

```

1685         case TPM_ALG_CFB:
1686             _cpri__AESDecryptCFB(decrypted, keySizeInBits, key, iv,
1687                                 dataSize, data);
1688             break;
1689         case TPM_ALG_ECB:
1690             _cpri__AESDecryptECB(decrypted, keySizeInBits, key,
1691                                 dataSize, data);
1692             break;
1693         default:
1694             pAssert(0);
1695     }
1696     break;
1697 }
1698 #endif //TPM_ALG_AES
1699 #ifdef TPM_ALG_SM4
1700     case TPM_ALG_SM4 :
1701         switch (mode)
1702         {
1703             case TPM_ALG_CTR:
1704                 _cpri__SM4DecryptCTR(decrypted, keySizeInBits, key, iv,
1705                                     dataSize, data);
1706                 break;
1707             case TPM_ALG_OFB:
1708                 _cpri__SM4DecryptOFB(decrypted, keySizeInBits, key, iv,
1709                                     dataSize, data);
1710                 break;
1711             case TPM_ALG_CBC:
1712                 _cpri__SM4DecryptCBC(decrypted, keySizeInBits, key, iv,
1713                                     dataSize, data);
1714                 break;
1715             case TPM_ALG_CFB:
1716                 _cpri__SM4DecryptCFB(decrypted, keySizeInBits, key, iv,
1717                                     dataSize, data);
1718                 break;
1719             case TPM_ALG_ECB:
1720                 _cpri__SM4DecryptECB(decrypted, keySizeInBits, key,
1721                                     dataSize, data);
1722                 break;
1723             default:
1724                 pAssert(0);
1725         }
1726         break;
1727 #endif //TPM_ALG_SM4
1728
1729     default:
1730         pAssert(FALSE);
1731         break;
1732 }
1733 return;
1734 }

```

#### 9.16.10.6 CryptSecretEncrypt()

This function creates a secret value and its associated secret structure using an asymmetric algorithm.

This function is used by TPM2\_MakeCredential().



Error Returns	Meaning
TPM_RC_ATTRIBUTES	<i>keyHandle</i> does not reference a valid decryption key
TPM_RC_KEY	invalid ECC key (public point is not on the curve)
TPM_RC_SCHEME	RSA key with an unsupported padding scheme
TPM_RC_VALUE	numeric value of the data to be decrypted is greater than the RSA key modulus

```

1735 TPM_RC
1736 CryptSecretEncrypt(
1737     TPMI_DH_OBJECT      keyHandle, // IN: encryption key handle
1738     const char          *label,     // IN: a null-terminated string as L
1739     TPM2B_DATA          *data,      // OUT: secret value
1740     TPM2B_ENCRYPTED_SECRET *secret  // OUT: secret structure
1741 )
1742 {
1743     TPM_RC      result = TPM_RC_SUCCESS;
1744     OBJECT      *encryptKey = ObjectGet(keyHandle); // TPM key used for encrypt
1745
1746     pAssert(data != NULL && secret != NULL);
1747
1748     // The output secret value has the size of the digest produced by the nameAlg.
1749     data->t.size = CryptGetHashDigestSize(encryptKey->publicArea.nameAlg);
1750
1751     pAssert(encryptKey->publicArea.objectAttributes.decrypt == SET);
1752
1753     switch(encryptKey->publicArea.type)
1754     {
1755 #ifdef TPM_ALG_RSA
1756     case TPM_ALG_RSA:
1757     {
1758         TPMT_RSA_DECRYPT      scheme;
1759
1760         // Use OAEP scheme
1761         scheme.scheme = TPM_ALG_OAEP;
1762         scheme.details.oaep.hashAlg = encryptKey->publicArea.nameAlg;
1763
1764         // Create secret data from RNG
1765         CryptGenerateRandom(data->t.size, data->t.buffer);
1766
1767         // Encrypt the data by RSA OAEP into encrypted secret
1768         result = CryptEncryptRSA(&secret->t.size, secret->t.secret,
1769                                 encryptKey, &scheme,
1770                                 data->t.size, data->t.buffer, label);
1771         if(result != TPM_RC_SUCCESS)
1772             return result;
1773     }
1774     break;
1775 #endif //TPM_ALG_RSA
1776
1777 #ifdef TPM_ALG_ECC
1778     case TPM_ALG_ECC:
1779     {
1780         TPMS_ECC_POINT      eccPublic;
1781         TPM2B_ECC_PARAMETER eccPrivate;
1782         TPMS_ECC_POINT      eccSecret;
1783         BYTE                *buffer = secret->t.secret;
1784
1785         // Need to make sure that the public point of the key is on the
1786         // curve defined by the key.
1787         if(!_cpri__EccIsPointOnCurve(
1788             encryptKey->publicArea.parameters.eccDetail.curveID,
1789             &encryptKey->publicArea.unique.ecc))

```

```

1790         return TPM_RC_KEY;
1791
1792     // Call crypto engine to create an auxiliary ECC key
1793     // We assume crypt engine initialization should always success.
1794     // Otherwise, TPM should go to failure mode.
1795     CryptNewEccKey(encryptKey->publicArea.parameters.eccDetail.curveID,
1796                  &eccPublic, &eccPrivate);
1797
1798     // Marshal ECC public to secret structure. This will be used by the
1799     // recipient to decrypt the secret with their private key.
1800     secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
1801
1802     // Compute ECDH shared secret which is R = [d]Q where d is the private
1803     // part of the ephemeral key and Q is the public part of a TPM key.
1804     // TPM_RC_KEY error return from CryptComputeECDHSecret because the
1805     // auxiliary ECC key is just created according to the parameters of
1806     // input ECC encrypt key.
1807     if( CryptEccPointMultiply(&eccSecret,
1808                             encryptKey->publicArea.parameters.eccDetail.curveID,
1809                             &eccPrivate,
1810                             &encryptKey->publicArea.unique.ecc)
1811        != CRYPT_SUCCESS)
1812         return TPM_RC_KEY;
1813
1814
1815     // The secret value is computed from Z using KDFe as:
1816     // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
1817     // Where:
1818     // HashID the nameAlg of the decrypt key
1819     // Z the x coordinate (Px) of the product (P) of the point (Q) of
1820     // the secret and the private x coordinate (de,V) of the
1821     // decryption key
1822     // Use a null-terminated string containing "SECRET"
1823     // PartyUInfo the x coordinate of the point in the secret (Qe,U )
1824     // PartyVInfo the x coordinate of the public key (Qs,V )
1825     // bits the number of bits in the digest of HashID
1826     // Retrieve seed from KDFe
1827
1828     CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
1829              &eccPublic.x.b, &encryptKey->publicArea.unique.ecc.x.b,
1830              data->t.size * 8, data->t.buffer);
1831 }
1832 break;
1833 #endif //TPM_ALG_ECC
1834
1835 default:
1836     FAIL(FATAL_ERROR_INTERNAL);
1837     break;
1838 }
1839
1840 return TPM_RC_SUCCESS;
1841 }

```

#### 9.16.10.7 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.

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_KEY	key of unsupported type
TPM_RC_NO_RESULT	multiplication resulted in ECC point at infinity
TPM_RC_SIZE	data to decrypt is not of the same size as RSA key
TPM_RC_VALUE	For RSA key, numeric value of the encrypted data is greater than the modulus, or the recovered data is larger than the output buffer. For <i>keyedHash</i> or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.
TPM_RC_FAILURE	internal error

```

1842 TPM_RC
1843 CryptSecretDecrypt(
1844     TPM_HANDLE          tpmKey,           // IN: decrypt key
1845     TPM2B_NONCE         *nonceCaller,    // IN: nonceCaller. It is needed for
1846                                     // symmetric decryption. For
1847                                     // asymmetric decryption, this
1848                                     // parameter is NULL
1849     const char          *label,          // IN: a null-terminated string as L
1850     TPM2B_ENCRYPTED_SECRET *secret,      // IN: input secret
1851     TPM2B_DATA          *data           // OUT: decrypted secret value
1852 )
1853 {
1854     TPM_RC      result = TPM_RC_SUCCESS;
1855     OBJECT      *decryptKey = ObjectGet(tpmKey); //TPM key used for decrypting
1856
1857     // Decryption for secret
1858     switch(decryptKey->publicArea.type)
1859     {
1860
1861     #ifndef TPM_ALG_RSA
1862     case TPM_ALG_RSA:
1863     {
1864         TPMT_RSA_DECRYPT      scheme;
1865
1866         // Use OAEP scheme
1867         scheme.scheme = TPM_ALG_OAEP;
1868         scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
1869
1870         // Set the output buffer capacity
1871         data->t.size = sizeof(data->t.buffer);
1872
1873         // Decrypt seed by RSA OAEP
1874         result = CryptDecryptRSA(&data->t.size, data->t.buffer, decryptKey,
1875                                 &scheme,
1876                                 secret->t.size, secret->t.secret, label);
1877         if( result == TPM_RC_SUCCESS
1878             && data->t.size > CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
1879             return TPM_RC_VALUE;
1880         return result;
1881     }
1882     break;
1883     #endif //TPM_ALG_RSA
1884
1885     #ifndef TPM_ALG_ECC

```

```

1886     case TPM_ALG_ECC:
1887     {
1888         TPMS_ECC_POINT      eccPublic;
1889         TPMS_ECC_POINT      eccSecret;
1890         BYTE                 *buffer = secret->t.secret;
1891         INT32                size = secret->t.size;
1892
1893         // Retrieve ECC point from secret buffer
1894         result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
1895         if(result != TPM_RC_SUCCESS)
1896             return result;
1897
1898         result = CryptEccPointMultiply(&eccSecret,
1899                                     decryptKey->publicArea.parameters.eccDetail.curveID,
1900                                     &decryptKey->sensitive.sensitive.ecc,
1901                                     &eccPublic);
1902
1903         if(result != TPM_RC_SUCCESS)
1904             return result;
1905
1906         // Set the size of the "recovered" secret value to be the size of the digest
1907         // produced by the nameAlg.
1908         data->t.size = CryptGetHashDigestSize(decryptKey->publicArea.nameAlg);
1909
1910         // The secret value is computed from Z using KDFe as:
1911         // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
1912         // Where:
1913         // HashID  the nameAlg of the decrypt key
1914         // Z       the x coordinate (Px) of the product (P) of the point (Q) of
1915         //         the secret and the private x coordinate (de,V) of the
1916         //         decryption key
1917         // Use a null-terminated string containing "SECRET"
1918         // PartyUInfo the x coordinate of the point in the secret (Qe,U )
1919         // PartyVInfo the x coordinate of the public key (Qs,V )
1920         // bits      the number of bits in the digest of HashID
1921         // Retrieve seed from KDFe
1922         CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
1923                 &eccPublic.x.b,
1924                 &decryptKey->publicArea.unique.ecc.x.b,
1925                 data->t.size * 8, data->t.buffer);
1926     }
1927     break;
1928 #endif //TPM_ALG_ECC
1929
1930     case TPM_ALG_KEYEDHASH:
1931
1932         // The seed size can not be bigger than the digest size of nameAlg
1933         if(secret->t.size >
1934            CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
1935             return TPM_RC_VALUE;
1936
1937         // Retrieve seed by XOR Obfuscation:
1938         // seed = XOR(secret, hash, key, nonceCaller, nullNonce)
1939         // where:
1940         // secret  the secret parameter from the TPM2_StartAuthHMAC command
1941         //          which contains the seed value
1942         // hash    nameAlg of tpmKey
1943         // key     the key or data value in the object referenced by
1944         //          entityHandle in the TPM2_StartAuthHMAC command
1945         // nonceCaller the parameter from the TPM2_StartAuthHMAC command
1946         // nullNonce a zero-length nonce
1947         {
1948             // XOR Obfuscation in place
1949             CryptXORObfuscation(decryptKey->publicArea.nameAlg,
1950                               &decryptKey->sensitive.sensitive.bits.b,
1951                               &nonceCaller->b, NULL,

```

```

1952         secret->t.size, secret->t.secret);
1953
1954         // Copy decrypted seed
1955         MemoryCopy2B(&data->b, &secret->b);
1956     }
1957     break;
1958 case TPM_ALG_SYMCIPHER:
1959 {
1960     TPM2B_IV          iv = {0};
1961     TPMT_SYM_DEF_OBJECT *symDef;
1962
1963     // The seed size can not be bigger than the digest size of nameAlg
1964     if(secret->t.size >
1965         CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
1966         return TPM_RC_VALUE;
1967
1968     symDef = &decryptKey->publicArea.parameters.symDetail;
1969
1970     iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
1971                                             symDef->keyBits.sym);
1972     pAssert(iv.t.size != 0);
1973
1974     if(nonceCaller->t.size >= iv.t.size)
1975         MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
1976     else
1977         MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
1978                   nonceCaller->t.size);
1979
1980     // CFB decrypt in place, using nonceCaller as iv
1981     CryptSymmetricDecrypt(secret->t.secret, symDef->algorithm,
1982                          symDef->keyBits.sym, TPM_ALG_CFB,
1983                          decryptKey->sensitive.sensitive.sym.t.buffer,
1984                          &iv, secret->t.size, secret->t.secret);
1985
1986     // Copy decrypted seed
1987     MemoryCopy2B(&data->b, &secret->b);
1988 }
1989 break;
1990 default:
1991     return TPM_RC_KEY;
1992     break;
1993 }
1994
1995 return TPM_RC_SUCCESS;
1996 }

```

#### 9.16.10.8 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```

1997 TPM_RC
1998 CryptParameterEncryption(
1999     TPM_HANDLE          handle,           // IN: encrypt session handle
2000     TPM2B              *nonceCaller,     // IN: nonce caller
2001     UINT16             leadingSizeInByte, // IN: the size of the leading size
2002                                     // field in bytes
2003     TPM2B_AUTH         *extraKey,       // IN: additional key material other
2004                                     // than session auth
2005     BYTE               *buffer          // IN/OUT: parameter buffer to be
2006                                     // encrypted
2007 )
2008 {
2009     SESSION             *session = SessionGet(handle); // encrypt session
2010     TPM2B_TYPE(SYM_KEY, (sizeof(extraKey->t.buffer) * 2));

```

```

2011     TPM2B_SYM_KEY    key;           // encryption key
2012     UINT32          cipherSize = 0; // size of cipher text
2013
2014     pAssert( (session->sessionKey.t.size + extraKey->t.size)
2015             <= <K>sizeof(key.t.buffer));
2016     // Retrieve encrypted data size.
2017     if(leadingSizeInByte == 2)
2018     {
2019         // Extract the first two bytes as the size field as the data size
2020         // encrypt
2021         cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
2022         // advance the buffer
2023         buffer = &buffer[2];
2024     }
2025     else if(leadingSizeInByte == 4)
2026     {
2027         // use the first four bytes to indicate the number of bytes to encrypt
2028         cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
2029         //advance pointer
2030         buffer = &buffer[4];
2031     }
2032     else
2033     {
2034         pAssert(FALSE);
2035     }
2036
2037     // Compute encryption key by concatenating sessionAuth with extra key
2038     MemoryCopy2B(&key.b, &session->sessionKey.b);
2039     MemoryConcat2B(&key.b, &extraKey->b);
2040
2041     if (session->symmetric.algorithm == TPM_ALG_XOR)
2042
2043         // XOR parameter encryption formulation:
2044         // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
2045         return CryptXORObfuscation(session->authHashAlg, &(key.b),
2046                                   &(session->nonceTPM.b),
2047                                   nonceCaller, cipherSize, buffer);
2048     else
2049         return ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
2050                               session->symmetric.keyBits.aes, &(key.b),
2051                               nonceCaller, &(session->nonceTPM.b),
2052                               cipherSize, buffer);
2053 }

```

#### 9.16.10.9 CryptParameterDecryption()

This function does in-place decryption of a command parameter.

Error Returns	Meaning
Unmarshal errors	if input buffer is in wrong canonical format

```

2054 TPM_RC
2055 CryptParameterDecryption(
2056     TPM_HANDLE    handle,           // IN: encrypted session handle
2057     TPM2B         *nonceCaller,    // IN: nonce caller
2058     UINT32        bufferSize,      // IN: size of parameter buffer
2059     UINT16        leadingSizeInByte, // IN: the size of the leading size
2060                                     // field in byte
2061     TPM2B_AUTH    *extraKey,       // IN: the authValue
2062     BYTE          *buffer           // IN/OUT: parameter buffer to be
2063                                     // decrypted
2064 )
2065 {

```

```

2066     SESSION      *session = SessionGet(handle); // encrypt session
2067     // The hmac key is going to be the concatenation of the session key and any
2068     // additional key material (like the authValue). The size of both of these
2069     // is the size of the buffer which can contain a TPMT_HA.
2070     TPM2B_TYPE(HMAC_KEY, sizeof(extraKey->t.buffer));
2071     TPM2B_HMAC_KEY      key;           // decryption key
2072
2073
2074     UINT32          cipherSize = 0;           // size of cipher text
2075
2076     // Retrieve encrypted data size.
2077     if(leadingSizeInByte == 2)
2078     {
2079         // The first two bytes of the buffer are the size of the
2080         // data to be decrypted
2081         cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
2082         buffer = &buffer[2]; // advance the buffer
2083     }
2084     else if(leadingSizeInByte == 4)
2085     {
2086         // the leading size is four bytes so get the four byte size field
2087         cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
2088         buffer = &buffer[4]; //advance pointer
2089     }
2090     else
2091     {
2092         pAssert(FALSE);
2093     }
2094     if(cipherSize > bufferSize)
2095         return TPM_RC_SIZE;
2096
2097     // Compute decryption key by concatenating sessionAuth with extra input key
2098     MemoryCopy2B(&key.b, &session->sessionKey.b);
2099     MemoryConcat2B(&key.b, &extraKey->b);
2100
2101     if(session->symmetric.algorithm == TPM_ALG_XOR)
2102         // XOR parameter decryption formulation:
2103         // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
2104         // Call XOR obfuscation function
2105         return CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
2106                                     &(session->nonceTPM.b), cipherSize, buffer);
2107     else
2108         // Assume that it is one of the symmetric block ciphers.
2109         return ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
2110                               session->symmetric.keyBits.sym,
2111                               &key.b, nonceCaller, &session->nonceTPM.b,
2112                               cipherSize, buffer);
2113
2114 }

```

#### 9.16.10.10 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```

2115 void
2116 CryptComputeSymmetricUnique(
2117     TPMI_ALG_HASH      nameAlg,           // IN: object name algorithm
2118     TPMT_SENSITIVE     *sensitive,       // IN: sensitive area
2119     TPM2B_DIGEST       *unique,         // OUT: unique buffer
2120 )
2121 {
2122     HASH_STATE hashState;
2123
2124     pAssert(sensitive != NULL || unique != NULL);

```

```

2125
2126 // Compute the public value as the hash of sensitive.symkey || unique.buffer
2127 unique->t.size = CryptGetHashDigestSize(nameAlg);
2128 CryptStartHash(nameAlg, &hashState);
2129
2130 // Add obfuscation value
2131 CryptUpdateDigest2B(&hashState, &sensitive->seedValue.b);
2132
2133 // Add sensitive value
2134 CryptUpdateDigest2B(&hashState, &sensitive->sensitive.any.b);
2135
2136 CryptCompleteHash2B(&hashState, &unique->b);
2137
2138 return;
2139 }

```

#### 9.16.10.11 CryptComputeSymValue()

This function computes the *seedValue* field in sensitive. It contains the obfuscation value for symmetric object and a seed value for storage key.

```

2140 void
2141 CryptComputeSymValue(
2142     TPM_HANDLE          parentHandle, // IN: parent handle of the
2143                               // object to be created
2144     TPMT_PUBLIC         *publicArea, // IN/OUT: the public area template
2145     TPMT_SENSITIVE     *sensitive, // IN: sensitive area
2146     TPM2B_SEED         *seed, // IN: the seed
2147     TPMT_ALG_HASH      hashAlg, // IN: hash algorithm for KDFa
2148     TPM2B_NAME         *name // IN: object name
2149 )
2150 {
2151     TPM2B_AUTH *proof = NULL;
2152
2153     if(CryptIsAsymAlgorithm(publicArea->type))
2154     {
2155         // Generate seedValue only when an asymmetric key is a storage key
2156         if(publicArea->objectAttributes.decrypt == SET
2157            && publicArea->objectAttributes.restricted == SET)
2158         {
2159             // If this is a primary object in the endorsement hierarchy, use
2160             // ehProof in the creation of the symmetric seed so that child
2161             // objects in the endorsement hierarchy are voided on TPM2_Clear()
2162             // or TPM2_ChangeEPS()
2163             if( parentHandle == TPM_RH_ENDORSEMENT
2164                && publicArea->objectAttributes.fixedTPM == SET)
2165                 proof = &gp.ehProof;
2166         }
2167         else
2168         {
2169             sensitive->seedValue.t.size = 0;
2170             return;
2171         }
2172     }
2173
2174     // For all the object type, the size of seedValue is the digest size of nameAlg
2175     sensitive->seedValue.t.size = CryptGetHashDigestSize(publicArea->nameAlg);
2176
2177     // Compute seedValue using KDFa
2178     CryptKDFa(hashAlg,
2179              &seed->b,
2180              "seedValue", // This string is a vendor-
2181              // specific information
2182              &name->b, // computed from the public template

```



```

2183         proof,
2184         sensitive->seedValue.t.size * 8,
2185         sensitive->seedValue.t.buffer, NULL);
2186
2187     return;
2188 }
2189

```

#### 9.16.10.12 CryptCreateObject()

This function creates an object. It:

- fills in the created key in public and sensitive area;
- creates a random number in sensitive area for symmetric keys; and
- compute the unique id in public area for symmetric keys.

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_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

```

2190 TPM_RC
2191 CryptCreateObject(
2192     TPM_HANDLE         parentHandle,           // IN/OUT: indication of the
2193                                     //          seed source
2194     TPMT_PUBLIC        *publicArea,           // IN/OUT: public area
2195     TPMS_SENSITIVE_CREATE *sensitiveCreate,   // IN: sensitive creation
2196     TPMT_SENSITIVE     *sensitive            // OUT: sensitive area
2197 )
2198 {
2199     // Next value is a placeholder for a random seed that is used in
2200     // key creation when the parent is not a primary seed. It has the same
2201     // size as the primary seed.
2202
2203     TPM2B_SEED         localSeed;           // data to seed key creation if this
2204                                     //          is not a primary seed
2205
2206     TPM2B_SEED         *seed = NULL;
2207     TPM_RC              result;
2208
2209     TPM2B_NAME          name;
2210     TPM_ALG_ID          hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
2211     OBJECT              *parent;
2212     UINT32              counter;
2213
2214     // Set the sensitive type for the object
2215     sensitive->sensitiveType = publicArea->type;
2216     ObjectComputeName(publicArea, &name);
2217
2218     // For all objects, copy the initial auth data
2219     sensitive->authValue = sensitiveCreate->userAuth;
2220
2221     // If this is a permanent handle assume that it is a hierarchy
2222     if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
2223     {
2224         seed = HierarchyGetPrimarySeed(parentHandle);
2225     }
2226     else

```

```

2227     {
2228         // If not hierarchy handle, get parent
2229         parent = ObjectGet(parentHandle);
2230         hashAlg = parent->publicArea.nameAlg;
2231
2232         // Use random value as seed for non-primary objects
2233         localSeed.t.size = PRIMARY_SEED_SIZE;
2234         CryptGenerateRandom(PRIMARY_SEED_SIZE, localSeed.t.buffer);
2235         seed = &localSeed;
2236     }
2237
2238     switch(publicArea->type)
2239     {
2240 #ifdef TPM_ALG_RSA
2241         // Create RSA key
2242         case TPM_ALG_RSA:
2243             result = CryptGenerateKeyRSA(publicArea, sensitive,
2244                                         hashAlg, seed, &name, &counter);
2245             if(result != TPM_RC_SUCCESS)
2246                 return result;
2247             break;
2248 #endif // TPM_ALG_RSA
2249
2250 #ifdef TPM_ALG_ECC
2251         // Create ECC key
2252         case TPM_ALG_ECC:
2253             result = CryptGenerateKeyECC(publicArea, sensitive,
2254                                         hashAlg, seed, &name, &counter);
2255             if(result != TPM_RC_SUCCESS)
2256                 return result;
2257             break;
2258 #endif // TPM_ALG_ECC
2259
2260         // Collect symmetric key information
2261         case TPM_ALG_SYMCIPHER:
2262             return CryptGenerateKeySymmetric(publicArea, sensitiveCreate,
2263                                             sensitive, hashAlg, seed, &name);
2264             break;
2265         case TPM_ALG_KEYEDHASH:
2266             return CryptGenerateKeyedHash(publicArea, sensitiveCreate,
2267                                         sensitive, hashAlg, seed, &name);
2268             break;
2269         default:
2270             FAIL(FATAL_ERROR_INTERNAL);
2271             break;
2272     }
2273
2274     // Only asymmetric keys should reach here
2275     CryptComputeSymValue(parentHandle, publicArea, sensitive, seed,
2276                         hashAlg, &name);
2277
2278     return TPM_RC_SUCCESS;
2279 }
2280

```

#### 9.16.10.13 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.

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```

2281  BOOL
2282  CryptObjectIsPublicConsistent(
2283      TPMT_PUBLIC      *publicArea          // IN: public area
2284  )
2285  {
2286      switch (publicArea->type)
2287      {
2288      #ifdef TPM_ALG_RSA
2289          case TPM_ALG_RSA:
2290              // RSA key size validation is handled by unmarshal process. No further
2291              // check is needed at this point.
2292              break;
2293      #endif //TPM_ALG_RSA
2294      #ifdef TPM_ALG_ECC
2295          case TPM_ALG_ECC:
2296              {
2297                  const ECC_CURVE          *curveValue;
2298
2299                  // Check that the public point is on the indicated curve.
2300                  if(!CryptEccIsPointOnCurve(publicArea->parameters.eccDetail.curveID,
2301                                             &publicArea->unique.ecc))
2302                      return FALSE;
2303                  curveValue = CryptEccGetCurveDataPointer(
2304                              publicArea->parameters.eccDetail.curveID);
2305                  // The input ECC curve must be a supported curve
2306                  pAssert(curveValue != NULL);
2307                  if(
2308                      curveValue->sign.scheme != TPM_ALG_NULL
2309                      && publicArea->parameters.eccDetail.scheme.scheme !=
2310                      curveValue->sign.scheme)
2311                      return FALSE;
2312              }
2313      #endif //TPM_ALG_ECC
2314
2315          default:
2316              // Symmetric object common checks
2317              // There is nothing to check with a symmetric key that is public only. Also
2318              // not sure that there is anything useful to be done with it either.
2319              return TRUE;
2320      }
2321
2322      // Asymmetric stuff falls through and is checked for consistent key sizes in
2323      // the public area
2324      if(!CryptAreKeySizesConsistent(publicArea))
2325          return FALSE;
2326      return TRUE;
2327  }

```

#### 9.16.10.14 CryptObjectPublicPrivateMatch()

This function checks the cryptographic binding between the public and sensitive areas.

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.

```

2328 TPM_RC
2329 CryptObjectPublicPrivateMatch(
2330     OBJECT          *object      // IN: the object to check
2331 )
2332 {
2333     TPMT_PUBLIC      *publicArea;
2334     TPMT_SENSITIVE   *sensitive;
2335
2336     pAssert(object != NULL);
2337     publicArea = &object->publicArea;
2338     sensitive = &object->sensitive;
2339     if(publicArea->type != sensitive->sensitiveType)
2340         return TPM_RC_TYPE;
2341
2342     switch(publicArea->type)
2343     {
2344     #ifdef TPM_ALG_RSA
2345     case TPM_ALG_RSA:
2346         // The public and private key sizes need to be consistent
2347         if(sensitive->sensitive.rsa.t.size != publicArea->unique.rsa.t.size/2)
2348             return TPM_RC_BINDING;
2349
2350         // Load key by computing the private exponent
2351         return CryptLoadPrivateRSA(object);
2352         break;
2353     #endif
2354     #ifdef TPM_ALG_ECC
2355         // This function is called from ObjectLoad() which has already checked to
2356         // see that the public point is on the curve so no need to repeat that
2357         // check.
2358     case TPM_ALG_ECC:
2359         if( publicArea->unique.ecc.x.t.size
2360             != sensitive->sensitive.ecc.t.size)
2361             return TPM_RC_BINDING;
2362         if(publicArea->nameAlg != TPM_ALG_NULL)
2363         {
2364             TPMS_ECC_POINT    publicToCompare;
2365             // Compute ECC public key
2366             CryptEccPointMultiply(&publicToCompare,
2367                                   publicArea->parameters.eccDetail.curveID,
2368                                   &sensitive->sensitive.ecc, NULL);
2369             // Compare ECC public key
2370             if( (!Memory2BEqual(&publicArea->unique.ecc.x.b,
2371                                 &publicToCompare.x.b))
2372                 || (!Memory2BEqual(&publicArea->unique.ecc.y.b,
2373                                     &publicToCompare.y.b)))
2374                 return TPM_RC_BINDING;
2375         }
2376         return TPM_RC_SUCCESS;
2377         break;
2378     #endif
2379     case TPM_ALG_KEYEDHASH:
2380         break;
2381     case TPM_ALG_SYMCIPHER:
2382         if( (publicArea->parameters.symDetail.keyBits.sym + 7)/8
2383             != sensitive->sensitive.sym.t.size)
2384             return TPM_RC_BINDING;
2385         break;

```

```

2386     default:
2387         // The choice here is an assert or a return of a bad type for the object
2388         return TPM_RC_TYPE;
2389         break;
2390     }
2391
2392     // For asymmetric keys, the algorithm for validating the linkage between
2393     // the public and private areas is algorithm dependent. For symmetric keys
2394     // the linkage is based on hashing the symKey and obfuscation values.
2395     if(publicArea->nameAlg != TPM_ALG_NULL)
2396     {
2397         TPM2B_DIGEST    uniqueToCompare;
2398
2399         // Compute unique for symmetric key
2400         CryptComputeSymmetricUnique(publicArea->nameAlg, sensitive,
2401                                     &uniqueToCompare);
2402
2403         // Compare unique
2404         if(!Memory2BEqual(&publicArea->unique.sym.b,
2405                           &uniqueToCompare.b))
2406             return TPM_RC_BINDING;
2407     }
2408     return TPM_RC_SUCCESS;
2409 }

```

#### 9.16.10.15 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

```

2410 TPMI_ALG_HASH
2411 CryptGetSignHashAlg(
2412     TPMT_SIGNATURE    *auth           // IN: signature
2413 )
2414 {
2415     pAssert(auth->sigAlg != TPM_ALG_NULL);
2416
2417     // Get authHash algorithm based on signing scheme
2418     switch(auth->sigAlg)
2419     {
2420
2421     #ifdef TPM_ALG_RSA
2422     case TPM_ALG_RSASSA:
2423         return auth->signature.rsassa.hash;
2424
2425     case TPM_ALG_RSAPSS:
2426         return auth->signature.rsapss.hash;
2427
2428     #endif //TPM_ALG_RSA
2429
2430     #ifdef TPM_ALG_ECC
2431     case TPM_ALG_ECDSA:
2432         return auth->signature.ecdsa.hash;
2433
2434     #endif //TPM_ALG_ECC
2435
2436     case TPM_ALG_HMAC:
2437         return auth->signature.hmac.hashAlg;
2438
2439     default:
2440         return TPM_ALG_NULL;
2441     }
2442 }

```

**9.16.10.16 CryptIsSplitSign()**

This function is used to determine if the signing operation is a split signing operation that required a TPM2\_Commit().

```

2443 BOOL
2444 CryptIsSplitSign(
2445     TPM_ALG_ID      scheme           // IN: the algorithm selector
2446 )
2447 {
2448     if( scheme != scheme
2449 #   ifdef   TPM_ALG_ECDAE
2450         || scheme == TPM_ALG_ECDAE
2451 #   endif   // TPM_ALG_ECDAE
2452
2453
2454     )
2455         return TRUE;
2456     return FALSE;
2457 }

```

**9.16.10.17 CryptIsSignScheme()**

This function indicates if a scheme algorithm is a sign algorithm.

```

2458 BOOL
2459 CryptIsSignScheme(
2460     TPMI_ALG_ASYNC_SCHEME  scheme
2461 )
2462 {
2463     switch(scheme)
2464     {
2465 #ifdef TPM_ALG_RSA
2466         // If RSA is implemented, then both signing schemes are required
2467         case TPM_ALG_RSASSA:
2468         case TPM_ALG_RSAPSS:
2469             return TRUE;
2470             break;
2471 #endif //TPM_ALG_RSA
2472
2473 #ifdef TPM_ALG_ECC
2474         // If ECC is implemented ECDSA is required
2475         case TPM_ALG_ECDSA:
2476 #ifdef TPM_ALG_ECDAE
2477         // ECDAE is optional
2478         case TPM_ALG_ECDAE:
2479 #endif
2480 #ifdef   TPM_ALG_ECSCNORR
2481         // Schnorr is also optional
2482         case TPM_ALG_ECSCNORR:
2483 #endif
2484 #ifdef   TPM_ALG_SM2
2485         case TPM_ALG_SM2:
2486 #endif
2487             return TRUE;
2488             break;
2489 #endif //TPM_ALG_ECC
2490         default:
2491             return FALSE;
2492             break;
2493     }
2494 }

```

### 9.16.10.18 CryptIsDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```

2495  BOOL
2496  CryptIsDecryptScheme(
2497      TPMI_ALG_ASYNC_SCHEME    scheme
2498  )
2499  {
2500      switch(scheme)
2501      {
2502      #ifndef TPM_ALG_RSA
2503          // If RSA is implemented, then both decrypt schemes are required
2504          case TPM_ALG_RSAES:
2505          case TPM_ALG_OAEP:
2506              return TRUE;
2507              break;
2508      #endif //TPM_ALG_RSA
2509
2510      #ifndef TPM_ALG_ECC
2511          // If ECC is implemented ECDH is required
2512          case TPM_ALG_ECDH:
2513      #ifndef TPM_ALG_SM2
2514          case TPM_ALG_SM2:
2515      #endif
2516      #ifndef TPM_ALG_ECMQV
2517          case TPM_ALG_ECMQV:
2518      #endif
2519              return TRUE;
2520              break;
2521      #endif //TPM_ALG_ECC
2522          default:
2523              return FALSE;
2524              break;
2525      }
2526  }

```

### 9.16.10.19 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.

Error Returns	Meaning
TPM_RC_KEY	key referenced by <i>signHandle</i> is not a signing key
TPM_RC_SCHEME	both <i>scheme</i> and key's default scheme are empty; or <i>scheme</i> is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from <i>scheme</i>

```

2527  TPM_RC
2528  CryptSelectSignScheme(
2529      TPMI_DH_OBJECT    signHandle,          // IN: handle of signing key
2530      TPMT_SIG_SCHEME   *scheme             // IN/OUT: signing scheme
2531  )
2532  {
2533      OBJECT            *signObject;
2534      TPMT_SIG_SCHEME   *objectScheme;

```

```

2535     TPMT_PUBLIC          *publicArea;
2536
2537     // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
2538     // of the setting of scheme
2539     if(signHandle == TPM_RH_NULL)
2540     {
2541         scheme->scheme = TPM_ALG_NULL;
2542         scheme->details.any.hashAlg = TPM_ALG_NULL;
2543         return TPM_RC_SUCCESS;
2544     }
2545
2546     // Get sign object pointer
2547     signObject = ObjectGet(signHandle);
2548     publicArea = &signObject->publicArea;
2549
2550     // is this a signing key?
2551     if(!publicArea->objectAttributes.sign)
2552         return TPM_RC_KEY;
2553
2554     if(CryptIsAsymAlgorithm(publicArea->type))
2555         objectScheme =
2556             (TPMT_SIG_SCHEME *)&publicArea->parameters.asymDetail.scheme;
2557     else
2558         objectScheme =
2559             (TPMT_SIG_SCHEME *)&publicArea->parameters.keyedHashDetail.scheme;
2560
2561     // If the object doesn't have a default scheme, then use the input scheme.
2562     if(objectScheme->scheme == TPM_ALG_NULL)
2563     {
2564         // Input and default can't both be NULL
2565         if(scheme->scheme == TPM_ALG_NULL)
2566             return TPM_RC_SCHEME;
2567
2568         // Assume that the scheme is compatible with the key. If not,
2569         // we will generate an error in the signing operation.
2570         return TPM_RC_SUCCESS;
2571     }
2572
2573     else if(scheme->scheme == TPM_ALG_NULL)
2574     {
2575         // input scheme is NULL so use default
2576
2577         // First, check to see if the default requires that the caller provide
2578         // scheme data
2579         if(CryptIsSplitSign(objectScheme->scheme))
2580             return TPM_RC_SCHEME;
2581
2582         scheme->scheme = objectScheme->scheme;
2583         scheme->details.any.hashAlg = objectScheme->details.any.hashAlg;
2584         return TPM_RC_SUCCESS;
2585     }
2586     // Both input and object have scheme selectors
2587     // If the scheme and the hash are not the same then...
2588     if( objectScheme->scheme != scheme->scheme
2589        || objectScheme->details.any.hashAlg != scheme->details.any.hashAlg)
2590         return TPM_RC_SCHEME;
2591
2592     return TPM_RC_SUCCESS;
2593 }

```



### 9.16.10.20 CryptSign()

Sign a digest by an asymmetric key. This function is called by attestation commands and the generic TPM2\_Sign() command. This function checks the key type, scheme and digest size. Note, it does not check if the sign operation is allowed for restricted key. It should be checked before the function is called.

Error Returns	Meaning
TPM_RC_ATTRIBUTES	<i>signHandle</i> references not a signing key
TPM_RC_SCHEME	<i>signScheme</i> is not compatible with the signing key type
TPM_RC_VALUE	<i>digest</i> value is greater than the modulus of <i>signHandle</i> or size of <i>hashData</i> does not match hash algorithm in <i>signScheme</i> (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```

2594 TPM_RC
2595 CryptSign(
2596     TPMI_DH_OBJECT      signHandle,          // IN: The handle of sign key
2597     TPMT_SIG_SCHEME     *signScheme,         // IN: sign scheme.
2598     TPM2B_DIGEST        *digest,            // IN: The digest being signed
2599     TPMT_SIGNATURE      *signature          // OUT: signature
2600 )
2601 {
2602     OBJECT               *signKey = ObjectGet(signHandle);
2603
2604     // check if input handle is a sign key
2605     if (signKey->publicArea.objectAttributes.sign != SET)
2606         return TPM_RC_ATTRIBUTES;
2607
2608     // Must have the private portion loaded. This check is made during
2609     // authorization.
2610     pAssert(signKey->attributes.publicOnly == CLEAR);
2611
2612     // Initialize signature scheme
2613     signature->sigAlg = signScheme->scheme;
2614
2615     // Initialize signature hash
2616     signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
2617
2618     // perform sign operation based on different key type
2619 #ifndef TPM_ALG_RSA
2620     if(signKey->publicArea.type == TPM_ALG_RSA)
2621     {
2622         // Sign it
2623         return CryptSignRSA(signKey, signScheme, digest, signature);
2624     }
2625 #endif //TPM_ALG_RSA
2626
2627 #ifndef TPM_ALG_ECC
2628     if(signKey->publicArea.type == TPM_ALG_ECC)
2629     {
2630         // Perform the signature operation
2631         return CryptSignECC(signKey, signScheme, digest, signature);
2632     }
2633 #endif //TPM_ALG_ECC
2634
2635     if(signKey->publicArea.type == TPM_ALG_KEYEDHASH)
2636     {
2637         // Sign
2638         return CryptSignHMAC(signKey, signScheme, digest, signature);
2639     }
2640
2641     pAssert(FALSE);
2642     return TPM_RC_ATTRIBUTES; // This is unreachable code but this makes the

```

```

2643 // compiler happy
2644
2645 }

```

### 9.16.10.21 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.

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SCHEME	the scheme is not supported

```

2646 TPM_RC
2647 CryptVerifySignature(
2648     TPMI_DH_OBJECT    keyHandle,           // IN: The handle of sign key
2649     TPM2B_DIGEST      *digest,           // IN: The digest being validated
2650     TPMT_SIGNATURE     *signature        // IN: signature
2651 )
2652 {
2653     OBJECT             *authObject = ObjectGet(keyHandle);
2654     TPMT_PUBLIC        *publicArea = &authObject->publicArea;
2655
2656
2657     switch (publicArea->type)
2658     {
2659
2660     #ifdef TPM_ALG_RSA
2661     case TPM_ALG_RSA:
2662         return CryptRSAVerifySignature(authObject, digest, signature);
2663         break;
2664     #endif //TPM_ALG_RSA
2665
2666     #ifdef TPM_ALG_ECC
2667     case TPM_ALG_ECC:
2668         return CryptECCVerifySignature(authObject, digest, signature);
2669         break;
2670
2671     #endif // TPM_ALG_ECC
2672
2673     case TPM_ALG_KEYEDHASH:
2674         return CryptHMACVerifySignature(authObject, digest, signature);
2675         break;
2676
2677     default:
2678         pAssert(FALSE);
2679         return TPM_RC_SCHEME; // This is unreachable but it makes the compiler
2680                               // happy.
2681         break;
2682     }
2683
2684 }

```

## 9.16.11 Math functions

### 9.16.11.1 CryptDivide()

This function interfaces to the math library for large number divide.

Error Returns	Meaning
TPM_RC_SIZE	<i>quotient</i> or <i>remainder</i> is too small to receive the result

```

2685 TPM_RC
2686 CryptDivide(
2687     TPM2B     *numerator,      // IN: numerator
2688     TPM2B     *denominator,   // IN: denominator
2689     TPM2B     *quotient,      // OUT: quotient = numerator / denominator.
2690     TPM2B     *remainder      // OUT: numerator mod denominator.
2691 )
2692 {
2693     pAssert( numerator != NULL && denominator!= NULL
2694             && (quotient != NULL || remainder != NULL)
2695             );
2696     // assume denominator is not 0
2697     pAssert(denominator->size != 0);
2698
2699     return TranslateCryptErrors(_math__Div(numerator,
2700                                         denominator,
2701                                         quotient,
2702                                         remainder)
2703                                );
2704 }

```

### 9.16.11.2 CryptCompare()

This function interfaces to the math library for large number, unsigned compare.

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```

2705 int
2706 CryptCompare(
2707     const UINT32  aSize,      // IN: size of a
2708     const BYTE    *a,        // IN: a buffer
2709     const UINT32  bSize,      // IN: size of b
2710     const BYTE    *b        // IN: b buffer
2711 )
2712 {
2713     int          borrow = 0;
2714     int          notZero = 0;
2715     int          i;
2716     // If a has more digits than b, then a is greater than b if
2717     // any of the more significant bytes is non zero
2718     if((i = (int)aSize - (int)bSize) > 0)
2719         for(; i > 0; i--)
2720             if(*a++) // means a > b
2721                 return 1;
2722     // If b has more digits than a, then b is greater if any of the
2723     // more significant bytes is non zero
2724     if(i < 0) <Q>// Means that b is longer than a

```

```

2725     for(; i < 0; i++)
2726         if(*b++) // means that b > a
2727             return -1;
2728     // Either the vales are the same size or the upper bytes of a or b are
2729     // all zero, so compare the rest
2730     i = (aSize > bSize) ? bSize : aSize;
2731     a = &a[i-1];
2732     b = &b[i-1];
2733     for(; i > 0; i--)
2734     {
2735         borrow = *a-- - *b-- + borrow;
2736         notZero = notZero || borrow;
2737         borrow >= 8;
2738     }
2739     // if there is a borrow, then b > a
2740     if(borrow)
2741         return -1;
2742     // either a > b or they are the same
2743     return notZero;
2744 }

```

### 9.16.11.3 CryptCompareSigned()

This function interfaces to the math library for large number, signed compare.

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```

2745 int
2746 CryptCompareSigned(
2747     UINT32     aSize,           // IN: size of a
2748     BYTE       *a,             // IN: a buffer
2749     UINT32     bSize,           // IN: size of b
2750     BYTE       *b,             // IN: b buffer
2751 )
2752 {
2753     int     signA, signB;       // sign of a and b
2754
2755     // For positive or 0, sign_a is 1
2756     // for negative, sign_a is 0
2757     signA = ((a[0] & 0x80) == 0) ? 1 : 0;
2758
2759     // For positive or 0, sign_b is 1
2760     // for negative, sign_b is 0
2761     signB = ((b[0] & 0x80) == 0) ? 1 : 0;
2762
2763     if(signA != signB)
2764     {
2765         return signA - signB;
2766     }
2767
2768     if(signA == 1)
2769         // do unsigned compare function
2770         return CryptCompare(aSize, a, bSize, b);
2771     else
2772         // do unsigned compare the other way
2773         return 0 - CryptCompare(aSize, a, bSize, b);
2774 }

```

## 9.16.12 Self Testing Functions

### 9.16.12.1 Introduction

Self testing mechanism is hardware dependent and is not available at a software simulator environment. So we do not really deploy a self testing mechanism here, but always gives a pseudo return for all the self-test functions. Vendors should replace these functions with implementations that perform proper self-test.

### 9.16.12.2 CryptSelfTest

This function is called to start a full self-test.

NOTE: the behavior in this function is NOT the correct behavior for a real TPM implementation. An artificial behavior is placed here due to the limitation of a software simulation environment. For the correct behavior, consult the part 3 specification for TPM2\_SelfTest().

Error Returns	Meaning
TPM_RC_TESTING	if <i>fullTest</i> is YES

```

2775 TPM_RC
2776 CryptSelfTest(
2777     TPMI_YES_NO          fullTest          // IN: if full test is required
2778 )
2779 {
2780     if(fullTest == YES)
2781         return TPM_RC_TESTING;
2782     else
2783         return TPM_RC_SUCCESS;
2784 }

```

### 9.16.12.3 CryptIncrementalSelfTest

This function is used to start an incremental self-test.

Error Returns	Meaning
TPM_RC_TESTING	if <i>toTest</i> list is not empty

```

2785 TPM_RC
2786 CryptIncrementalSelfTest(
2787     TPML_ALG          *toTest,          // IN: list of algorithms to be tested
2788     TPML_ALG          *toDoList        // OUT: list of algorithms needing test
2789 )
2790 {
2791     CRYPT_RESULT      retVal;
2792     retVal = _cpri_IncrementalSelfTest(toTest, toDoList);
2793     if(TranslateCryptErrors(retVal) == TPM_RC_SUCCESS)
2794         return TPM_RC_SUCCESS;
2795     else
2796         return TPM_RC_TESTING;
2797 }

```

### 9.16.12.4 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 the part 3 specification for TPM2\_GetTestResult().

```

2798 TPM_RC
2799 CryptGetTestResult(
2800     TPM2B_MAX_BUFFER *outData           // OUT: test result data
2801 )
2802 {
2803     outData->t.size = 0;
2804     return TPM_RC_SUCCESS;
2805 }

```

## 9.16.13 Capability Support

### 9.16.13.1 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

Return Value	Meaning
YES	if no more ECC curve is available
NO	if there are more ECC curves not reported

```

2806 #ifndef TPM_ALG_ECC /*% 5
2807 TPME_YES_NO
2808 CryptCapGetECCCurve(
2809     TPM_ECC_CURVE    curveID,           // IN: the starting ECC curve
2810     UINT32           maxCount,          // IN: count of returned curves
2811     TPML_ECC_CURVE  *curveList        // OUT: ECC curve list
2812 )
2813 {
2814     TPME_YES_NO    more = NO;
2815     UINT16         i;
2816     UINT32         count = _cpri__EccGetCurveCount();
2817     TPM_ECC_CURVE curve;
2818
2819     // Initialize output property list
2820     curveList->count = 0;
2821
2822     // The maximum count of curves we may return is MAX_ECC_CURVES
2823     if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
2824
2825     // Scan the eccCurveValues array
2826     for(i = 0; i < count; i++)
2827     {
2828         curve = _cpri__GetCurveIdByIndex(i);
2829         // If curveID is less than the starting curveID, skip it
2830         if(curve < curveID)
2831             continue;
2832
2833         if(curveList->count < maxCount)
2834         {
2835             // If we have not filled up the return list, add more curves to
2836             // it
2837             curveList->eccCurves[curveList->count] = curve;
2838             curveList->count++;
2839         }
2840         else
2841         {
2842             // If the return list is full but we still have curves
2843             // available, report this and stop iterating
2844             more = YES;

```

```

2845         break;
2846     }
2847 }
2848 }
2849 }
2850     return more;
2851 }
2852 }

```

### 9.16.13.2 CryptCapGetEccCurveNumber()

This function returns the number of ECC curves supported by the TPM.

```

2853     UINT32
2854     CryptCapGetEccCurveNumber(void)
2855     {
2856         // There is an array that holds the curve data. Its size divided by the
2857         // size of an entry is the number of values in the table.
2858         return _cpri__EccGetCurveCount();
2859     }
2860 #endif //TPM_ALG_ECC // % 5

```

### 9.16.13.3 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.

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```

2861     BOOL
2862     CryptAreKeySizesConsistent(
2863         TPMT_PUBLIC *publicArea // IN: the public area to check
2864     )
2865     {
2866     #ifdef TPM_ALG_RSA
2867         if(publicArea->type == TPM_ALG_RSA)
2868         {
2869             // The key size in bits is filtered by the unmarshaling
2870             return ( ((publicArea->parameters.rsaDetail.keyBits+7)/8)
2871                 == publicArea->unique.rsa.t.size);
2872         }
2873     #endif //TPM_ALG_RSA
2874
2875     #ifdef TPM_ALG_ECC
2876         if(publicArea->type == TPM_ALG_ECC)
2877         {
2878             UINT16 keySizeInBytes;
2879             TPM_ECC_CURVE curveId = publicArea->parameters.eccDetail.curveID;
2880
2881             keySizeInBytes = CryptEccGetKeySizeInBytes(curveId);
2882
2883             return ( keySizeInBytes > 0
2884                 && publicArea->unique.ecc.x.t.size <= keySizeInBytes
2885                 && publicArea->unique.ecc.y.t.size <= keySizeInBytes);
2886         }
2887     #endif //TPM_ALG_ECC
2888

```

```

2889     return 0;
2890 }

```

## 9.17 Ticket.c

### 9.17.1 Introduction

This clause contains the functions used for ticket computations.

### 9.17.2 Includes

```

1 #include "InternalRoutines.h"

```

### 9.17.3 Functions

#### 9.17.3.1 TicketIsSafe()

This function indicates if producing a ticket is safe. It checks if the leading bytes of an input buffer is TPM\_GENERATED\_VALUE or its substring of canonical form. If so, it is not safe to produce ticket for an input buffer claiming to be TPM generated buffer

Return Value	Meaning
TRUE	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;
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 }

```

#### 9.17.3.2 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

```



```

28                                     //      values
29     TPMT_TK_VERIFIED    *ticket      // OUT: verified ticket
30 )
31 {
32     TPM2B_AUTH          *proof;
33     HMAC_STATE         hmacState;
34
35     // Fill in ticket fields
36     ticket->tag = TPM_ST_VERIFIED;
37     ticket->hierarchy = hierarchy;
38
39     // Use the proof value of the hierarchy
40     proof = HierarchyGetProof(hierarchy);
41
42     // Start HMAC
43     ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
44                                             &proof->b, &hmacState);
45
46     // add TPM_ST_VERIFIED
47     CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
48
49     // add digest
50     CryptUpdateDigest2B(&hmacState, &digest->b);
51
52     // add key name
53     CryptUpdateDigest2B(&hmacState, &keyName->b);
54
55     // complete HMAC
56     CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
57
58     return;
59 }

```

### 9.17.3.3 TicketComputeAuth()

This function creates a TPMT\_TK\_AUTH ticket.

```

60 void
61 TicketComputeAuth(
62     TPM_ST              type,           // IN: the type of ticket.
63     TPMI_RH_HIERARCHY  hierarchy,     // IN: hierarchy constant for ticket
64     UINT64             timeout,       // IN: timeout
65     TPM2B_DIGEST       *cpHashA,     // IN: input cpHashA
66     TPM2B_NONCE        *policyRef,    // IN: input policyRef
67     TPM2B_NAME         *entityName,   // IN: name of entity
68     TPMT_TK_AUTH      *ticket        // OUT: Created ticket
69 )
70 {
71     TPM2B_AUTH          *proof;
72     HMAC_STATE         hmacState;
73
74     // Get proper proof
75     proof = HierarchyGetProof(hierarchy);
76
77     // Fill in ticket fields
78     ticket->tag = type;
79     ticket->hierarchy = hierarchy;
80
81     // Start HMAC
82     ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
83                                             &proof->b, &hmacState);
84
85     // Adding TPM_ST_AUTH
86     CryptUpdateDigestInt(&hmacState, sizeof(UINT16), &ticket->tag);

```

```

87
88     // Adding timeout
89     CryptUpdateDigestInt(&hmacState, sizeof(UINT64), &timeout);
90
91     // Adding cpHash
92     CryptUpdateDigest2B(&hmacState, &cpHashA->b);
93
94     // Adding policyRef
95     CryptUpdateDigest2B(&hmacState, &policyRef->b);
96
97     // Adding keyName
98     CryptUpdateDigest2B(&hmacState, &entityName->b);
99
100    // Compute HMAC
101    CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
102
103    return;
104 }

```

#### 9.17.3.4 TicketComputeHashCheck()

This function creates a TPMT\_TK\_HASHCHECK ticket.

```

105 void
106 TicketComputeHashCheck(
107     TPMI_RH_HIERARCHY    hierarchy,    // IN: hierarchy constant for ticket
108     TPM2B_DIGEST         *digest,      // IN: input digest
109     TPMT_TK_HASHCHECK    *ticket      // OUT: Created ticket
110 )
111 {
112     TPM2B_AUTH            *proof;
113     HMAC_STATE           hmacState;
114
115     // Get proper proof
116     proof = HierarchyGetProof(hierarchy);
117
118     // Fill in ticket fields
119     ticket->tag = TPM_ST_HASHCHECK;
120     ticket->hierarchy = hierarchy;
121
122     ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
123                                             &proof->b, &hmacState);
124
125     // Add TPM_ST_HASHCHECK
126     CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
127
128     // Add digest
129     CryptUpdateDigest2B(&hmacState, &digest->b);
130
131     // Compute HMAC
132     CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
133
134     return;
135 }

```

#### 9.17.3.5 TicketComputeCreation()

This function creates a TPMT\_TK\_CREATION ticket.

```

136 void
137 TicketComputeCreation(
138     TPMI_RH_HIERARCHY    hierarchy,    // IN: hierarchy for ticket
139     TPM2B_NAME            *name,       // IN: object name

```

```
140     TPM2B_DIGEST      *creation,      // IN: creation hash
141     TPMT_TK_CREATION  *ticket        // OUT: created ticket
142 )
143 {
144     TPM2B_AUTH        *proof;
145     HMAC_STATE        hmacState;
146
147     // Get proper proof
148     proof = HierarchyGetProof(hierarchy);
149
150     // Fill in ticket fields
151     ticket->tag = TPM_ST_CREATION;
152     ticket->hierarchy = hierarchy;
153
154     ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
155                                             &proof->b, &hmacState);
156
157     // Add TPM_ST_CREATION
158     CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
159
160     // Add name
161     CryptUpdateDigest2B(&hmacState, &name->b);
162
163     // Add creation hash
164     CryptUpdateDigest2B(&hmacState, &creation->b);
165
166     // Compute HMAC
167     CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
168
169     return;
170 }
```

**Annex A**  
(informative)  
**Implementation Dependent**

**A.1 Introduction**

This header file contains definitions that are derived from the values in the annexes of part 2. This file would change based on the implementation.

The values shown in this version of the file reflect the example settings in part 2.

## A.2 Implementation.h

```

1  #ifndef      _IMPLEMENTATION_H
2  #define      _IMPLEMENTATION_H
3  #ifndef      ALG_ALL
4  #define      ALG_ALL      NO
5  #endif

```

Part2AnnexParser() Generated (Mar 6, 2013 11:47:21 AM)

```

6  #include      "BaseTypes.h"
7  #ifdef TRUE
8  #undef TRUE
9  #endif
10 #ifdef FALSE
11 #undef FALSE
12 #endif

```

Table 205 -- SHA1 Hash Values

```

13 #define      SHA1_DIGEST_SIZE      20
14 #define      SHA1_BLOCK_SIZE      64
15 #define      SHA1_DER_SIZE      15
16 #define      SHA1_DER      {\
17     0x30,0x21,0x30,0x09,0x06,0x05,0x2B,0x0E,0x03,0x02,0x1A,0x05,0x00,0x04,0x14}

```

Table 206 -- SHA256 Hash Values

```

18 #define      SHA256_DIGEST_SIZE      32
19 #define      SHA256_BLOCK_SIZE      64
20 #define      SHA256_DER_SIZE      19
21 #define      SHA256_DER      {\
22     0x30,0x31,0x30,0x0d,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x01,\
23     0x05,0x00,0x04,0x20}

```

Table 207 -- SHA384 Hash Values

```

24 #define      SHA384_DIGEST_SIZE      48
25 #define      SHA384_BLOCK_SIZE      128
26 #define      SHA384_DER_SIZE      19
27 #define      SHA384_DER      {\
28     0x30,0x41,0x30,0x0d,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x02,\
29     0x05,0x00,0x04,0x30}

```

Table 208 -- SHA512 Hash Values

```

30 #define      SHA512_DIGEST_SIZE      64
31 #define      SHA512_BLOCK_SIZE      128
32 #define      SHA512_DER_SIZE      19
33 #define      SHA512_DER      {\
34     0x30,0x51,0x30,0x0d,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x03,\
35     0x05,0x00,0x04,0x40}

```

Table 210 -- SM3\_256 Hash Values

```

36 #define      SM3_256_DIGEST_SIZE      32
37 #define      SM3_256_BLOCK_SIZE      64
38 #define      SM3_256_DER_SIZE      18
39 #define      SM3_256_DER      {\
40     0x30,0x30,0x30,0x0c,0x06,0x08,0x2a,0x81,0x1c,0x81,0x45,0x01,0x83,0x11,0x05,\
41     0x00,0x04,0x20}

```

Table 211 -- Architectural Limits Values

```
42 #define MAX_SESSION_NUMBER 3
```

Table 213 -- Logic Values

```
43 #define YES 1
44 #define NO 0
45 #define TRUE 1
46 #define FALSE 0
47 #define SET 1
48 #define CLEAR 0
```

Table 214 -- Processor Values

```
49 #define BIG_ENDIAN_TPM NO // 0
50 #define LITTLE_ENDIAN_TPM YES // 1
51 #define NO_AUTO_ALIGN NO // 0
```

Table 215 -- Implemented Algorithms

```
52 #define ALG_RSA YES // 1
53 #define ALG_SHA1 YES // 1
54 #define ALG_HMAC YES // 1
55 #define ALG_AES YES // 1
56 #define ALG_MGF1 YES // 1
57 #define ALG_XOR YES // 1
58 #define ALG_KEYEDHASH YES // 1
59 #define ALG_SHA256 YES // 1
60 #define ALG_SHA384 NO // 0
61 #define ALG_SHA512 NO // 0
62 #define ALG_SM3_256 YES // 1
63 #define ALG_SM4 YES // 1
64 #define ALG_RSASSA YES // 1
65 #define ALG_RSAES YES // 1
66 #define ALG_RSAPSS YES // 1
67 #define ALG_OAEP YES // 1
68 #define ALG_ECC YES // 1
69 #define ALG_ECDH YES // 1
70 #define ALG_ECDSA YES // 1
71 #define ALG_ECDSA YES // 1
72 #define ALG_SM2 YES // 1
73 #define ALG_ECSCNORR YES // 1
74 #define ALG_ECMQV NO // 0
75 #define ALG_SYMCIPHER YES // 1
76 #define ALG_KDF1_SP800_56a YES // 1
77 #define ALG_KDF2 NO // 0
78 #define ALG_KDF1_SP800_108 YES // 1
79 #define ALG_CTR YES // 1
80 #define ALG_OFB YES // 1
81 #define ALG_CBC YES // 1
82 #define ALG_CFB YES // 1
83 #define ALG_ECB YES // 1
```

Table 216 -- Implemented Commands

```
84 #define CC_ActivateCredential YES // 1
85 #define CC_Certify YES // 1
86 #define CC_CertifyCreation YES // 1
87 #define CC_ChangeEPS YES // 1
88 #define CC_ChangePPS YES // 1
89 #define CC_Clear YES // 1
90 #define CC_ClearControl YES // 1
```

```

91 #define CC_ClockRateAdjust YES // 1
92 #define CC_ClockSet YES // 1
93 #define CC_Commit ALG_ECC // 1
94 #define CC_ContextLoad YES // 1
95 #define CC_ContextSave YES // 1
96 #define CC_Create YES // 1
97 #define CC_CreatePrimary YES // 1
98 #define CC_DictionaryAttackLockReset YES // 1
99 #define CC_DictionaryAttackParameters YES // 1
100 #define CC_Duplicate YES // 1
101 #define CC_ECC_Parameters ALG_ECC // 1
102 #define CC_ECDH_KeyGen ALG_ECC // 1
103 #define CC_ECDH_ZGen ALG_ECC // 1
104 #define CC_EncryptDecrypt YES // 1
105 #define CC_EventSequenceComplete YES // 1
106 #define CC_EvictControl YES // 1
107 #define CC_FieldUpgradeData NO // 0
108 #define CC_FieldUpgradeStart NO // 0
109 #define CC_FirmwareRead NO // 0
110 #define CC_FlushContext YES // 1
111 #define CC_GetCapability YES // 1
112 #define CC_GetCommandAuditDigest YES // 1
113 #define CC_GetCommand YES // 1
114 #define CC_GetSessionAuditDigest YES // 1
115 #define CC_GetTestResult YES // 1
116 #define CC_GetTime YES // 1
117 #define CC_Hash YES // 1
118 #define CC_HashSequenceStart YES // 1
119 #define CC_HierarchyChangeAuth YES // 1
120 #define CC_HierarchyControl YES // 1
121 #define CC_HMAC YES // 1
122 #define CC_HMAC_Start YES // 1
123 #define CC_Import YES // 1
124 #define CC_IncrementalSelfTest YES // 1
125 #define CC_Load YES // 1
126 #define CC_LoadExternal YES // 1
127 #define CC_MakeCredential YES // 1
128 #define CC_NV_Certify YES // 1
129 #define CC_NV_ChangeAuth YES // 1
130 #define CC_NV_DefineSpace YES // 1
131 #define CC_NV_Extend YES // 1
132 #define CC_NV_GlobalWriteLock YES // 1
133 #define CC_NV_Increment YES // 1
134 #define CC_NV_Read YES // 1
135 #define CC_NV_ReadLock YES // 1
136 #define CC_NV_ReadPublic YES // 1
137 #define CC_NV_SetBits YES // 1
138 #define CC_NV_UndefineSpace YES // 1
139 #define CC_NV_UndefineSpaceSpecial YES // 1
140 #define CC_NV_Write YES // 1
141 #define CC_NV_WriteLock YES // 1
142 #define CC_ObjectChangeAuth YES // 1
143 #define CC_PCR_Allocate YES // 1
144 #define CC_PCR_Event YES // 1
145 #define CC_PCR_Extend YES // 1
146 #define CC_PCR_Read YES // 1
147 #define CC_PCR_Reset YES // 1
148 #define CC_PCR_SetAuthPolicy YES // 1
149 #define CC_PCR_SetAuthValue YES // 1
150 #define CC_PolicyAuthorize YES // 1
151 #define CC_PolicyAuthValue YES // 1
152 #define CC_PolicyCommandCode YES // 1
153 #define CC_PolicyCounterTimer YES // 1
154 #define CC_PolicyCpHash YES // 1
155 #define CC_PolicyDuplicationSelect YES // 1
156 #define CC_PolicyGetDigest YES // 1

```



```

157 #define CC_PolicyLocality YES // 1
158 #define CC_PolicyNameHash YES // 1
159 #define CC_PolicyNV YES // 1
160 #define CC_PolicyOR YES // 1
161 #define CC_PolicyPassword YES // 1
162 #define CC_PolicyPCR YES // 1
163 #define CC_PolicyPhysicalPresence YES // 1
164 #define CC_PolicyRestart YES // 1
165 #define CC_PolicySecret YES // 1
166 #define CC_PolicySigned YES // 1
167 #define CC_PolicyTicket YES // 1
168 #define CC_PP_Commands YES // 1
169 #define CC_Quote YES // 1
170 #define CC_ReadClock YES // 1
171 #define CC_ReadPublic YES // 1
172 #define CC_Rewrap YES // 1
173 #define CC_RSA_Decrypt ALG_RSA // 1
174 #define CC_RSA_Encrypt ALG_RSA // 1
175 #define CC_SelfTest YES // 1
176 #define CC_SequenceComplete YES // 1
177 #define CC_SequenceUpdate YES // 1
178 #define CC_SetAlgorithmSet YES // 1
179 #define CC_SetCommandCodeAuditStatus YES // 1
180 #define CC_SetPrimaryPolicy YES // 1
181 #define CC_Shutdown YES // 1
182 #define CC_Sign YES // 1
183 #define CC_StartAuthSession YES // 1
184 #define CC_Startup YES // 1
185 #define CC_StirRandom YES // 1
186 #define CC_TestParms YES // 1
187 #define CC_Unseal YES // 1
188 #define CC_VerifySignature YES // 1
189 #define CC_ZGen_2Phase YES // 1
190 #define CC_EC_Ephemeral YES // 1

```

Table 217 -- RSA Algorithm Constants

```

191 #define RSA_KEY_SIZES_BITS {1024, 2048} // {1024,2048}
192 #define MAX_RSA_KEY_BITS 2048
193 #define MAX_RSA_KEY_BYTES ((MAX_RSA_KEY_BITS + 7) / 8) // 256

```

Table 218 -- ECC Algorithm Constants

```

194 #define ECC_CURVES {\
195     TPM_ECC_NIST_P256,TPM_ECC_BN_P256,TPM_ECC_SM2_P256}#define ECC_KEY_SIZES_BITS
    {256}
196 #define MAX_ECC_KEY_BITS 256
197 #define MAX_ECC_KEY_BYTES ((MAX_ECC_KEY_BITS + 7) / 8) // 32

```

Table 219 -- AES Algorithm Constants

```

198 #define AES_KEY_SIZES_BITS {128}
199 #define MAX_AES_KEY_BITS 128
200 #define MAX_AES_BLOCK_SIZE_BYTES 16
201 #define MAX_AES_KEY_BYTES ((MAX_AES_KEY_BITS + 7) / 8) // 16

```

Table 220 -- SM4 Algorithm Constants

```

202 #define SM4_KEY_SIZES_BITS {128}
203 #define MAX_SM4_KEY_BITS 128
204 #define MAX_SM4_BLOCK_SIZE_BYTES 16
205 #define MAX_SM4_KEY_BYTES ((MAX_SM4_KEY_BITS + 7) / 8) // 16

```

Table 221 -- Symmetric Algorithm Constants

```

206 #define MAX_SYM_KEY_BITS MAX_AES_KEY_BITS // 128
207 #define MAX_SYM_KEY_BYTES MAX_AES_KEY_BYTES // 16
208 #define MAX_SYM_BLOCK_SIZE MAX_AES_BLOCK_SIZE_BYTES // 16

```

Table 222 -- Implementation Values

```

209 #define FIELD_UPGRADE_IMPLEMENTED NO // 0
210 typedef UINT16 BSIZE;
211 #define BUFFER_ALIGNMENT 4
212 #define IMPLEMENTATION_PCR 24
213 #define PLATFORM_PCR 24
214 #define DRTM_PCR 17
215 #define NUM_LOCALITIES 5
216 #define MAX_HANDLE_NUM 3
217 #define MAX_ACTIVE_SESSIONS 64
218 typedef UINT16 CONTEXT_SLOT;
219 typedef UINT64 CONTEXT_COUNTER;
220 #define MAX_LOADED_SESSIONS 3
221 #define MAX_SESSION_NUM 3
222 #define MAX_LOADED_OBJECTS 3
223 #define MIN_EVICT_OBJECTS 2
224 #define PCR_SELECT_MIN ((PLATFORM_PCR+7)/8) // 3
225 #define PCR_SELECT_MAX ((IMPLEMENTATION_PCR+7)/8) // 3
226 #define NUM_POLICY_PCR_GROUP 1
227 #define NUM_AUTHVALUE_PCR_GROUP 1
228 #define MAX_CONTEXT_SIZE 4000
229 #define MAX_DIGEST_BUFFER 1024
230 #define MAX_NV_INDEX_SIZE 1024
231 #define MAX_CAP_BUFFER 1024
232 #define NV_MEMORY_SIZE 16384
233 #define NUM_STATIC_PCR 16
234 #define MAX_ALG_LIST_SIZE 64
235 #define TIMER_PRESCALE 100000
236 #define PRIMARY_SEED_SIZE 32
237 #define CONTEXT_ENCRYPT_ALG TPM_ALG_AES
238 #define CONTEXT_ENCRYPT_KEY_BITS MAX_SYM_KEY_BITS // 128
239 #define CONTEXT_ENCRYPT_KEY_BYTES ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
240 #define CONTEXT_INTEGRITY_HASH_ALG TPM_ALG_SHA256
241 #define CONTEXT_INTEGRITY_HASH_SIZE SHA256_DIGEST_SIZE // 32
242 #define PROOF_SIZE CONTEXT_INTEGRITY_HASH_SIZE // 32
243 #define NV_CLOCK_UPDATE_INTERVAL 12
244 #define NUM_POLICY_PCR 1
245 #define MAX_COMMAND_SIZE 4096
246 #define MAX_RESPONSE_SIZE 4096
247 #define ORDERLY_BITS 8
248 #define MAX_ORDERLY_COUNT ((1 << ORDERLY_BITS) - 1) <Q>// 255
249 #define ALG_ID_FIRST TPM_ALG_FIRST
250 #define ALG_ID_LAST TPM_ALG_LAST
251 #define MAX_SYM_DATA 128
252 #define MAX_RNG_ENTROPY_SIZE 64
253 #define RAM_INDEX_SPACE 512
254 #define RSA_DEFAULT_PUBLIC_EXPONENT 0x00010001
255 #define ENABLE_PCR_NO_INCREMENT YES // 1
256 #define CRT_FORMAT_RSA YES // 1
257 #define PRIVATE_VENDOR_SPECIFIC_BYTES (\
258     (MAX_RSA_KEY_BYTES/2)*(3+CRT_FORMAT_RSA*2))
259 #define MAX_HASH_BLOCK_SIZE 0
260 #define MAX_DIGEST_SIZE 0
261 #if (SHA1_BLOCK_SIZE * ALG_SHA1) > MAX_HASH_BLOCK_SIZE
262 #undef MAX_HASH_BLOCK_SIZE
263 #define MAX_HASH_BLOCK_SIZE SHA1_BLOCK_SIZE
264 #endif
265 #if (SHA1_DIGEST_SIZE * ALG_SHA1) > MAX_DIGEST_SIZE

```

```

266 #undef MAX_DIGEST_SIZE
267 #define MAX_DIGEST_SIZE SHA1_DIGEST_SIZE
268 #endif
269 #if (SHA256_BLOCK_SIZE * ALG_SHA256) > MAX_HASH_BLOCK_SIZE
270 #undef MAX_HASH_BLOCK_SIZE
271 #define MAX_HASH_BLOCK_SIZE SHA256_BLOCK_SIZE
272 #endif
273 #if (SHA256_DIGEST_SIZE * ALG_SHA256) > MAX_DIGEST_SIZE
274 #undef MAX_DIGEST_SIZE
275 #define MAX_DIGEST_SIZE SHA256_DIGEST_SIZE
276 #endif
277 #if (SHA384_BLOCK_SIZE * ALG_SHA384) > MAX_HASH_BLOCK_SIZE
278 #undef MAX_HASH_BLOCK_SIZE
279 #define MAX_HASH_BLOCK_SIZE SHA384_BLOCK_SIZE
280 #endif
281 #if (SHA384_DIGEST_SIZE * ALG_SHA384) > MAX_DIGEST_SIZE
282 #undef MAX_DIGEST_SIZE
283 #define MAX_DIGEST_SIZE SHA384_DIGEST_SIZE
284 #endif
285 #if (SHA512_BLOCK_SIZE * ALG_SHA512) > MAX_HASH_BLOCK_SIZE
286 #undef MAX_HASH_BLOCK_SIZE
287 #define MAX_HASH_BLOCK_SIZE SHA512_BLOCK_SIZE
288 #endif
289 #if (SHA512_DIGEST_SIZE * ALG_SHA512) > MAX_DIGEST_SIZE
290 #undef MAX_DIGEST_SIZE
291 #define MAX_DIGEST_SIZE SHA512_DIGEST_SIZE
292 #endif
293 #if (SM3_256_BLOCK_SIZE * ALG_SM3_256) > MAX_HASH_BLOCK_SIZE
294 #undef MAX_HASH_BLOCK_SIZE
295 #define MAX_HASH_BLOCK_SIZE SM3_256_BLOCK_SIZE
296 #endif
297 #if (SM3_256_DIGEST_SIZE * ALG_SM3_256) > MAX_DIGEST_SIZE
298 #undef MAX_DIGEST_SIZE
299 #define MAX_DIGEST_SIZE SM3_256_DIGEST_SIZE
300 #endif
301 #define HASH_COUNT (ALG_SHA1+ALG_SHA256+ALG_SHA384+ALG_SHA512+ALG_SM3_256)

```

Part2Parser() Generated (Mar 6, 2013 11:47:26 AM)

Table 7 -- TPM\_ALG\_ID Constants </O,S>

```

302 typedef UINT16 TPM_ALG_ID;
303 #define TPM_ALG_ERROR (TPM_ALG_ID) (0x0000) // a: ; D:
304 #define TPM_ALG_FIRST (TPM_ALG_ID) (0x0001) // a: ; D:
305 #if ALG_RSA == YES || ALG_ALL == YES
306 #define TPM_ALG_RSA (TPM_ALG_ID) (0x0001) // a: A O; D:
307 #endif
308 #if ALG_SHA1 == YES || ALG_ALL == YES
309 #define TPM_ALG_SHA (TPM_ALG_ID) (0x0004) // a: H; D:
310 #endif
311 #if ALG_SHA1 == YES || ALG_ALL == YES
312 #define TPM_ALG_SHA1 (TPM_ALG_ID) (0x0004) // a: H; D:
313 #endif
314 #if ALG_HMAC == YES || ALG_ALL == YES
315 #define TPM_ALG_HMAC (TPM_ALG_ID) (0x0005) // a: H X; D:
316 #endif
317 #if ALG_AES == YES || ALG_ALL == YES
318 #define TPM_ALG_AES (TPM_ALG_ID) (0x0006) // a: S; D:
319 #endif
320 #if ALG_MGF1 == YES || ALG_ALL == YES
321 #define TPM_ALG_MGF1 (TPM_ALG_ID) (0x0007) // a: H M; D:
322 #endif
323 #if ALG_KEYEDHASH == YES || ALG_ALL == YES
324 #define TPM_ALG_KEYEDHASH (TPM_ALG_ID) (0x0008) // a: H E X O; D:
325 #endif
326 #if ALG_XOR == YES || ALG_ALL == YES

```

```

327 #define      TPM_ALG_XOR                (TPM_ALG_ID) (0x000A)           // a: H S; D:
328 #endif
329 #if ALG_SHA256 == YES || ALG_ALL == YES
330 #define      TPM_ALG_SHA256            (TPM_ALG_ID) (0x000B)           // a: H; D:
331 #endif
332 #if ALG_SHA384 == YES || ALG_ALL == YES
333 #define      TPM_ALG_SHA384            (TPM_ALG_ID) (0x000C)           // a: H; D:
334 #endif
335 #if ALG_SHA512 == YES || ALG_ALL == YES
336 #define      TPM_ALG_SHA512            (TPM_ALG_ID) (0x000D)           // a: H; D:
337 #endif
338 #define      TPM_ALG_NULL                (TPM_ALG_ID) (0x0010)           // a: ; D:
339 #if ALG_SM3_256 == YES || ALG_ALL == YES
340 #define      TPM_ALG_SM3_256          (TPM_ALG_ID) (0x0012)           // a: H; D:
341 #endif
342 #if ALG_SM4 == YES || ALG_ALL == YES
343 #define      TPM_ALG_SM4                (TPM_ALG_ID) (0x0013)           // a: S; D:
344 #endif
345 #if ALG_RSASSA == YES || ALG_ALL == YES
346 #define      TPM_ALG_RSASSA            (TPM_ALG_ID) (0x0014)           // a: A X; D: RSA
347 #endif
348 #if ALG_RSAES == YES || ALG_ALL == YES
349 #define      TPM_ALG_RSAES              (TPM_ALG_ID) (0x0015)           // a: A E; D: RSA
350 #endif
351 #if ALG_RSAPSS == YES || ALG_ALL == YES
352 #define      TPM_ALG_RSAPSS            (TPM_ALG_ID) (0x0016)           // a: A X; D: RSA
353 #endif
354 #if ALG_OAEP == YES || ALG_ALL == YES
355 #define      TPM_ALG_OAEP              (TPM_ALG_ID) (0x0017)           // a: A E; D: RSA
356 #endif
357 #if ALG_ECDSA == YES || ALG_ALL == YES
358 #define      TPM_ALG_ECDSA              (TPM_ALG_ID) (0x0018)           // a: A X; D: ECC
359 #endif
360 #if ALG_ECDH == YES || ALG_ALL == YES
361 #define      TPM_ALG_ECDH              (TPM_ALG_ID) (0x0019)           // a: A M; D: ECC
362 #endif
363 #if ALG_ECDAE == YES || ALG_ALL == YES
364 #define      TPM_ALG_ECDAE              (TPM_ALG_ID) (0x001A)           // a: A X; D: ECC
365 #endif
366 #if ALG_SM2 == YES || ALG_ALL == YES
367 #define      TPM_ALG_SM2                (TPM_ALG_ID) (0x001B)           // a: A X E; D: ECC
368 #endif
369 #if ALG_ECSCNORR == YES || ALG_ALL == YES
370 #define      TPM_ALG_ECSCNORR          (TPM_ALG_ID) (0x001C)           // a: A X; D: ECC
371 #endif
372 #if ALG_ECMQV == YES || ALG_ALL == YES
373 #define      TPM_ALG_ECMQV              (TPM_ALG_ID) (0x001D)           // a: A E; D: ECC
374 #endif
375 #if ALG_KDF1_SP800_56a == YES || ALG_ALL == YES
376 #define      TPM_ALG_KDF1_SP800_56a    (TPM_ALG_ID) (0x0020)           // a: H M; D: ECC
377 #endif
378 #if ALG_KDF2 == YES || ALG_ALL == YES
379 #define      TPM_ALG_KDF2              (TPM_ALG_ID) (0x0021)           // a: H M; D:
380 #endif
381 #if ALG_KDF1_SP800_108 == YES || ALG_ALL == YES
382 #define      TPM_ALG_KDF1_SP800_108    (TPM_ALG_ID) (0x0022)           // a: H M; D:
383 #endif
384 #if ALG_ECC == YES || ALG_ALL == YES
385 #define      TPM_ALG_ECC                (TPM_ALG_ID) (0x0023)           // a: A O; D:
386 #endif
387 #if ALG_SYMCIPHER == YES || ALG_ALL == YES
388 #define      TPM_ALG_SYMCIPHER          (TPM_ALG_ID) (0x0025)           // a: O; D:
389 #endif
390 #if ALG_CTR == YES || ALG_ALL == YES
391 #define      TPM_ALG_CTR                (TPM_ALG_ID) (0x0040)           // a: S E; D:
392 #endif

```

```

393 #if ALG_OFB == YES || ALG_ALL == YES
394 #define TPM_ALG_OFB (TPM_ALG_ID) (0x0041) // a: S E; D:
395 #endif
396 #if ALG_CBC == YES || ALG_ALL == YES
397 #define TPM_ALG_CBC (TPM_ALG_ID) (0x0042) // a: S E; D:
398 #endif
399 #if ALG_CFB == YES || ALG_ALL == YES
400 #define TPM_ALG_CFB (TPM_ALG_ID) (0x0043) // a: S E; D:
401 #endif
402 #if ALG_ECB == YES || ALG_ALL == YES
403 #define TPM_ALG_ECB (TPM_ALG_ID) (0x0044) // a: S E; D:
404 #endif
405 #define TPM_ALG_LAST (TPM_ALG_ID) (0x0044) // a: ; D:

```

Table 8 -- TPM\_ECC\_CURVE Constants &lt;/O,S&gt;

```

406 typedef UINT16 TPM_ECC_CURVE;
407 #define TPM_ECC_NONE (TPM_ECC_CURVE) (0x0000)
408 #define TPM_ECC_NIST_P192 (TPM_ECC_CURVE) (0x0001)
409 #define TPM_ECC_NIST_P224 (TPM_ECC_CURVE) (0x0002)
410 #define TPM_ECC_NIST_P256 (TPM_ECC_CURVE) (0x0003)
411 #define TPM_ECC_NIST_P384 (TPM_ECC_CURVE) (0x0004)
412 #define TPM_ECC_NIST_P521 (TPM_ECC_CURVE) (0x0005)
413 #define TPM_ECC_BN_P256 (TPM_ECC_CURVE) (0x0010)
414 #define TPM_ECC_BN_P638 (TPM_ECC_CURVE) (0x0011)
415 #define TPM_ECC_SM2_P256 (TPM_ECC_CURVE) (0x0020)
416 #endif

```

**Annex B**  
(informative)  
**Cryptographic Library Interface**

## B.1 Introduction

The files in this annex 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 this annex, 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 this clause.

There are cases where the cryptographic library could have been used for a specific function but not all functions of the same group. An example is that 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. For example, the RSA key generation schemes produces serviceable RSA keys but the method is not compliant with FIPS 186-3. 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.

## B.2 CryptoEngine.h

### B.2.1. 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 TPM 2.0 specification: Integer values are considered to be an array of one or more bytes. The byte at offset zero within the array is the most significant byte of the integer.

### B.2.2. Defines

```
1 #ifndef CRYPTO_ENGINE_H
2 #define CRYPTO_ENGINE_H
```

This header contains *memcpy* and *memset* functions that are used by functions in the *CryptoEngine()*. The *memcpy* and *memset* functions are easy to implement in a local version that would not use the system string library. However, the *OpenSSL()* code uses these values and, regrettably, the the *OpenSSL()* headers below will pull in all kinds of system libraries. So, rather than try to eliminate use of the standard library implementations of *memcpy* and *memset*, we use the standard ones.

```
3 #include <string.h>
4 #include <openssl/aes.h>
5 #include <openssl/evp.h>
6 #include <openssl/sha.h>
7 #include <openssl/ec.h>
8 #include <openssl/rand.h>
9 #include <openssl/bn.h>
10 #include <openssl/ec_lcl.h>
11 #define ALG_ALL YES
12 #define CRYPTO_ENGINE
13 #include "CryptoBaseTypes.h"
14 #include "CryptPri.h"
15 #include "TpmError.h"
16 #include <<K>bool.h>
17 #include <swap.h>
18 #include <Implementation.h>
19 #include <TPMB.h>
20 #ifndef MAX
21 # define MAX(a, b) ((a) > (b) ? (a) : b)
22 #endif
23 #define MAX_2B_BYTES MAX((MAX_RSA_KEY_BYTES * ALG_RSA), \
24                        MAX((MAX_ECC_PARAMETER_BYTES * ALG_ECC), \
25                            MAX_DIGEST_SIZE))
26 #include "Platform.h"
```

These are structures that can't be shared with *CryptUtil()* This types is used in *CryptoEngine()* to hold a hash state

```
27 typedef BYTE    HASH_STATE_BUFFER[MAX_HASH_STATE_SIZE];
28 #define OSSL_HASH_STATE_DATA_SIZE    (MAX_HASH_STATE_SIZE - 8)
29 typedef struct {
30     union {
31         EVP_MD_CTX    context;
32         BYTE          data[OSSL_HASH_STATE_DATA_SIZE];
33     } u;
34     INT16             copySize;
```



```
35 } OSSL_HASH_STATE;
```

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.

```
36 typedef struct {
37     HASH_STATE_BUFFER      iPadCtx;
38     HASH_STATE_BUFFER      oPadCtx;
39     TPM2B                  *extra;
40     UINT32                  *outer;
41     TPM_ALG_ID              hashAlg;
42     UINT16                  keySizeInBits;
43 } KDFa_CONTEXT;
44 #define assert2Bsize(a) pAssert((a).size <= <K>sizeof((a).buffer))
```

Include the function prototypes when all the types are defined.

```
45 #include <CpriCryptPri_fp.h>
46 #include <MathFunctions_fp.h>
47 #include <CpriRNG_fp.h>
48 #include <CpriHash_fp.h>
49 #include <CpriSym_fp.h>
50 #ifdef TPM_ALG_RSA
51 #   ifdef RSA_KEY_SIEVE
52 #       include "RsaKeySieve.h"
53 #       include "RsaKeySieve_fp.h"
54 #   endif
55 #   include "CpriRSA_fp.h"
56 #endif
57 #ifdef TPM_ALG_ECC
58 #   include "CpriDataEcc.h"
59 #   include "CpriECC_fp.h"
60 #endif
61 #define MAX_ECC_PARAMETER_BYTES 32
62 #endif // CRYPTO_ENGINE_H
```

## 9.18 CryptPri.h

### 9.18.1.1 Introduction

This file contains constant definition shared by *CryptUtil()* and and the parts of the *CryptoEngine()*.

```

1  #ifndef _CRYPT_PRI_H
2  #define _CRYPT_PRI_H
3  #ifndef CRYPTO_ENGINE
4  #include "BaseTypes.h"
5  #endif
6  #include "tpmError.h"
7  #include "swap.h"
8  #include "Implementation.h"
9  #include "TPMB.h"
10 #include "bool.h"
11 #ifndef NULL
12 #define NULL    0
13 #endif
14 typedef UINT16  NUMBYTES;      // When a size is a number of bytes
15 typedef UINT32  NUMDIGITS;     // When a size is a number of "digits"
16 extern  UINT32  g_entropySize;
17 extern  BYTE    g_entropy[];

```

### 9.18.1.2 Hash-related Structures

```

18 typedef struct {
19     const TPM_ALG_ID    alg;
20     const NUMBYTES      digestSize;
21     const NUMBYTES      blockSize;
22     const NUMBYTES      derSize;
23     const BYTE          der[20];
24 } 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 plus the value of the *hashAlg* ID.

```

25 #define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
26 // #define HASH_STATE_SIZE ((MAX_HASH_STATE_SIZE + sizeof(UINT64) -
1) / sizeof(UINT64))

```

This is an array that will hold any of the hash contexts. It is defined as an array of 8-octet values so that the compiler will align the structure.

```

27 typedef UINT64  HASH_STATE_ARRAY[(MAX_HASH_STATE_SIZE + 7)/8];

```

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 8-byte values so that a decent compiler will put the structure on an 8-byte 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.

```

28 typedef struct _HASH_STATE
29 {
30     HASH_STATE_ARRAY    state;
31     TPM_ALG_ID          hashAlg;
32 } CPRI_HASH_STATE, *PCPRI_HASH_STATE;
33 extern const HASH_INFO  g_hashData[HASH_COUNT + 1];

```

### 9.18.1.3 Asymmetric Structures and values

```
34 #ifndef TPM_ALG_ECC
```

#### 9.18.1.4 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

```
35 #include "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.

```
36 typedef struct {
37     UINT32      curveID;           // The curve identifier
38     TPMS_ECC_POINT *publicPoint;  // Pointer to the public point
39     TPM2B      *privateKey;      // Pointer to the private key
40 } ECC_KEY;
41 #endif // TPM_ALG_ECC
42 #ifdef TPM_ALG_RSA
```

#### 9.18.1.5 RSA-related Structures

This structure is a succinct representation of the cryptographic components of an RSA key.

```
43 typedef struct {
44     UINT32      exponent;         // The public exponent pointer
45     TPM2B      *publicKey;       // Pointer to the public modulus
46     TPM2B      *privateKey;     // The private exponent (not a prime)
47 } RSA_KEY;
48 #endif // TPM_ALG_RSA
49 #ifdef TPM_ALG_RSA
50 #   ifdef TPM_ALG_ECC
51 #       if MAX_RSA_KEY_BYTES > MAX_ECC_KEY_BYTES
52 #           define MAX_NUMBER_SIZE      MAX_RSA_KEY_BYTES
53 #       else
54 #           define MAX_NUMBER_SIZE      MAX_ECC_KEY_BYTES
55 #       endif
56 #   else // RSA but no ECC
57 #       define MAX_NUMBER_SIZE          MAX_RSA_KEY_BYTES
58 #   endif
59 #elif defined TPM_ALG_ECC
60 #   define MAX_NUMBER_SIZE              MAX_ECC_KEY_BYTES
61 #else
62 #   error No asymmetric algorithm implemented.
63 #endif
64 typedef INT16      CRYPT_RESULT;
65 #define CRYPT_RESULT_MIN      INT16_MIN
66 #define CRYPT_RESULT_MAX      INT16_MAX
```

< 0	recoverable error
0	success
> 0	command specific return value (generally a digest size)

```
67 #define CRYPT_FAIL            ((CRYPT_RESULT) 1)
68 #define CRYPT_SUCCESS        ((CRYPT_RESULT) 0)
69 #define CRYPT_NO_RESULT      ((CRYPT_RESULT) -1)
70 #define CRYPT_SCHEME         ((CRYPT_RESULT) -2)
```

```
71 #define CRYPT_PARAMETER      ((CRYPT_RESULT) -3)
72 #define CRYPT_UNDERFLOW     ((CRYPT_RESULT) -4)
73 #define CRYPT_POINT         ((CRYPT_RESULT) -5)
74 #define CRYPT_CANCEL        ((CRYPT_RESULT) -6)
75 #define CRYPT_UNEXPECTED    ((CRYPT_RESULT) -7)
76 typedef UINT64              HASH_CONTEXT[MAX_HASH_STATE_SIZE/sizeof(UINT64)];
```

If this is included by a TPM. lib function, then bring in the function prototypes for the crypto engine. Otherwise, defer until the additional *CryptoEngine()* types have been defined.

```
77 #ifndef CRYPTO_ENGINE
78 #include "CpriCryptPri_fp.h"
79 #include "MathFunctions_fp.h"
80 #include "CpriRNG_fp.h"
81 #include "CpriHash_fp.h"
82 #include "CpriSym_fp.h"
83 #ifdef TPM_ALG_RSA
84 #include "CpriRSA_fp.h"
85 #endif
86 #ifdef TPM_ALG_ECC
87 #include "CpriDataEcc.h"
88 #include "CpriECC_fp.h"
89 #endif
90 #endif
91 #endif
```

## 9.19 CryptoBaseTypes.h

```
1 #ifndef _CRYPTO_BASETYPES_H
2 #define _CRYPTO_BASETYPES_H
```

Avoid include of baseTypes.h if this file is included

```
3 #define _BASETYPES_H
4 #include "stdint.h"
5 typedef uint8_t          UINT8;
6 typedef uint8_t          BYTE;
7 typedef int8_t           INT8;
8 typedef int              BOOL;
9 typedef uint16_t         UINT16;
10 typedef int16_t          INT16;
11 typedef uint64_t         UINT64;
12 typedef int64_t          INT64;
13 typedef struct {
14     UINT16      size;
15     BYTE        buffer[1];
16 } TPM2B;
17 #endif
```

### B.3 CpriData.c

This file should be included by the library hash module.

```
1  const HASH_INFO  g_hashData[HASH_COUNT + 1] = {
2  #if  ALG_SHA1 == YES
3      {TPM_ALG_SHA1,    SHA1_DIGEST_SIZE,    SHA1_BLOCK_SIZE,
4       SHA1_DER_SIZE,  SHA1_DER},
5  #endif
6  #if  ALG_SHA256 == YES
7      {TPM_ALG_SHA256,  SHA256_DIGEST_SIZE,  SHA256_BLOCK_SIZE,
8       SHA256_DER_SIZE, SHA256_DER},
9  #endif
10 #if  ALG_SHA384 == YES
11     {TPM_ALG_SHA384,  SHA384_DIGEST_SIZE,  SHA384_BLOCK_SIZE,
12     SHA384_DER_SIZE,  SHA384_DER},
13 #endif
14 #if  ALG_SHA512 == YES
15     {TPM_ALG_SHA512,  SHA512_DIGEST_SIZE,  SHA512_BLOCK_SIZE,
16     SHA512_DER_SIZE,  SHA512_DER},
17 #endif
18 #if  ALG_SM3_256 == YES
19     {TPM_ALG_SM3_256,  SM3_256_DIGEST_SIZE,  SM3_256_BLOCK_SIZE,
20     SM3_256_DER_SIZE,  SM3_256_DER},
21 #endif
22     {TPM_ALG_NULL,0,0,0,{0}}
23 };
```

## B.4 MathFunctions.c

### B.4.1. 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 Part 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 TPM 2.0 specification. It defines an integer as "an array of one or more octets with the most significant octet at the lowest index of the array." An additional value is needed to indicate the number of significant bytes.

```
1 #include "CryptoEngine.h"
```

### B.4.2. Externally Accessible Functions

#### B.4.2.1. `_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.

Return Value	Meaning
0	no significant bytes, value is zero
>0	number of significant bytes

```
2  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 }
```

**B.4.2.2. `_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.

Return Value	Meaning
TRUE	number denormalized
FALSE	number already larger than the desired size

```

17  BOOL
18  _math_Denormalize2B(
19      TPM2B      *in,      // IN:OUT TPM2B number to denormalize
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.4.2.3. `_math__sub()`**

This function to subtract one unsigned value from another  $c = a - b$ .  $c$  may be the same as  $a$  or  $b$ .

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  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
54      UINT16          *cSize,      // OUT: set to MAX(aSize, bSize)
55      BYTE            *c,        // OUT: the difference
56  )

```



```

57 {
58     int          borrow = 0;
59     int          notZero = 0;
60     int          i;
61     int          i2;
62
63     // set c to the longer of a or b
64     *cSize = (UINT16)((aSize > bSize) ? aSize : bSize);
65     // pick the shorter of a and b
66     i = (aSize > bSize) ? bSize : aSize;
67     i2 = *cSize - i;
68     a = &a[aSize - 1];
69     b = &b[bSize - 1];
70     c = &c[*cSize - 1];
71     for(; i > 0; i--)
72     {
73         borrow = *a-- - *b-- + borrow;
74         *c-- = (BYTE)borrow;
75         notZero = notZero || borrow;
76         borrow >>= 8;
77     }
78     if(aSize > bSize)
79     {
80         for(;i2 > 0; i2--)
81         {
82             borrow = *a-- + borrow;
83             *c-- = (BYTE)borrow;
84             notZero = notZero || borrow;
85             borrow >>= 8;
86         }
87     }
88     else if(aSize < bSize)
89     {
90         for(;i2 > 0; i2--)
91         {
92             borrow = 0 - *b-- + borrow;
93             *c-- = (BYTE)borrow;
94             notZero = notZero || borrow;
95             borrow >>= 8;
96         }
97     }
98     // if there is a borrow, then b > a
99     if(borrow)
100         return -1;
101     // either a > b or they are the same
102     return notZero;
103 }

```

#### B.4.2.4. `_math__Inc()`

This function increments a large, big-endian number value by one.

Return Value	Meaning
0	result is zero
!0	result is not zero

```

104 int
105 _math__Inc(
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.4.2.5. `_math__Dec`

This function decrements a large, ENDIAN value by one.

```

118 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.4.2.6. `_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 denormalized 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  $aSize + bSize$ .

Return Value	Meaning
< 0	indicates an error
>= 0	the size of the product

```

131 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 + bSize
139 )
140 {
141     BIGNUM           *bnA;
142     BIGNUM           *bnB;
143     BIGNUM           *bnP;
144     BN_CTX           *context;
145     int               retVal = 0;
146
147
148     // First check that pSize is large enough if present
149     if((pSize != NULL) && (*pSize < (aSize + bSize)))
150         return CRYPT_PARAMETER;
151     pAssert(*pSize < MAX_2B_BYTES);
152     //

```

```

153     // Allocate space for BIGNUM context
154     //
155     context = BN_CTX_new();
156     if(context == NULL)
157         FAIL(FATAL_ERROR_ALLOCATION);
158     bnA = BN_CTX_get(context);
159     bnB = BN_CTX_get(context);
160     bnP = BN_CTX_get(context);
161     if (bnP == NULL)
162         FAIL(FATAL_ERROR_ALLOCATION);
163
164     // Convert the inputs to BIGNUMs
165     //
166     if (BN_bin2bn(a, aSize, bnA) == NULL || BN_bin2bn(b, bSize, bnB) == NULL)
167         FAIL(FATAL_ERROR_INTERNAL);
168
169     // Perform the multiplication
170     //
171     if (BN_mul(bnP, bnA, bnB, context) != 1)
172         FAIL(FATAL_ERROR_INTERNAL);
173
174
175     // If the size of the results is allowed to float, then set the return
176     // size. Otherwise, it might be necessary to denormalize the results
177     retVal = BN_num_bytes(bnP);
178     if(pSize == NULL)
179     {
180         BN_bn2bin(bnP, &p[aSize + bSize - retVal]);
181         memset(p, 0, aSize + bSize - retVal);
182         retVal = aSize + bSize;
183     }
184     else
185     {
186         BN_bn2bin(bnP, p);
187         *pSize = retVal;
188     }
189
190     BN_CTX_end(context);
191     BN_CTX_free(context);
192     return retVal;
193 }

```

#### B.4.2.7. `_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.

Return Value	Meaning
CRYPT_SUCCESS	operation complete
CRYPT_UNDERFLOW	$q$ or $r$ is too small to receive the result

```

194 CRYPT_RESULT
195 _math__Div(
196     const TPM2B      *n,      // IN: numerator
197     const TPM2B      *d,      // IN: denominator
198     TPM2B             *q,      // OUT: quotient
199     TPM2B             *r,      // OUT: remainder
200 )
201 {
202     BIGNUM             *bnN;
203     BIGNUM             *bnD;
204     BIGNUM             *bnQ;

```

```

205     BIGNUM         *bnR;
206     BN_CTX        *context;
207     CRYPT_RESULT   retVal = CRYPT_SUCCESS;
208
209     // Get structures for the big number representations
210     context = BN_CTX_new();
211     if(context == NULL)
212         FAIL(FATAL_ERROR_ALLOCATION);
213     BN_CTX_start(context);
214     bnN = BN_CTX_get(context);
215     bnD = BN_CTX_get(context);
216     bnQ = BN_CTX_get(context);
217     bnR = BN_CTX_get(context);
218
219     // Errors in BN_CTX_get() are sticky so only need to check the last allocation
220     if ( bnR == NULL
221         || BN_bin2bn(n->buffer, n->size, bnN) == NULL
222         || BN_bin2bn(d->buffer, d->size, bnD) == NULL)
223         FAIL(FATAL_ERROR_INTERNAL);
224
225     // Check for divide by zero.
226     if(BN_num_bits(bnD) == 0)
227         FAIL(FATAL_ERROR_DIVIDE_ZERO);
228
229     // Perform the division
230     if (BN_div(bnQ, bnR, bnN, bnD, context) != 1)
231         FAIL(FATAL_ERROR_INTERNAL);
232
233
234     // Convert the BIGNUM result back to our format
235     if(q != NULL) // If the quotient is being returned
236     {
237         if(!BnTo2B(q, bnQ, q->size))
238         {
239             retVal = CRYPT_UNDERFLOW;
240             goto Done;
241         }
242     }
243     if(r != NULL) // If the remainder is being returned
244     {
245         if(!BnTo2B(r, bnR, r->size))
246             retVal = CRYPT_UNDERFLOW;
247     }
248
249 Done:
250     BN_CTX_end(context);
251     BN_CTX_free(context);
252
253     return retVal;
254 }

```

#### B.4.2.8. `_math_uComp()`

This function compare two unsigned values.

Return Value	Meaning
1	if (a > b)
0	if (a = b)
-1	if (a < b)

```

255     int
256     _math_uComp(

```

```

257     const UINT32      aSize,      // IN: size of a
258     const BYTE       *a,         // IN: a
259     const UINT32      bSize,      // IN: size of b
260     const BYTE       *b         // IN: b
261 )
262 {
263     int          borrow = 0;
264     int          notZero = 0;
265     int          i;
266     // If a has more digits than b, then a is greater than b if
267     // any of the more significant bytes is non zero
268     if((i = (int)aSize - (int)bSize) > 0)
269         for(; i > 0; i--)
270             if(*a++) // means a > b
271                 return 1;
272     // If b has more digits than a, then b is greater if any of the
273     // more significant bytes is non zero
274     if(i < 0) <Q>// Means that b is longer than a
275         for(; i < 0; i++)
276             if(*b++) // means that b > a
277                 return -1;
278     // Either the vales are the same size or the upper bytes of a or b are
279     // all zero, so compare the rest
280     i = (aSize > bSize) ? bSize : aSize;
281     a = &a[i-1];
282     b = &b[i-1];
283     for(; i > 0; i--)
284     {
285         borrow = *a-- - *b-- + borrow;
286         notZero = notZero || borrow;
287         borrow >>= 8;
288     }
289     // if there is a borrow, then b > a
290     if(borrow)
291         return -1;
292     // either a > b or they are the same
293     return notZero;
294 }

```

#### B.4.2.9. `_math__Comp()`

Compare two signed integers:

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```

295     int
296     _math__Comp(
297         const UINT32      aSize,      // IN: size of a
298         const BYTE       *a,         // IN: a buffer
299         const UINT32      bSize,      // IN: size of b
300         const BYTE       *b         // IN: b buffer
301     )
302 {
303     int          signA, signB;      // sign of a and b
304
305     // For positive or 0, sign_a is 1
306     // for negative, sign_a is 0
307     signA = ((a[0] & 0x80) == 0) ? 1 : 0;
308 }

```

```

309     // For positive or 0, sign_b is 1
310     // for negative, sign_b is 0
311     signB = ((b[0] & 0x80) == 0) ? 1 : 0;
312
313     if(signA != signB)
314     {
315         return signA - signB;
316     }
317
318     if(signA == 1)
319         // do unsigned compare function
320         return _math_uComp(aSize, a, bSize, b);
321     else
322         // do unsigned compare the other way
323         return 0 - _math_uComp(aSize, a, bSize, b);
324 }

```

#### B.4.2.10. `_math_ModExp`

This function is used to do modular exponentiation in support of RSA. The most typical uses are:  $c = m^e \bmod n$  (RSA encrypt) and  $m = c^d \bmod n$  (RSA decrypt). When doing decryption, the  $e$  parameter of the function will contain the private exponent  $d$  instead of the public exponent  $e$ .

If the results will not fit in the provided buffer, an error is returned (`CRYPT_ERROR_UNDERFLOW`). If the results is smaller than the buffer, the results is de-normalized.

This version is intended for use with RSA and requires that  $m$  be less than  $n$ .

Return Value	Meaning
<code>CRYPT_SUCCESS</code>	exponentiation succeeded
<code>CRYPT_PARAMETER</code>	number to exponentiate is larger than the modulus
<code>CRYPT_UNDERFLOW</code>	result will not fit into the provided buffer

```

325 CRYPT_RESULT
326 _math_ModExp(
327     UINT32      cSize,      // IN: size of the results
328     BYTE       *c,         // OUT: results buffer
329     const UINT32 mSize,    // IN: size of number to be exponentiated
330     const BYTE  *m,        // IN: number to be exponentiated
331     const UINT32 eSize,    // IN: size of power
332     const BYTE  *e,        // IN: power
333     const UINT32 nSize,    // IN: modulus size
334     const BYTE  *n,        // IN: modulus
335 )
336 {
337     CRYPT_RESULT  retVal = CRYPT_SUCCESS;
338     BN_CTX       *context;
339     BIGNUM       *bnC;
340     BIGNUM       *bnM;
341     BIGNUM       *bnE;
342     BIGNUM       *bnN;
343     INT32        i;
344
345     context = BN_CTX_new();
346     if(context == NULL)
347         FAIL(FATAL_ERROR_ALLOCATION);
348     BN_CTX_start(context);
349     bnC = BN_CTX_get(context);
350     bnM = BN_CTX_get(context);
351     bnE = BN_CTX_get(context);
352     bnN = BN_CTX_get(context);
353 }

```

```

354 // Errors for BN_CTX_get are sticky so only need to check last allocation
355 if (bnN == NULL)
356     FAIL(FATAL_ERROR_ALLOCATION);
357
358 //convert arguments
359 if ( BN_bin2bn(m, mSize, bnM) == NULL
360     || BN_bin2bn(e, eSize, bnE) == NULL
361     || BN_bin2bn(n, nSize, bnN) == NULL)
362     FAIL(FATAL_ERROR_INTERNAL);
363
364 // Don't do exponentiation if the number being exponentiated is
365 // larger than the modulus.
366 if (BN_ucmp(bnM, bnN) >= 0)
367 {
368     retVal = CRYPT_PARAMETER;
369     goto Cleanup;
370 }
371 // Perform the exponentiation
372 if (!(BN_mod_exp(bnC, bnM, bnE, bnN, context)))
373     FAIL(FATAL_ERROR_INTERNAL);
374
375 // Convert the results
376 // Make sure that the results will fit in the provided buffer.
377 if ((unsigned)BN_num_bytes(bnC) > cSize)
378 {
379     retVal = CRYPT_UNDERFLOW;
380     goto Cleanup;
381 }
382 i = cSize - BN_num_bytes(bnC);
383 BN_bn2bin(bnC, &c[i]);
384 memset(c, 0, i);
385
386 Cleanup:
387 // Free up allocated BN values
388 BN_CTX_end(context);
389 BN_CTX_free(context);
390 return retVal;
391 }

```

#### B.4.2.11. `_math_IsPrime()`

Check if an integer is probably a prime.

Return Value	Meaning
TRUE	if the integer is probably a prime
FALSE	if the integer is definitely not a prime

```

392 BOOL
393 _math_IsPrime(
394     const UINT32    primeSize,    // IN: prime size
395     const BYTE     *prime        // IN: prime
396 )
397 {
398 #if defined RSA_KEY_SIEVE && (PRIME_DIFF_TABLE_BYTES >= 6542)
399     // The only use of this function is for checking the primality of the
400     // public exponent in _cpri_GenerateKeyRSA. Rather than pull in all the
401     // OpenSSL prime number handling when we have the tables available locally,
402     // we null out this function.
403     pAssert(TRUE);
404     return FALSE;
405 #else
406     int    isPrime;

```

```
407     BIGNUM *p;
408
409     // Assume the size variables are not overflow, which should not happen in
410     // the contexts that this function will be called.
411     pAssert((int) primeSize >= 0);
412     if((p = BN_new()) == NULL)
413         FAIL(FATAL_ERROR_ALLOCATION);
414     if (BN_bin2bn(prime, primeSize, p) == NULL)
415         FAIL(FATAL_ERROR_INTERNAL);
416
417     //
418     // BN_is_prime returning -1 means that it ran into an error.
419     // It should only return 0 or 1
420     //
421     if((isPrime = BN_is_prime_ex(p, BN_prime_checks, NULL, NULL)) < 0)
422         FAIL(FATAL_ERROR_INTERNAL);
423
424     if(p != NULL)
425         BN_clear_free(p);
426     return (isPrime == 1);
427 #endif
428 }
```



## B.5 CpriCryptPri.c

This file contains implementation of crypto primitives. This is a simulator of a 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 TPM 2.0 specification: Integer values are considered to be an array of one or more bytes. The byte at offset zero within the array is the most significant byte of the integer.

```
1  #include "CryptoEngine.h"
```

### B.5.1. Initialization and Shutdown

#### B.5.1.1. \_cpri\_\_InitCryptoUnits()

Initialize crypto units

Return Value	Meaning
--------------	---------

```
2  CRYPT_RESULT
3  _cpri__InitCryptoUnits(void)
4  {
5      _cpri__RngStartup();
6      _cpri__HashStartup();
7      _cpri__SymStartup();
8
9      #ifdef TPM_ALG_RSA
10     _cpri__RsaStartup();
11     #endif
12
13     #ifdef TPM_ALG_ECC
14     _cpri__EccStartup();
15     #endif
16
17     return 0;
18 }
```

#### B.5.1.2. \_cpri\_\_StopCryptoUnits()

Shut down the crypto function units

```
19 void
20 _cpri__StopCryptoUnits(void)
21 {
22     return;
23 }
```

#### B.5.1.3. \_cpri\_\_Startup()

Start the crypto after startup

```
24 BOOL
25 _cpri__Startup(
26     void
27 )
28 {
29
30     return( _cpri__HashStartup()
```

```

31         &&_cpri__RngStartup()
32 #ifndef TPM_ALG_RSA
33         &&_cpri__RsaStartup()
34 #endif // TPM_ALG_RSA
35 #ifndef TPM_ALG_ECC
36         &&_cpri__EccStartup()
37 #endif // TPM_ALG_ECC
38         &&_cpri__SymStartup());
39 }

```

#### B.5.1.4. `_cpri__IncrementalSelfTest()`

This function is used to start an incremental self-test. It always returns success.

NOTE: the behavior in this function is NOT the correct behavior for a real TPM implementation. An artificial behavior is placed here due to the limitation of a software simulation environment. For the correct behavior, consult the part 3 specification for `TPM2_IncrementalSelfTest()`.

```

40 CRYPT_RESULT
41 _cpri__IncrementalSelfTest(
42     TPML_ALG          *toTest,           // IN: list of algorithms to be tested
43     TPML_ALG          *toDoList        // OUT: list of algorithms need test
44 )
45 {
46     // Always copy toTest list to toDoList
47     *toDoList = *toTest;
48
49     return CRYPT_SUCCESS;
50 }

```

### B.5.2. Private Functions

#### B.5.2.1. Introduction

These functions are private to the `CryptoEngine()`. These functions use parameter types that are not known outside of the `CryptoEngine()`.

This file is scanned by a tool that extracts function prototypes. It will put these function in the `OpenSSLCryptPri_fp.h` file. So that these private functions are not visible outside of the `CryptoEngine()`, an `#ifdef` guard is used. The function prototype file will contain these files but they will only have an effect if `CRYPTO_ENGINE_H` is defined. Any file that include `CryptEngine.h` is considered to be part of the `CryptoEngine()`.

```

51 // #ifdef CRYPTO_ENGINE_H

```

#### B.5.2.2. `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. If `size` is zero, then `'outVal->size'` determines how big the result will be. Otherwise, `size` determines the size of the resulting array and `'outVal->size'` is set accordingly.

Return Value	Meaning
0	failure (probably fatal)
1	conversion successful

```

52 BOOL

```

```

53 BnTo2B(
54     TPM2B      *outVal,    // OUT: place for the result
55     BIGNUM     *inVal,     // IN: number to convert
56     UINT16     size       // IN: size of the output.
57 )
58 {
59     BYTE      *pb = outVal->buffer;
60
61     if(size == 0)
62         size = outVal->size;
63     else
64         outVal->size = size;
65
66     size = size - (((UINT16) BN_num_bits(inVal) + 7) / 8);
67     if(size < 0)
68         return FALSE;
69     for(;size > 0; size--)
70         *pb++ = 0;
71     BN_bn2bin(inVal, pb);
72     return TRUE;
73 }

```

### B.5.2.3. 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.

```

74 void
75 Copy2B(
76     TPM2B      *out,       // OUT: The TPM2B to receive the copy
77     TPM2B      *in        // IN: the TPM2B to copy
78 )
79 {
80     BYTE      *pIn = in->buffer;
81     BYTE      *pOut = out->buffer;
82     int       count;
83     out->size = in->size;
84     for(count = in->size; count > 0; count--)
85         *pOut++ = *pIn++;
86     return;
87 }

```

### B.5.2.4. BnFrom2B()

This function creates a BIGNUM from a TPM2B and fails if the conversion fails.

```

88 BIGNUM *
89 BnFrom2B(
90     BIGNUM     *out,       // OUT: The BIGNUM
91     const TPM2B *in        // IN: the TPM2B to copy
92 )
93 {
94     if(BN_bin2bn(in->buffer, in->size, out) == NULL)
95         FAIL(FATAL_ERROR_INTERNAL);
96     return out;
97 }
98 //#endif CRYPTO_ENGINE_H

```

## B.6 CpriRNG.c

```

1  // #define __TPM_RNG_FOR_DEBUG__
2  #include "CryptoEngine.h"
3  #ifndef __TPM_RNG_FOR_DEBUG__ // %
4  TPM2B_TYPE(B64, 64);
5  const TPM2B_B64 randomSeed = {
6      64, "Special version of the RNG to be used only during TPM debug!!!!";
7  static UINT32 rngCounter = 923;
8  BOOL
9  __cpri__RngStartup(void)
10 {
11     memcpy(randomSeed.t.buffer,
12           "Special version of the RNG to be used only during TPM debug!!!!",
13           64);
14     randomSeed.t.size = 64;
15     rngCounter = 923;
16     return TRUE;
17 }

```

### B.6.1.1. \_\_cpri\_\_StirRandom()

Set random entropy

```

18 CRYPT_RESULT
19 __cpri__StirRandom(
20     INT32     entropySize,
21     BYTE      *entropy
22 )
23 {
24     if (entropySize >= 0)
25     {
26         randomSeed.t.size = (entropySize > 64) ? 64 : entropySize;
27         memcpy(randomSeed.t.buffer, entropy, randomSeed.t.size);
28         rngCounter = 0;
29     }
30     return CRYPT_SUCCESS;
31 }
32 }

```

### B.6.1.2. \_\_cpri\_\_GenerateRandom()

Generate a *randomSize* number or random bytes.

```

33 UINT16
34 __cpri__GenerateRandom(
35     INT32     randomSize,
36     BYTE      *buffer
37 )
38 {
39     //
40     // We don't do negative sizes or ones that are too large
41     if (randomSize < 0 || randomSize > UINT16_MAX)
42         return 0;
43     // RAND_bytes uses 1 for success and we use 0
44
45     __cpri__KDFa(TPM_ALG_SHA256,
46                 &randomSeed.b,
47                 "Not really random numbers",
48                 NULL,
49                 NULL,
50                 randomSize * 8,

```

```

51         buffer,
52         &rngCounter,
53         FALSE);
54
55     return randomSize;
56 }
57 #else    //%
```

## B.6.2. Random Number Generation

```

58 BOOL
59 _cpri__RngStartup(void)
60 {
61     UINT32     entropySize;
62     BYTE       entropy[MAX_RNG_ENTROPY_SIZE];
63     // Collect entropy until we have enough
64     for(entropySize = 0;
65        entropySize < MAX_RNG_ENTROPY_SIZE;
66        entropySize += _plat__GetEntropy(&entropy[entropySize],
67                                       MAX_RNG_ENTROPY_SIZE - entropySize));
68     // Seed OpenSSL with entropy
69     RAND_seed(entropy, entropySize);
70     return TRUE;
71 }
```

### B.6.2.1. \_cpri\_\_StirRandom()

Set random entropy

```

72 CRYPT_RESULT
73 _cpri__StirRandom(
74     INT32     entropySize,
75     BYTE      *entropy
76 )
77 {
78     if (entropySize >= 0)
79     {
80         RAND_add((const void *)entropy, (int) entropySize, 0.0);
81     }
82     return CRYPT_SUCCESS;
83 }
84 }
```

### B.6.2.2. \_cpri\_\_GenerateRandom()

Generate a *randomSize* number or random bytes.

```

85 UINT16
86 _cpri__GenerateRandom(
87     INT32     randomSize,
88     BYTE      *buffer
89 )
90 {
91     //
92     // We don't do negative sizes or ones that are too large
93     if (randomSize < 0 || randomSize > UINT16_MAX)
94         return 0;
95     // RAND_bytes uses 1 for success and we use 0
96     if(RAND_bytes(buffer, randomSize) == 1)
97         return (UINT16)randomSize;
98     else
99         return 0;
```

```
100 }  
101 #endif //%
```

## B.7 CpriHash.c

### B.7.1. Description

This file contains implementation of cryptographic functions for hashing.

### B.7.2. Includes, Defines, and Types

```
1 #include "CryptoEngine.h"
```

Temporary aliasing of SM3 to SHA256 until SM3 is available

```
2 #define EVP_sm3_256 EVP_sha256
```

### B.7.3. Static Functions

#### B.7.3.1. GetHashServer()

This function returns the address of the hash server function

```
3 static EVP_MD *
4 GetHashServer(
5     TPM_ALG_ID hashAlg
6 )
7 {
8     switch (hashAlg)
9     {
10 #ifdef TPM_ALG_SHA1
11     case TPM_ALG_SHA1:
12         return (EVP_MD *)EVP_sha1();
13         break;
14 #endif
15 #ifdef TPM_ALG_SHA256
16     case TPM_ALG_SHA256:
17         return (EVP_MD *)EVP_sha256();
18         break;
19 #endif
20 #ifdef TPM_ALG_SHA384
21     case TPM_ALG_SHA384:
22         return (EVP_MD *)EVP_sha384();
23         break;
24 #endif
25 #ifdef TPM_ALG_SHA512
26     case TPM_ALG_SHA512:
27         return (EVP_MD *)EVP_sha512();
28         break;
29 #endif
30 #ifdef TPM_ALG_SM3_256
31     case TPM_ALG_SM3_256:
32         return (EVP_MD *)EVP_sm3_256();
33         break;
34 #endif
35     case TPM_ALG_NULL:
36         return NULL;
37     default:
38         FAIL(FATAL_ERROR_INTERNAL);
39         return NULL;
40     }
41 }
```

**B.7.3.2. MarshalHashState()**

This function copies an *OpenSSL()* hash context into a caller provided buffer.

Return Value	Meaning
> 0	the number of bytes of buf used.

```

42  static UINT16
43  MarshalHashState(
44      EVP_MD_CTX      *ctx,           // IN: Context to marshal
45      BYTE            *buf,           // OUT: The buffer that will receive the
46                                     // context. This buffer is at least
47                                     // MAX_HASH_STATE_SIZE bytes
48  )
49  {
50      // make sure everything will fit
51      pAssert(ctx->digest->ctx_size <= ( MAX_HASH_STATE_SIZE
52                                          - sizeof(INT16)
53                                          - sizeof(TPM_ALG_ID)));
54
55      // Copy the context data
56      memcpy(buf, (void*) ctx->md_data, ctx->digest->ctx_size);
57
58      return (UINT16)ctx->digest->ctx_size;
59  }

```

**B.7.3.3. 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).

```

60  static UINT16
61  GetHashState(
62      EVP_MD_CTX      *ctx,           // OUT: The context structure to receive
63                                     // the result of unmarshaling.
64      TPM_ALG_ID      algType,       // IN: The hash algorithm selector
65      BYTE            *buf,           // IN: Buffer containing marshaled hash data
66  )
67  {
68      EVP_MD          *evpmdAlgorithm = NULL;
69
70      pAssert(ctx != NULL);
71
72      EVP_MD_CTX_init(ctx);
73      evpmdAlgorithm = GetHashServer(algType);
74      if(evpmdAlgorithm == NULL)
75          return 0;
76
77      // This also allocates the ctx->md_data
78      if((EVP_DigestInit_ex(ctx, evpmdAlgorithm, NULL)) != 1)
79          FAIL(FATAL_ERROR_INTERNAL);
80
81      memcpy(ctx->md_data, buf, ctx->digest->ctx_size);
82      return (UINT16)ctx->digest->ctx_size;
83  }

```

**B.7.3.4. GetHashInfoPointer()**

This function returns a pointer to the hash info for the algorithm. If the algorithm is not supported a pointer to an error block is returned.



```

84  static const HASH_INFO *
85  GetHashInfoPointer(
86      TPM_ALG_ID  hashAlg
87  )
88  {
89      UINT32 i, tableSize;
90
91      // Get the table size of g_hashData
92      tableSize = sizeof(g_hashData) / sizeof(g_hashData[0]);
93
94      for(i = 0; i < tableSize - 1; i++)
95      {
96          if(g_hashData[i].alg == hashAlg)
97              return &g_hashData[i];
98      }
99      pAssert(hashAlg == TPM_ALG_NULL);
100     return &g_hashData[tableSize-1];
101 }

```

## B.7.4. Hash Functions

### B.7.4.1. \_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.

```

102  BOOL
103  _cpri__HashStartup(
104      void
105  )
106  {
107      return TRUE;
108  }

```

### B.7.4.2. \_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 an *index* of 2 will return the last. All other index values will return TPM\_ALG\_NULL.

Return Value	Meaning
TPM_ALG_XXX ()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```

109  TPM_ALG_ID
110  _cpri__GetHashAlgByIndex(
111      UINT32  index  // IN: the index
112  )
113  {
114      if(index >= HASH_COUNT)
115          return TPM_ALG_NULL;
116      return g_hashData[index].alg;
117  }

```

### B.7.4.3. \_cpri\_\_GetHashBlockSize()

Returns the size of the block used for the hash

Return Value	Meaning
< 0	the algorithm is not a supported hash
>=	the digest size (0 for TPM_ALG_NULL)

```

118  UINT16
119  __cpri_GetHashBlockSize(
120      TPM_ALG_ID  hashAlg    // IN: hash algorithm to look up
121  )
122  {
123      return GetHashInfoPointer(hashAlg)->blockSize;
124  }

```

#### B.7.4.4. \_\_cpri\_\_GetHashDER

This function returns a pointer to the DER string for the algorithm and indicates its size.

```

125  UINT16
126  __cpri_GetHashDER(
127      TPM_ALG_ID  hashAlg,    // IN: the algorithm to look up
128      const BYTE  **p
129  )
130  {
131      const HASH_INFO  *q;
132      q = GetHashInfoPointer(hashAlg);
133      *p = &q->der[0];
134      return q->derSize;
135  }

```

#### B.7.4.5. \_\_cpri\_\_GetDigestSize()

Gets the digest size of the algorithm. The algorithm is required to be supported.

Return Value	Meaning
=0	the digest size for TPM_ALG_NULL
>0	the digest size of a hash algorithm

```

136  UINT16
137  __cpri_GetDigestSize(
138      TPM_ALG_ID  hashAlg    // IN: hash algorithm to look up
139  )
140  {
141      return GetHashInfoPointer(hashAlg)->digestSize;
142  }

```

#### B.7.4.6. \_\_cpri\_\_GetContextAlg()

This function returns the algorithm associated with a hash context

```

143  TPM_ALG_ID
144  __cpri_GetContextAlg(
145      void  *hashState // IN: the hash context
146  )
147  {
148      CPRI_HASH_STATE  *state = (CPRI_HASH_STATE *)hashState;
149      return state->hashAlg;
150  }

```

**B.7.4.7. `_cpri_CopyHashState`**

This function is used to **clone** a `CPRI_HASH_STATE`. The return value is the size of the state.

```

151  UINT16
152  _cpri_CopyHashState (
153      void    *out,           // OUT: destination of the state
154      void    *in            // IN: source of the state
155  )
156  {
157      CPRI_HASH_STATE    *i = (CPRI_HASH_STATE *)in;
158      CPRI_HASH_STATE    *o = (CPRI_HASH_STATE *)out;
159      EVP_MD_CTX_init(&o->u.context);
160      EVP_MD_CTX_copy_ex(&o->u.context, &i->u.context);
161      o->size = i->size;
162      o->hashAlg = i->hashAlg;
163      return sizeof(CPRI_HASH_STATE);
164  }

```

**B.7.4.8. `_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.

Return Value	Meaning
0	hash is <code>TPM_ALG_NULL</code>
>0	digest size

```

165  UINT16
166  _cpri_StartHash(
167      TPM_ALG_ID    hashAlg,           // IN: hash algorithm
168      BOOL          sequence,         // IN: TRUE if the state should be saved
169      void          *hashState        // OUT: the state of hash stack.
170                                          // This buffer will be used in
171                                          // hash update and completion. Caller
172                                          // should allocate this buffer with size of
173                                          // MAX_HASH_STATE_SIZE
174  )
175  {
176      EVP_MD_CTX      localState;
177      CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
178      EVP_MD_CTX      *context;
179      EVP_MD          *evpmdAlgorithm = NULL;
180      UINT16          retVal = 0;
181
182      if(sequence)
183          context = &localState;
184      else
185          context = &state->u.context;
186
187      state->hashAlg = hashAlg;
188
189      EVP_MD_CTX_init(context);
190      evpmdAlgorithm = GetHashServer(hashAlg);
191      if(evpmdAlgorithm == NULL)
192          goto Cleanup;
193
194      if(EVP_DigestInit_ex(context, evpmdAlgorithm, NULL) != 1)
195          FAIL(FATAL_ERROR_INTERNAL);

```

```

196     retVal = (CRYPT_RESULT)EVP_MD_CTX_size(context);
197
198 Cleanup:
199     if(retVal > 0)
200     {
201         if (sequence)
202         {
203             if((state->size = MarshalHashState(context, state->u.data)) == 0)
204             {
205                 // If MarshalHashState returns a negative number, it is an error
206                 // code and not a hash size so copy the error code to be the return
207                 // from this function and set the actual stateSize to zero.
208                 retVal = state->size;
209                 state->size = 0;
210             }
211             // Do the cleanup
212             EVP_MD_CTX_cleanup(context);
213         }
214         else
215             state->size = -1;
216     }
217     else
218         state->size = 0;
219     return retVal;
220 }

```

#### B.7.4.9. `_cpri__UpdateHash()`

Add data to a hash or HMAC stack.

```

221 void
222 _cpri__UpdateHash(
223     void      *hashState,    // IN: the hash context information
224     UINT32    dataSize,     // IN: the size of data to be added to the digest
225     BYTE      *data,        // IN: data to be hashed
226 )
227 {
228     EVP_MD_CTX      localContext;
229     CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
230     EVP_MD_CTX      *context;
231     CRYPT_RESULT    retVal = CRYPT_SUCCESS;
232
233     if(state->size == 0)
234         return;
235     if(state->size > 0)
236     {
237         context = &localContext;
238         if((retVal = GetHashState(context, state->hashAlg, state->u.data)) <= 0)
239             return;
240     }
241     else
242         context = &state->u.context;
243
244     if(EVP_DigestUpdate(context, data, dataSize) != 1)
245         FAIL(FATAL_ERROR_INTERNAL);
246     else if( state->size > 0
247             && (retVal= MarshalHashState(context, state->u.data)) >= 0)
248     {
249         // retVal is the size of the marshaled data. Make sure that it is consistent
250         // by ensuring that we didn't get more than allowed
251         if(retVal < state->size)
252             FAIL(FATAL_ERROR_INTERNAL);
253         else
254             EVP_MD_CTX_cleanup(context);

```

```

255     }
256     return;
257 }

```

#### B.7.4.10. `_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  $\leq 0$ .

Return Value	Meaning
0	no data returned
> 0	the number of bytes in the digest

```

258     UINT16
259     _cpri__CompleteHash(
260         void      *hashState,      // IN: the state of hash stack
261         UINT32    dOutSize,        // IN: size of digest buffer
262         BYTE      *dOut            // OUT: hash digest
263     )
264     {
265         EVP_MD_CTX      localState;
266         CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
267         EVP_MD_CTX      *context;
268         UINT16           retVal;
269         int              hLen;
270         BYTE             temp[MAX_DIGEST_SIZE];
271         BYTE             *rBuffer = dOut;
272
273         if(state->size == 0)
274             return 0;
275         if(state->size > 0)
276         {
277             context = &localState;
278             if((retVal = GetHashState(context, state->hashAlg, state->u.data)) <= 0)
279                 goto Cleanup;
280         }
281         else
282             context = &state->u.context;
283
284         hLen = EVP_MD_CTX_size(context);
285         if((unsigned)hLen > dOutSize)
286             rBuffer = temp;
287         if(EVP_DigestFinal_ex(context, rBuffer, NULL) == 1)
288         {
289             if(rBuffer != dOut)
290             {
291                 if(dOut != NULL)
292                 {
293                     memcpy(dOut, temp, dOutSize);
294                 }
295                 retVal = (UINT16)dOutSize;
296             }
297             else
298             {
299                 retVal = (UINT16)hLen;
300             }
301             state->size = 0;
302         }
303         else
304         {
305             retVal = 0; // Indicate that no data is returned

```

```

306     }
307 Cleanup:
308     EVP_MD_CTX_cleanup(context);
309     return retVal;
310 }

```

#### B.7.4.11. `_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.

Return Value	Meaning
>= 0	number of bytes in <i>digest</i> (may be zero)

```

311 UINT16
312 _cpri__HashBlock(
313     TPM_ALG_ID hashAlg,          // IN: The hash algorithm
314     UINT32     dataSize,        // IN: size of buffer to hash
315     BYTE       *data,           // IN: the buffer to hash
316     UINT32     digestSize,     // IN: size of the digest buffer
317     BYTE       *digest          // OUT: hash digest
318 )
319 {
320     EVP_MD_CTX     hashContext;
321     EVP_MD         *hashServer = NULL;
322     UINT16         retVal = 0;
323     BYTE           b[MAX_DIGEST_SIZE]; // temp buffer in case digestSize not
324     // a full digest
325     unsigned int   dSize = _cpri__GetDigestSize(hashAlg);
326
327
328     // If there is no digest to compute return
329     if(dSize == 0)
330         return 0;
331
332     // After the call to EVP_MD_CTX_init(), will need to call EVP_MD_CTX_cleanup()
333     EVP_MD_CTX_init(&hashContext); // Initialize the local hash context
334     hashServer = GetHashServer(hashAlg); // Find the hash server
335
336     // It is an error if the digest size is non-zero but there is no server
337     if( (hashServer == NULL)
338        || (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
339        || (EVP_DigestUpdate(&hashContext, data, dataSize) != 1))
340         FAIL(FATAL_ERROR_INTERNAL);
341     else
342     {
343         // If the size of the digest produced (dSize) is larger than the available
344         // buffer (digestSize), then put the digest in a temp buffer and only copy
345         // the most significant part into the available buffer.
346         if(dSize > digestSize)
347         {
348             if(EVP_DigestFinal_ex(&hashContext, b, &dSize) != 1)
349                 FAIL(FATAL_ERROR_INTERNAL);
350             memcpy(digest, b, digestSize);
351             retVal = (UINT16)digestSize;
352         }
353     else
354     {
355         if((EVP_DigestFinal_ex(&hashContext, digest, &dSize)) != 1)
356             FAIL(FATAL_ERROR_INTERNAL);
357         retVal = (UINT16) dSize;

```

```

358     }
359 }
360     EVP_MD_CTX_cleanup(&hashContext);
361     return retVal;
362 }

```

## B.7.5. HMAC Functions

### B.7.5.1. `_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*.

Return Value	Meaning
>= 0	number of bytes in digest produced by <i>hashAlg</i> (may be zero)

```

363     UINT16
364     _cpri__StartHMAC(
365         TPM_ALG_ID    hashAlg,    // IN: the algorithm to use
366         BOOL          sequence,   // IN: indicates if the state should be saved
367         void          *state,     // IN/OUT: the state buffer
368         UINT16        keySize,    // IN: the size of the HMAC key
369         BYTE          *key,       // IN: the HMAC key
370         TPM2B         *oPadKey    // OUT: the key prepared for the oPad round
371     )
372 {
373     CPRI_HASH_STATE  localState;
374     UINT16           blockSize = _cpri__GetHashBlockSize(hashAlg);
375     UINT16           digestSize;
376     BYTE             *pb;        // temp pointer
377     UINT32           i;
378
379
380     if(keySize > blockSize)
381     {
382         if((digestSize = _cpri__StartHash(hashAlg, FALSE, &localState)) == 0)
383             return 0;
384         _cpri__UpdateHash(&localState, keySize, key);
385         _cpri__CompleteHash(&state, digestSize, oPadKey->buffer);
386         oPadKey->size = digestSize;
387     }
388     else
389     {
390         memcpy(oPadKey->buffer, key, keySize);
391         oPadKey->size = keySize;
392     }
393     // XOR the key with iPad (0x36)
394     pb = oPadKey->buffer;
395     for(i = oPadKey->size; i > 0; i--)
396         *pb++ ^= 0x36;
397
398     // if the keySize is smaller than a block, fill the rest with 0x36
399     for(i = blockSize - oPadKey->size; i > 0; i--)
400         *pb++ = 0x36;
401
402     // Increase the oPadSize to a full block
403     oPadKey->size = blockSize;
404
405     // Start a new hash with the HMAC key
406     // This will go in the caller's state structure and may be a sequence or not

```

```

407
408     if((digestSize = _cpri__StartHash(hashAlg, sequence, state)) > 0)
409     {
410
411         _cpri__UpdateHash(state, oPadKey->size, oPadKey->buffer);
412
413         // XOR the key block with 0x5c ^ 0x36
414         for(pb = oPadKey->buffer, i = blockSize; i > 0; i--)
415             *pb++ ^= (0x5c ^ 0x36);
416     }
417
418     return digestSize;
419 }

```

### B.7.5.2. \_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.

Return Value	Meaning
>= 0	number of bytes in <i>dOut</i> (may be zero)

```

420     UINT16
421     _cpri__CompleteHMAC(
422         void                *hashState,        // IN: the state of hash stack
423         TPM2B               *oPadKey,         // IN: the HMAC key in oPad format
424         UINT32              dOutSize,         // IN: size of digest buffer
425         BYTE                *dOut            // OUT: hash digest
426     )
427     {
428         BYTE                digest[MAX_DIGEST_SIZE];
429         CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
430         CPRI_HASH_STATE  localState;
431         UINT16            digestSize = _cpri__GetDigestSize(state->hashAlg);
432
433
434         _cpri__CompleteHash(hashState, digestSize, digest);
435
436         // Using the local hash state, do a hash with the oPad
437         if(_cpri__StartHash(state->hashAlg, FALSE, &localState) != digestSize)
438             return 0;
439
440         _cpri__UpdateHash(&localState, oPadKey->size, oPadKey->buffer);
441         _cpri__UpdateHash(&localState, digestSize, digest);
442         return _cpri__CompleteHash(&localState, dOutSize, dOut);
443     }

```

## B.7.6. Mask and Key Generation Functions

### B.7.6.1. \_crypi\_MGF1()

This function performs MGF1 using the selected hash. MGF1 is  $T(n) = T(n-1) || H(\text{seed} || \text{counter})$ . This function returns the length of the mask produced which could be zero if the digest algorithm is not supported



Return Value	Meaning
0	hash algorithm not supported
> 0	should be the same as <i>mSize</i>

```

444  CRYPT_RESULT
445  _cpri_MGF1(
446      UINT32      mSize,      // IN: length of the mask to be produced
447      BYTE        *mask,      // OUT: buffer to receive the mask
448      TPM_ALG_ID  hashAlg,    // IN: hash to use
449      UINT32      sSize,      // IN: size of the seed
450      BYTE        *seed       // IN: seed size
451  )
452  {
453      EVP_MD_CTX      hashContext;
454      EVP_MD          *hashServer = NULL;
455      CRYPT_RESULT    retVal = 0;
456      BYTE            b[MAX_DIGEST_SIZE]; // temp buffer in case mask is not an
457      // even multiple of a full digest
458      CRYPT_RESULT    dSize = _cpri_GetDigestSize(hashAlg);
459      unsigned int    digestSize = (UINT32)dSize;
460      UINT32          remaining;
461      UINT32          counter;
462      BYTE            swappedCounter[4];
463
464      // Parameter check
465      if(mSize > (1024*16)) // Semi-arbitrary maximum
466          FAIL(FATAL_ERROR_INTERNAL);
467
468      // If there is no digest to compute return
469      if(dSize <= 0)
470          return 0;
471
472      EVP_MD_CTX_init(&hashContext); // Initialize the local hash context
473      hashServer = GetHashServer(hashAlg); // Find the hash server
474      if(hashServer == NULL)
475          // If there is no server, then there is no digest
476          return 0;
477
478      for(counter = 0, remaining = mSize; remaining > 0; counter++)
479      {
480          // Because the system may be either Endian...
481          UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
482
483          // Start the hash and include the seed and counter
484          if( (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
485             || (EVP_DigestUpdate(&hashContext, seed, sSize) != 1)
486             || (EVP_DigestUpdate(&hashContext, swappedCounter, 4) != 1)
487             )
488              FAIL(FATAL_ERROR_INTERNAL);
489
490          // Handling the completion depends on how much space remains in the mask
491          // buffer. If it can hold the entire digest, put it there. If not
492          // put the digest in a temp buffer and only copy the amount that
493          // will fit into the mask buffer.
494          if(remaining < (<K>unsigned)dSize)
495          {
496              if(EVP_DigestFinal_ex(&hashContext, b, &digestSize) != 1)
497                  FAIL(FATAL_ERROR_INTERNAL);
498              memcpy(mask, b, remaining);
499              break;
500          }
501          else
502          {

```

```

503         if(EVP_DigestFinal_ex(&hashContext, mask, &digestSize) != 1)
504             FAIL(FATAL_ERROR_INTERNAL);
505         remaining -= dSize;
506         mask = &mask[dSize];
507     }
508     retVal = (CRYPTO_RESULT)mSize;
509 }
510
511 EVP_MD_CTX_cleanup(&hashContext);
512 return retVal;
513 }

```

### B.7.6.2. `_cpri_KDFa()`

This function performs the key generation according to Part 1 of the TPM specification.

This function returns the number of bytes generated which may be zero.

The *key* and *keyStream* pointers are not allowed to be NULL. The other pointer values may be NULL. The value of *sizeInBits* must be no larger than  $(2^{18})-1 = 256\text{K bits}$  (32385 bytes).

The **once** parameter is set to allow incremental generation of a large value. If this flag is TRUE, **sizeInBits** will be used in the HMAC computation but only one iteration of the KDF is performed. This would be used for XOR obfuscation so that the mask value can be generated in digest-sized chunks rather than having to be generated all at once in an arbitrarily large buffer and then *XORed()* into the result. If **once** is TRUE, then **sizeInBits** must be a multiple of 8.

Any error in the processing of this command is considered fatal.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the <i>keyStream</i> buffer

```

514 UINT16
515 _cpri_KDFa(
516     TPM_ALG_ID    hashAlg,           // IN: hash algorithm used in HMAC
517     TPM2B         *key,             // IN: HMAC key
518     const char    *label,          // IN: a 0-byte terminated label used in KDF
519     TPM2B         *contextU,       // IN: context U
520     TPM2B         *contextV,       // IN: context V
521     UINT32        sizeInBits,      // IN: size of generated key in bits
522     BYTE          *keyStream,      // OUT: key buffer
523     UINT32        *counterInOut,    // IN/OUT: caller may provide the iteration counter
524                                     //         for incremental operations to avoid
525                                     //         large intermediate buffers.
526     BOOL          once             // IN: TRUE if only one iteration is performed
527                                     //         FALSE if iteration count determined by
528                                     //         "sizeInBits"
529 )
530 {
531     UINT32        counter = 0;      // counter value
532     INT32         lLen;           // length of the label
533     INT16         hLen;           // length of the hash
534     INT16         bytes;         // number of bytes to produce
535     BYTE          *stream = keyStream;
536     BYTE          marshaledUint32[4];
537     BYTE          hashState[MAX_HASH_STATE_SIZE];
538     TPM2B_MAX_HASH_BLOCK hmacKey;
539
540     pAssert(key != NULL && keyStream != NULL);
541     pAssert(once == FALSE || (sizeInBits & 7) == 0);
542

```

```

543     if(counterInOut != NULL)
544         counter = *counterInOut;
545
546     // Prepare label buffer. Calculate its size and keep the last 0 byte
547     for(lLen = 0; label[lLen++] != 0; );
548
549     // Get the hash size. If it is less than or 0, either the
550     // algorithm is not supported or the hash is TPM_ALG_NULL
551     // In either case the digest size is zero. This is the only return
552     // other than the one at the end. All other exits from this function
553     // are fatal errors. After we check that the algorithm is supported
554     // anything else that goes wrong is an implementation flaw.
555     if((hLen = (INT16) _cpri__GetDigestSize(hashAlg)) == 0)
556         return 0;
557
558     // If the size of the request is larger than the numbers will handle,
559     // it is a fatal error.
560     pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);
561
562     bytes = once ? hLen : (INT16)((sizeInBits + 7) / 8);
563
564     // Generate required bytes
565     for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
566     {
567         if(bytes < hLen)
568             hLen = bytes;
569
570         counter++;
571         // Start HMAC
572         if(_cpri__StartHMAC(hashAlg,
573                             FALSE,
574                             &hashState,
575                             key->size,
576                             &key->buffer[0],
577                             &hmacKey.b) <= 0)
578             FAIL(FATAL_ERROR_INTERNAL);
579
580         // Adding counter
581         UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
582         _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
583
584         // Adding label
585         if(label != NULL)
586             _cpri__UpdateHash(&hashState, lLen, (BYTE *)label);
587
588         // Adding contextU
589         if(contextU != NULL)
590             _cpri__UpdateHash(&hashState, contextU->size, contextU->buffer);
591
592         // Adding contextV
593         if(contextV != NULL)
594             _cpri__UpdateHash(&hashState, contextV->size, contextV->buffer);
595
596         // Adding size in bits
597         UINT32_TO_BYTE_ARRAY(sizeInBits, marshaledUint32);
598         _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
599
600         // Compute HMAC. At the start of each iteration, hLen is set
601         // to the smaller of hLen and bytes. This causes bytes to decrement
602         // exactly to zero to complete the loop
603         _cpri__CompleteHMAC(&hashState, &hmacKey.b, hLen, stream);
604     }
605
606     // Mask off bits if the required bits is not a multiple of byte size
607     if((sizeInBits % 8) != 0)
608         keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);

```

```

609     if(counterInOut != NULL)
610         *counterInOut = counter;
611     return (CRYPT_RESULT)((sizeInBits + 7)/8);
612 }

```

### B.7.6.3. `_cpri_KDFe()`

`KDFe()` as defined in TPM specification part 1.

This function returns the number of bytes generated which may be zero.

The `Z` and `keyStream` pointers are not allowed to be NULL. The other pointer values may be NULL. The value of `sizeInBits` must be no larger than  $(2^{18})-1 = 256\text{K bits}$  (32385 bytes). Any error in the processing of this command is considered fatal.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the <code>keyStream</code> buffer

```

613     UINT16
614     _cpri_KDFe(
615         TPM_ALG_ID    hashAlg,           // IN: hash algorithm used in HMAC
616         TPM2B         *Z,               // IN: Z
617         const char    *label,           // IN: a 0-byte terminated label using in KDF
618         TPM2B         *partyUInfo,      // IN: PartyUInfo
619         TPM2B         *partyVInfo,      // IN: PartyVInfo
620         UINT32        sizeInBits,       // IN: size of generated key in bits
621         BYTE          *keyStream        // OUT: key buffer
622     )
623     {
624         UINT32        counter = 0;      // counter value
625         UINT32        lSize = 0;
626         BYTE          *stream = keyStream;
627         BYTE          hashState[MAX_HASH_STATE_SIZE];
628         INT16         hLen = (INT16) _cpri_GetDigestSize(hashAlg);
629         INT16         bytes;           // number of bytes to generate
630         BYTE          marshaledUint32[4];
631
632         pAssert(    keyStream != NULL
633                   && Z != NULL
634                   && ((sizeInBits + 7) / 8) < INT16_MAX);
635
636         if(hLen == 0)
637             return 0;
638
639         bytes = (INT16)((sizeInBits + 7) / 8);
640
641         // Prepare label buffer. Calculate its size and keep the last 0 byte
642         if(label != NULL)
643             for(lSize = 0; label[lSize++] != 0;);
644
645         // Generate required bytes
646         //The inner loop of that KDF uses:
647         // Hashi := H(counter | Z | OtherInfo) (5)
648         // Where:
649         // Hashi   the hash generated on the i-th iteration of the loop.
650         // H()     an approved hash function
651         // counter a 32-bit counter that is initialized to 1 and incremented
652         //         on each iteration
653         // Z       the X coordinate of the product of a public ECC key and a
654         //         different private ECC key.
655         // OtherInfo a collection of qualifying data for the KDF defined below.
656         // In this specification, OtherInfo will be constructed by:

```

```
657 // OtherInfo := Use | PartyUInfo | PartyVInfo
658 for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
659 {
660     if(bytes < hLen)
661         hLen = bytes;
662
663     counter++;
664     // Start hash
665     if(_cpri__StartHash(hashAlg, FALSE, &hashState) == 0)
666         return 0;
667
668     // Add counter
669     UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
670     _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
671
672     // Add Z
673     if(Z != NULL)
674         _cpri__UpdateHash(&hashState, Z->size, Z->buffer);
675
676     // Add label
677     if(label != NULL)
678         _cpri__UpdateHash(&hashState, lSize, (BYTE *)label);
679     else
680
681         // The SP800-108 specification requires a zero between the label
682         // and the context.
683         _cpri__UpdateHash(&hashState, 1, (BYTE *)"");
684
685     // Add PartyUInfo
686     if(partyUInfo != NULL)
687         _cpri__UpdateHash(&hashState, partyUInfo->size, partyUInfo->buffer);
688
689     // Add PartyVInfo
690     if(partyVInfo != NULL)
691         _cpri__UpdateHash(&hashState, partyVInfo->size, partyVInfo->buffer);
692
693     // Compute Hash. hLen was changed to be the smaller of bytes or hLen
694     // at the start of each iteration.
695     _cpri__CompleteHash(&hashState, hLen, stream);
696 }
697
698 // Mask off bits if the required bits is not a multiple of byte size
699 if((sizeInBits % 8) != 0)
700     keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
701
702 return (CRYPT_RESULT)((sizeInBits + 7) / 8);
703
704 }
```

## B.8 CpriSym.c

### B.8.1. 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 *OpenSSL*().

Currently, this module only supports AES encryption. The SM4 code actually calls an AES routine

### B.8.2. Includes, Defines, and Typedefs

```
1 #include "CryptoEngine.h"
```

The following sets of defines are used to allow use of the SM4 algorithm identifier while waiting for the SM4 implementation code to appear.

```
2 typedef AES_KEY SM4_KEY;
3 #define SM4_set_encrypt_key    AES_set_encrypt_key
4 #define SM4_set_decrypt_key   AES_set_decrypt_key
5 #define SM4_decrypt           AES_decrypt
6 #define SM4_encrypt           AES_encrypt
```

### B.8.3. Utility Functions

#### B.8.3.1. \_cpri\_SymStartup()

```
7 BOOL
8 _cpri_SymStartup(
9     void
10 )
11 {
12     return TRUE;
13 }
```

#### B.8.3.2. \_cpri\_GetSymmetricBlockSize()

This function returns the block size of the algorithm.

Return Value	Meaning
<= 0	cipher not supported
> 0	the cipher block size in bytes

```
14 INT16
15 _cpri_GetSymmetricBlockSize(
16     TPM_ALG_ID    symmetricAlg, // IN: the symmetric algorithm
17     UINT16        keySizeInBits // IN: the key size
18 )
19 {
20     switch (symmetricAlg)
21     {
22 #ifdef TPM_ALG_AES
23     case TPM_ALG_AES:
24 #endif
25 #ifdef TPM_ALG_SM4 // Both AES and SM4 use the same block size
26     case TPM_ALG_SM4:
27 #endif
28         if(keySizeInBits != 0) // This is mostly to have a reference to
```

```

29         // keySizeInBits for the compiler
30         return 16;
31     else
32         return 0;
33     break;
34
35     default:
36         return 0;
37     }
38 }

```

## B.8.4. AES Encryption

### B.8.4.1. `_cpri__AESEncryptCBC()`

This function performs AES encryption in CBC chain mode. The input *dIn* buffer is encrypted into *dOut*.

The input iv buffer is required to have a size equal to the block size (16 bytes). The *dInSize* is required to be a multiple of the block size.

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	<i>dInSize</i> is not a multiple of the block size

```

39 CRYPT_RESULT
40 _cpri__AESEncryptCBC(
41     BYTE        *dOut,           // OUT:
42     UINT32      keySizeInBits,  // IN: key size in bits
43     BYTE        *key,           // IN: key buffer. The size of this buffer
44                                     // in bytes is (keySizeInBits + 7) / 8
45     BYTE        *iv,            // IN/OUT: IV for decryption.
46     UINT32      dInSize,        // IN: data size (is required to be a multiple
47                                     // of 16 bytes
48     BYTE        *dIn            // IN/OUT: data buffer
49 )
50 {
51     AES_KEY      AesKey;
52     BYTE        *pIv;
53     INT32       dSize;           // Need a signed version
54     int         i;
55
56     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
57
58     if(dInSize == 0)
59         return CRYPT_SUCCESS;
60
61     pAssert(dInSize <= INT32_MAX);
62     dSize = (INT32)dInSize;
63
64     // For CBC, the data size must be an even multiple of the
65     // cipher block size
66     if((dSize % 16) != 0)
67         return CRYPT_PARAMETER;
68
69     // Create AES encrypt key schedule
70     if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
71         FAIL(FATAL_ERROR_INTERNAL);
72
73     // XOR the data block into the IV, encrypt the IV into the IV
74     // and then copy the IV to the output
75     for(; dSize > 0; dSize -= 16)
76     {

```

```

77     pIv = iv;
78     for(i = 16; i > 0; i--)
79         *pIv++ ^= *dIn++;
80     AES_encrypt(iv, iv, &AesKey);
81     pIv = iv;
82     for(i = 16; i > 0; i--)
83         *dOut++ = *pIv++;
84 }
85 return CRYPT_SUCCESS;
86 }

```

#### B.8.4.2. `_cpri__AESDecryptCBC()`

This function performs AES decryption in CBC chain mode. The input *dIn* buffer is decrypted into *dOut*.

The input iv buffer is required to have a size equal to the block size (16 bytes). The *dInSize* is required to be a multiple of the block size.

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	<i>dInSize</i> is not a multiple of the block size

```

87  CRYPT_RESULT
88  _cpri__AESDecryptCBC(
89      BYTE        *dOut,           // OUT: the decrypted data
90      UINT32      keySizeInBits,  // IN: key size in bits
91      BYTE        *key,           // IN: key buffer. The size of this buffer
92                          // in bytes is (keySizeInBits + 7) / 8
93      BYTE        *iv,           // IN/OUT: IV for decryption. The size of
94                          // this buffer if 16 byte.
95      UINT32      dInSize,       // IN: data size
96      BYTE        *dIn           // IN: data buffer
97  )
98  {
99      AES_KEY      AesKey;
100     BYTE        *pIv;
101     int         i;
102     BYTE        tmp[16];
103     BYTE        *pT = NULL;
104     INT32       dSize;
105
106     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
107
108     if(dInSize == 0)
109         return CRYPT_SUCCESS;
110
111     pAssert(dInSize <= INT32_MAX);
112     dSize = (INT32)dInSize;
113
114     // For CBC, the data size must be an even multiple of the
115     // cipher block size
116     if((dSize % 16) != 0)
117         return CRYPT_PARAMETER;
118
119     // Create AES key schedule
120     if (AES_set_decrypt_key(key, keySizeInBits, &AesKey) != 0)
121         FAIL(FATAL_ERROR_INTERNAL);
122
123     // Copy the input data to a temp buffer, decrypt the buffer into the output;
124     // XOR in the IV, and copy the temp buffer to the IV and repeat.
125     for(; dSize > 0; dSize -= 16)
126     {

```



```

127     pT = tmp;
128     for(i = 16; i > 0; i--)
129         *pT++ = *dIn++;
130     AES_decrypt(tmp, dOut, &AesKey);
131     pIv = iv;
132     pT = tmp;
133     for(i = 16; i > 0; i--)
134     {
135         *dOut++ ^= *pIv;
136         *pIv++ = *pT++;
137     }
138 }
139 return CRYPT_SUCCESS;
140 }

```

#### B.8.4.3. `_cpri__AESEncryptCFB()`

This function performs AES encryption in CFB chain mode. The *dOut* buffer receives the values encrypted *dIn*. The input *iv* is assumed to be the size of an encryption block (16 bytes). The *iv* buffer will be modified to contain the last encrypted block.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

141 CRYPT_RESULT
142 _cpri__AESEncryptCFB(
143     BYTE        *dOut,           // OUT: the encrypted
144     UINT32      keySizeInBits,  // IN: key size in bit
145     BYTE        *key,           // IN: key buffer. The size of this buffer
146                                     // in bytes is (keySizeInBits + 7) / 8
147     BYTE        *iv,            // IN/OUT: IV for decryption.
148     UINT32      dInSize,        // IN: data size
149     BYTE        *dIn            // IN/OUT: data buffer
150 )
151 {
152     BYTE        *pIv;
153     AES_KEY     AesKey;
154     INT32       dSize;          // Need a signed version of dInSize
155     int         i;
156
157     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
158
159     if(dInSize == 0)
160         return CRYPT_SUCCESS;
161
162     pAssert(dInSize <= INT32_MAX);
163     dSize = (INT32)dInSize;
164
165     // Create AES encryption key schedule
166     if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
167         FAIL(FATAL_ERROR_INTERNAL);
168
169     // Encrypt the IV into the IV, XOR in the data, and copy to output
170     for(; dSize > 0; dSize -= 16)
171     {
172         // Encrypt the current value of the IV
173         AES_encrypt(iv, iv, &AesKey);
174         pIv = iv;
175         for(i = (int)(dSize < 16) ? dSize : 16; i > 0; i--)
176             // XOR the data into the IV to create the cipher text
177             // and put into the output
178             *dOut++ = *pIv++ ^= *dIn++;
179     }

```

```

180     return CRYPT_SUCCESS;
181 }

```

#### B.8.4.4. `_cpri__AESDecryptCFB()`

This function performs AES decrypt in CFB chain mode. The `dOut` buffer receives the values decrypted from `dIn`.

The input `iv` is assumed to be the size of an encryption block (16 bytes). The `iv` buffer will be modified to contain the last decoded block, padded with zeros

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

182  CRYPT_RESULT
183  _cpri__AESDecryptCFB(
184      BYTE          *dOut,          // OUT: the decrypted data
185      UINT32        keySizeInBits, // IN: key size in bit
186      BYTE          *key,          // IN: key buffer. The size of this buffer
187                          //      in bytes is (keySizeInBits + 7) / 8
188      BYTE          *iv,          // IN/OUT: IV for decryption.
189      UINT32        dInSize,      // IN: data size
190      BYTE          *dIn          // IN/OUT: data buffer
191  )
192  {
193      BYTE          *pIv;
194      BYTE          tmp[16];
195      int          i;
196      BYTE          *pT;
197      AES_KEY      AesKey;
198      INT32        dSize;
199
200      pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
201
202      if(dInSize == 0)
203          return CRYPT_SUCCESS;
204
205      pAssert(dInSize <= INT32_MAX);
206      dSize = (INT32)dInSize;
207
208      // Create AES encryption key schedule
209      if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
210          FAIL(FATAL_ERROR_INTERNAL);
211
212      for(; dSize > 0; dSize -= 16)
213      {
214          // Encrypt the IV into the temp buffer
215          AES_encrypt(iv, tmp, &AesKey);
216          pT = tmp;
217          pIv = iv;
218          for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
219              // Copy the current cipher text to IV, XOR
220              // with the temp buffer and put into the output
221              *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
222      }
223      // If the inner loop (i loop) was smaller than 16, then dSize
224      // would have been smaller than 16 and it is now negative
225      // If it is negative, then it indicates how many fill bytes
226      // are needed to pad out the IV for the next round.
227      for(; dSize < 0; dSize++)
228          *pIv++ = 0;
229
230      return CRYPT_SUCCESS;

```

231 }  
}**B.8.4.5. \_cpri\_\_AEESEncryptCTR()**

This function performs AES encryption/decryption in CTR chain mode. The *dIn* buffer is encrypted into *dOut*. The input iv buffer is assumed to have a size equal to the AES block size (16 bytes). The iv will be incremented by the number of blocks (full and partial) that were encrypted.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

232 CRYPT_RESULT
233 _cpri__AEESEncryptCTR(
234     BYTE      *dOut,           // OUT: the encrypted data
235     UINT32    keySizeInBits, // IN: key size in bits
236     BYTE      *key,           // IN: key buffer. The size of this buffer
237                                     // in bytes is (keySizeInBits + 7) / 8
238     BYTE      *iv,           // IN/OUT: IV for decryption.
239     UINT32    dInSize,       // IN: data size
240     BYTE      *dIn           // IN: data buffer
241 )
242 {
243     BYTE      tmp[16];
244     BYTE      *pT;
245     AES_KEY   AesKey;
246     int       i;
247     INT32     dSize;
248
249     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
250
251     if(dInSize == 0)
252         return CRYPT_SUCCESS;
253
254     pAssert(dInSize <= INT32_MAX);
255     dSize = (INT32)dInSize;
256
257     // Create AES encryption schedule
258     if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
259         FAIL(FATAL_ERROR_INTERNAL);
260
261     for(; dSize > 0; dSize -= 16)
262     {
263         // Encrypt the current value of the IV(counter)
264         AES_encrypt(iv, (BYTE *)tmp, &AesKey);
265
266         //increment the counter (counter is big-endian so start at end)
267         for(i = 15; i >= 0; i--)
268             if((iv[i] += 1) != 0)
269                 break;
270
271         // XOR the encrypted counter value with input and put into output
272         pT = tmp;
273         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
274             *dOut++ = *dIn++ ^ *pT++;
275     }
276     return CRYPT_SUCCESS;
277 }

```

**B.8.4.6. `_cpri__AESDecryptCTR()`**

Counter mode decryption uses the same algorithm as encryption. The `_cpri__AESDecryptCTR()` function is implemented as a macro call to `_cpri__AESEncryptCTR()`. (skip)

```

278  /// #define _cpri__AESDecryptCTR(dOut, keySize, key, iv, dInSize, dIn) \
279  ///     _cpri__AESEncryptCTR(
280  ///         ((BYTE *)dOut),
281  ///         ((UINT32)keySize),
282  ///         ((BYTE *)key),
283  ///         ((BYTE *)iv),
284  ///         ((UINT32)dInSize),
285  ///         ((BYTE *)dIn)
286  ///     )
287  /// 

```

The `///` is used by the prototype extraction program to cause it to include the line in the prototype file after removing the `///`. Need an extra line with nothing on it so that a blank line will separate this macro from the next definition.

**B.8.4.7. `_cpri__AESEncryptECB()`**

AES encryption in ECB mode. The `data` buffer is modified to contain the cipher text.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

288  CRYPT_RESULT
289  _cpri__AESEncryptECB(
290      BYTE      *dOut,          // OUT: encrypted data
291      UINT32    keySizeInBits, // IN: key size in bit
292      BYTE      *key,          // IN: key buffer. The size of this buffer
293                          // in bytes is (keySizeInBits + 7) / 8
294      UINT32    dInSize,       // IN: data size
295      BYTE      *dIn           // IN: clear text buffer
296  )
297  {
298      AES_KEY    AesKey;
299      INT32      dSize;
300
301      pAssert(dOut != NULL && key != NULL && dIn != NULL);
302
303      if(dInSize == 0)
304          return CRYPT_SUCCESS;
305
306      pAssert(dInSize <= INT32_MAX);
307      dSize = (INT32)dInSize;
308
309      // For ECB, the data size must be an even multiple of the
310      // cipher block size
311      if((dSize % 16) != 0)
312          return CRYPT_PARAMETER;
313      // Create AES encrypting key schedule
314      if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
315          FAIL(FATAL_ERROR_INTERNAL);
316
317      for(; dSize > 0; dSize -= 16)
318      {
319          AES_encrypt(dIn, dOut, &AesKey);
320          dIn = &dIn[16];
321          dOut = &dOut[16];
322      }

```

```

323     return CRYPT_SUCCESS;
324 }

```

#### B.8.4.8. `_cpri__AESDecryptECB()`

This function performs AES decryption using ECB (not recommended). The cipher text *dIn* is decrypted into *dOut*.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

325 CRYPT_RESULT
326 _cpri__AESDecryptECB(
327     BYTE      *dOut,           // OUT: the clear text data
328     UINT32    keySizeInBits,  // IN: key size in bit
329     BYTE      *key,           // IN: key buffer. The size of this buffer
330                                     // in bytes is (keySizeInBits + 7) / 8
331     UINT32    dInSize,        // IN: data size
332     BYTE      *dIn            // IN: cipher text buffer
333 )
334 {
335     AES_KEY    AesKey;
336     INT32      dSize;
337
338     pAssert(dOut != NULL && key != NULL && dIn != NULL);
339
340     if(dInSize == 0)
341         return CRYPT_SUCCESS;
342
343     pAssert(dInSize <= INT32_MAX);
344     dSize = (INT32)dInSize;
345
346     // For ECB, the data size must be an even multiple of the
347     // cipher block size
348     if((dSize % 16) != 0)
349         return CRYPT_PARAMETER;
350
351     // Create AES decryption key schedule
352     if (AES_set_decrypt_key(key, keySizeInBits, &AesKey) != 0)
353         FAIL(FATAL_ERROR_INTERNAL);
354
355     for(; dSize > 0; dSize -= 16)
356     {
357         AES_decrypt(dIn, dOut, &AesKey);
358         dIn = &dIn[16];
359         dOut = &dOut[16];
360     }
361     return CRYPT_SUCCESS;
362 }

```

#### B.8.4.9. `_cpri__AESEncryptOFB()`

This function performs AES encryption/decryption in OFB chain mode. The *dIn* buffer is modified to contain the encrypted/decrypted text.

The input *iv* buffer is assumed to have a size equal to the block size (16 bytes). The returned value of *iv* will be the *n*th encryption of the IV, where *n* is the number of blocks (full or partial) in the data stream.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

363 CRYPT_RESULT
364 _cpri__AESEncryptOFB(
365     BYTE      *dOut,           // OUT: the encrypted/decrypted data
366     UINT32    keySizeInBits,  // IN: key size in bit
367     BYTE      *key,           // IN: key buffer. The size of this buffer
368                                     // in bytes is (keySizeInBits + 7) / 8
369     BYTE      *iv,           // IN/OUT: IV for decryption. The size of
370                                     // this buffer is 16 byte.
371     UINT32    dInSize,       // IN: data size
372     BYTE      *dIn           // IN: data buffer
373 )
374 {
375     BYTE      *pIv;
376     AES_KEY   AesKey;
377     INT32     dSize;
378     int       i;
379
380     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
381
382     if(dInSize == 0)
383         return CRYPT_SUCCESS;
384
385     pAssert(dInSize <= INT32_MAX);
386     dSize = (INT32)dInSize;
387
388     // Create AES key schedule
389     if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
390         FAIL(FATAL_ERROR_INTERNAL);
391
392     // This is written so that dIn and dOut may be the same
393
394     for(; dSize > 0; dSize -= 16)
395     {
396         // Encrypt the current value of the "IV"
397         AES_encrypt(iv, iv, &AesKey);
398
399         // XOR the encrypted IV into dIn to create the cipher text (dOut)
400         pIv = iv;
401         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
402             *dOut++ = (*pIv++ ^ *dIn++);
403     }
404     return CRYPT_SUCCESS;
405 }

```

#### B.8.4.10. **\_cpri\_\_AESDecryptOFB()**

OFB encryption and decryption use the same algorithms for both. The `_cpri__AESDecryptOFB()` function is implemented as a macro call to `_cpri__AESEncryptOFB()`. (skip)

```

406 //#define _cpri__AESDecryptOFB(dOut,keySizeInBits, key, iv, dInSize, dIn) \
407 /** _cpri__AESEncryptOFB ( \
408 /**     ((BYTE *)dOut), \
409 /**     ((UINT32)keySizeInBits), \
410 /**     ((BYTE *)key), \
411 /**     ((BYTE *)iv), \
412 /**     ((UINT32)dInSize), \
413 /**     ((BYTE *)dIn) \
414 /** )
415 /**

```

```
416 #ifndef TPM_ALG_SM4    //%
```

## B.8.5. SM4 Encryption

### B.8.5.1. `_cpri__SM4EncryptCBC()`

This function performs SM4 encryption in CBC chain mode. The input *dIn* buffer is encrypted into *dOut*.

The input iv buffer is required to have a size equal to the block size (16 bytes). The *dInSize* is required to be a multiple of the block size.

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	<i>dInSize</i> is not a multiple of the block size

```
417 CRYPT_RESULT
418 _cpri__SM4EncryptCBC(
419     BYTE        *dOut,           // OUT:
420     UINT32      keySizeInBits,  // IN: key size in bits
421     BYTE        *key,           // IN: key buffer. The size of this buffer
422                                     // in bytes is (keySizeInBits + 7) / 8
423     BYTE        *iv,            // IN/OUT: IV for decryption.
424     UINT32      dInSize,        // IN: data size (is required to be a multiple
425                                     // of 16 bytes
426     BYTE        *dIn            // IN/OUT: data buffer
427 )
428 {
429     SM4_KEY      Sm4Key;
430     BYTE        *pIv;
431     INT32      dSize;           // Need a signed version
432     int        i;
433
434     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
435
436     if(dInSize == 0)
437         return CRYPT_SUCCESS;
438
439     pAssert(dInSize <= INT32_MAX);
440     dSize = (INT32)dInSize;
441
442     // For CBC, the data size must be an even multiple of the
443     // cipher block size
444     if((dSize % 16) != 0)
445         return CRYPT_PARAMETER;
446
447     // Create SM4 encrypt key schedule
448     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
449         FAIL(FATAL_ERROR_INTERNAL);
450
451     // XOR the data block into the IV, encrypt the IV into the IV
452     // and then copy the IV to the output
453     for(; dSize > 0; dSize -= 16)
454     {
455         pIv = iv;
456         for(i = 16; i > 0; i--)
457             *pIv++ ^= *dIn++;
458         SM4_encrypt(iv, iv, &Sm4Key);
459         pIv = iv;
460         for(i = 16; i > 0; i--)
461             *dOut++ = *pIv++;
462     }
463     return CRYPT_SUCCESS;

```

464 }  
}**B.8.5.2. \_cpri\_\_SM4DecryptCBC()**

This function performs SM4 decryption in CBC chain mode. The input *dIn* buffer is decrypted into *dOut*.

The input *iv* buffer is required to have a size equal to the block size (16 bytes). The *dInSize* is required to be a multiple of the block size.

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	<i>dInSize</i> is not a multiple of the block size

```

465  CRYPT_RESULT
466  _cpri__SM4DecryptCBC(
467      BYTE      *dOut,          // OUT: the decrypted data
468      UINT32     keySizeInBits, // IN: key size in bits
469      BYTE      *key,          // IN: key buffer. The size of this buffer
470                          // in bytes is (keySizeInBits + 7) / 8
471      BYTE      *iv,          // IN/OUT: IV for decryption. The size of
472                          // this buffer if 16 byte.
473      UINT32     dInSize,      // IN: data size
474      BYTE      *dIn           // IN: data buffer
475  )
476  {
477      SM4_KEY     Sm4Key;
478      BYTE      *pIv;
479      int         i;
480      BYTE      tmp[16];
481      BYTE      *pT = NULL;
482      INT32      dSize;
483
484      pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
485
486      if(dInSize == 0)
487          return CRYPT_SUCCESS;
488
489      pAssert(dInSize <= INT32_MAX);
490      dSize = (INT32)dInSize;
491
492      // For CBC, the data size must be an even multiple of the
493      // cipher block size
494      if((dSize % 16) != 0)
495          return CRYPT_PARAMETER;
496
497      // Create SM4 key schedule
498      if (SM4_set_decrypt_key(key, keySizeInBits, &Sm4Key) != 0)
499          FAIL(FATAL_ERROR_INTERNAL);
500
501      // Copy the input data to a temp buffer, decrypt the buffer into the output;
502      // XOR in the IV, and copy the temp buffer to the IV and repeat.
503      for(; dSize > 0; dSize -= 16)
504      {
505          pT = tmp;
506          for(i = 16; i > 0; i--)
507              *pT++ = *dIn++;
508          SM4_decrypt(tmp, dOut, &Sm4Key);
509          pIv = iv;
510          pT = tmp;
511          for(i = 16; i > 0; i--)
512          {
513              *dOut++ ^= *pIv;

```



```

514         *pIv++ = *pT++;
515     }
516 }
517 return CRYPT_SUCCESS;
518 }

```

### B.8.5.3. `_cpri__SM4EncryptCFB()`

This function performs SM4 encryption in CFB chain mode. The *dOut* buffer receives the values encrypted *dIn*. The input *iv* is assumed to be the size of an encryption block (16 bytes). The *iv* buffer will be modified to contain the last encrypted block.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

519 CRYPT_RESULT
520 _cpri__SM4EncryptCFB(
521     BYTE        *dOut,           // OUT: the encrypted
522     UINT32      keySizeInBits,  // IN: key size in bit
523     BYTE        *key,           // IN: key buffer. The size of this buffer
524                                     // in bytes is (keySizeInBits + 7) / 8
525     BYTE        *iv,           // IN/OUT: IV for decryption.
526     UINT32      dInSize,       // IN: data size
527     BYTE        *dIn           // IN/OUT: data buffer
528 )
529 {
530     BYTE        *pIv;
531     SM4_KEY     Sm4Key;
532     INT32       dSize;         // Need a signed version of dInSize
533     int         i;
534
535     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
536
537     if(dInSize == 0)
538         return CRYPT_SUCCESS;
539
540     pAssert(dInSize <= INT32_MAX);
541     dSize = (INT32)dInSize;
542
543     // Create SM4 encryption key schedule
544     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
545         FAIL(FATAL_ERROR_INTERNAL);
546
547     // Encrypt the IV into the IV, XOR in the data, and copy to output
548     for(; dSize > 0; dSize -= 16)
549     {
550         // Encrypt the current value of the IV
551         SM4_encrypt(iv, iv, &Sm4Key);
552         pIv = iv;
553         for(i = (int)(dSize < 16) ? dSize : 16; i > 0; i--)
554             // XOR the data into the IV to create the cipher text
555             // and put into the output
556             *dOut++ = *pIv++ ^= *dIn++;
557     }
558     return CRYPT_SUCCESS;
559 }

```

### B.8.5.4. `_cpri__SM4DecryptCFB()`

This function performs SM4 decrypt in CFB chain mode. The *dOut* buffer receives the values decrypted from *dIn*.

The input *iv* is assumed to be the size of an encryption block (16 bytes). The *iv* buffer will be modified to contain the last decoded block, padded with zeros

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

560  CRYPT_RESULT
561  __cpri__SM4DecryptCFB(
562      BYTE      *dOut,          // OUT: the decrypted data
563      UINT32     keySizeInBits, // IN: key size in bit
564      BYTE      *key,          // IN: key buffer. The size of this buffer
565                          // in bytes is (keySizeInBits + 7) / 8
566      BYTE      *iv,           // IN/OUT: IV for decryption.
567      UINT32     dInSize,      // IN: data size
568      BYTE      *dIn           // IN/OUT: data buffer
569  )
570  {
571      BYTE      *pIv;
572      BYTE      tmp[16];
573      int       i;
574      BYTE      *pT;
575      SM4_KEY   Sm4Key;
576      INT32     dSize;
577
578      pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
579
580      if(dInSize == 0)
581          return CRYPT_SUCCESS;
582
583      pAssert(dInSize <= INT32_MAX);
584      dSize = (INT32)dInSize;
585
586      // Create SM4 encryption key schedule
587      if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
588          FAIL(FATAL_ERROR_INTERNAL);
589
590      for(; dSize > 0; dSize -= 16)
591      {
592          // Encrypt the IV into the temp buffer
593          SM4_encrypt(iv, tmp, &Sm4Key);
594          pT = tmp;
595          pIv = iv;
596          for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
597              // Copy the current cipher text to IV, XOR
598              // with the temp buffer and put into the output
599              *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
600      }
601      // If the inner loop (i loop) was smaller than 16, then dSize
602      // would have been smaller than 16 and it is now negative
603      // If it is negative, then it indicates how many fill bytes
604      // are needed to pad out the IV for the next round.
605      for(; dSize < 0; dSize++)
606          *iv++ = 0;
607
608      return CRYPT_SUCCESS;
609  }

```

#### B.8.5.5. \_\_cpri\_\_SM4EncryptCTR()

This function performs SM4 encryption/decryption in CTR chain mode. The *dIn* buffer is encrypted into *dOut*. The input *iv* buffer is assumed to have a size equal to the SM4 block size (16 bytes). The *iv* will be incremented by the number of blocks (full and partial) that were encrypted.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

610 CRYPT_RESULT
611 _cpri__SM4EncryptCTR(
612     BYTE      *dOut,          // OUT: the encrypted data
613     UINT32     keySizeInBits, // IN: key size in bits
614     BYTE      *key,          // IN: key buffer. The size of this buffer
615                                     // in bytes is (keySizeInBits + 7) / 8
616     BYTE      *iv,           // IN/OUT: IV for decryption.
617     UINT32     dInSize,      // IN: data size
618     BYTE      *dIn           // IN: data buffer
619 )
620 {
621     BYTE      tmp[16];
622     BYTE      *pT;
623     SM4_KEY   Sm4Key;
624     int       i;
625     INT32     dSize;
626
627     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
628
629     if(dInSize == 0)
630         return CRYPT_SUCCESS;
631
632     pAssert(dInSize <= INT32_MAX);
633     dSize = (INT32)dInSize;
634
635     // Create SM4 encryption schedule
636     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
637         FAIL(FATAL_ERROR_INTERNAL);
638
639     for(; dSize > 0; dSize--)
640     {
641         // Encrypt the current value of the IV(counter)
642         SM4_encrypt(iv, (BYTE *)tmp, &Sm4Key);
643
644         //increment the counter
645         for(i = 0; i < 16; i++)
646             if((iv[i] += 1) != 0)
647                 break;
648
649         // XOR the encrypted counter value with input and put into output
650         pT = tmp;
651         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
652             *dOut++ = *dIn++ ^ *pT++;
653     }
654     return CRYPT_SUCCESS;
655 }

```

#### B.8.5.6. **\_cpri\_\_SM4DecryptCTR()**

Counter mode decryption uses the same algorithm as encryption. The `_cpri__SM4DecryptCTR()` function is implemented as a macro call to `_cpri__SM4EncryptCTR()`. (skip)

```

656 ///define _cpri__SM4DecryptCTR(dOut, keySize, key, iv, dInSize, dIn) \
657 ///    _cpri__SM4EncryptCTR(                                \
658 ///        ((BYTE *)dOut),                                \
659 ///        ((UINT32)keySize),                               \
660 ///        ((BYTE *)key),                                  \
661 ///        ((BYTE *)iv),                                   \
662 ///        ((UINT32)dInSize),                               \

```

```

663 //%                ((BYTE *)dIn)                \
664 //%                )
665 //%

```

The `//%` is used by the prototype extraction program to cause it to include the line in the prototype file after removing the `//%`. Need an extra line with nothing on it so that a blank line will separate this macro from the next definition.

### B.8.5.7. `_cpri__SM4EncryptECB()`

SM4 encryption in ECB mode. The *data* buffer is modified to contain the cipher text.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

666 CRYPT_RESULT
667 _cpri__SM4EncryptECB(
668     BYTE        *dOut,           // OUT: encrypted data
669     UINT32      keySizeInBits,  // IN: key size in bit
670     BYTE        *key,           // IN: key buffer. The size of this buffer
671                               // in bytes is (keySizeInBits + 7) / 8
672     UINT32      dInSize,        // IN: data size
673     BYTE        *dIn            // IN: clear text buffer
674 )
675 {
676     SM4_KEY      Sm4Key;
677     INT32        dSize;
678
679     pAssert(dOut != NULL && key != NULL && dIn != NULL);
680
681     if(dInSize == 0)
682         return CRYPT_SUCCESS;
683
684     pAssert(dInSize <= INT32_MAX);
685     dSize = (INT32)dInSize;
686
687     // For ECB, the data size must be an even multiple of the
688     // cipher block size
689     if((dSize % 16) != 0)
690         return CRYPT_PARAMETER;
691     // Create SM4 encrypting key schedule
692     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
693         FAIL(FATAL_ERROR_INTERNAL);
694
695     for(; dSize > 0; dSize -= 16)
696     {
697         SM4_encrypt(dIn, dOut, &Sm4Key);
698         dIn = &dIn[16];
699         dOut = &dOut[16];
700     }
701     return CRYPT_SUCCESS;
702 }

```

### B.8.5.8. `_cpri__SM4DecryptECB()`

This function performs SM4 decryption using ECB (not recommended). The cipher text *dIn* is decrypted into *dOut*.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

703 CRYPT_RESULT
704 _cpri__SM4DecryptECB(
705     BYTE      *dOut,           // OUT: the clear text data
706     UINT32    keySizeInBits,  // IN: key size in bit
707     BYTE      *key,           // IN: key buffer. The size of this buffer
708                                     // in bytes is (keySizeInBits + 7) / 8
709     UINT32    dInSize,        // IN: data size
710     BYTE      *dIn            // IN: cipher text buffer
711 )
712 {
713     SM4_KEY    Sm4Key;
714     INT32      dSize;
715
716     pAssert(dOut != NULL && key != NULL && dIn != NULL);
717
718     if(dInSize == 0)
719         return CRYPT_SUCCESS;
720
721     pAssert(dInSize <= INT32_MAX);
722     dSize = (INT32)dInSize;
723
724     // For ECB, the data size must be an even multiple of the
725     // cipher block size
726     if((dSize % 16) != 0)
727         return CRYPT_PARAMETER;
728
729     // Create SM4 decryption key schedule
730     if (SM4_set_decrypt_key(key, keySizeInBits, &Sm4Key) != 0)
731         FAIL(FATAL_ERROR_INTERNAL);
732
733     for(; dSize > 0; dSize -= 16)
734     {
735         SM4_decrypt(dIn, dOut, &Sm4Key);
736         dIn = &dIn[16];
737         dOut = &dOut[16];
738     }
739     return CRYPT_SUCCESS;
740 }

```

#### B.8.5.9. **\_cpri\_\_SM4EncryptOFB()**

This function performs SM4 encryption/decryption in OFB chain mode. The *dIn* buffer is modified to contain the encrypted/decrypted text.

The input *iv* buffer is assumed to have a size equal to the block size (16 bytes). The returned value of *iv* will be the *n*th encryption of the IV, where *n* is the number of blocks (full or partial) in the data stream.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```

741 CRYPT_RESULT
742 _cpri__SM4EncryptOFB(
743     BYTE      *dOut,           // OUT: the encrypted/decrypted data
744     UINT32    keySizeInBits,  // IN: key size in bit
745     BYTE      *key,           // IN: key buffer. The size of this buffer
746                                     // in bytes is (keySizeInBits + 7) / 8
747     BYTE      *iv,            // IN/OUT: IV for decryption. The size of
748                                     // this buffer is 16 byte.

```

```

749     UINT32     dInSize,      // IN: data size
750     BYTE      *dIn         // IN: data buffer
751 )
752 {
753     BYTE      *pIv;
754     SM4_KEY   Sm4Key;
755     INT32     dSize;
756     int       i;
757
758     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
759
760     if(dInSize == 0)
761         return CRYPT_SUCCESS;
762
763     pAssert(dInSize <= INT32_MAX);
764     dSize = (INT32)dInSize;
765
766     // Create SM4 key schedule
767     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
768         FAIL(FATAL_ERROR_INTERNAL);
769
770     // This is written so that dIn and dOut may be the same
771
772     for(; dSize > 0; dSize -= 16)
773     {
774         // Encrypt the current value of the "IV"
775         SM4_encrypt(iv, iv, &Sm4Key);
776
777         // XOR the encrypted IV into dIn to create the cipher text (dOut)
778         pIv = iv;
779         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
780             *dOut++ = (*pIv++ ^ *dIn++);
781     }
782     return CRYPT_SUCCESS;
783 }

```

#### B.8.5.10. `_cpri__SM4DecryptOFB()`

OFB encryption and decryption use the same algorithms for both. The `_cpri__SM4DecryptOFB()` function is implemented as a macro call to `_cpri__SM4EncryptOFB()`. (skip)

```

784 // #define _cpri__SM4DecryptOFB(dOut, keySizeInBits, key, iv, dInSize, dIn) \
785 //     _cpri__SM4EncryptOFB ( \
786 //         ((BYTE *)dOut), \
787 //         ((UINT32)keySizeInBits), \
788 //         ((BYTE *)key), \
789 //         ((BYTE *)iv), \
790 //         ((UINT32)dInSize), \
791 //         ((BYTE *)dIn) \
792 //     )
793 //
794 #endif // % TPM_ALG_SM4

```

## B.9 RSA Files

### B.9.1. CpriRSA.c

#### B.9.1.1. 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 TPM 2.0 specification: Integer values are considered to be an array of one or more bytes. The byte at offset zero within the array is the most significant byte of the integer. The interface uses TPM2B as a big number format for numeric values passed to/from *CryptUtil()*.

#### B.9.1.2. Includes

```
1 #include "CryptoEngine.h"
```

#### B.9.1.3. Local Functions

##### B.9.1.3.1. RsaPrivateExponent()

This function computes the private exponent  $de = 1 \text{ mod } (p-1)(q-1)$ . 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 denormalized.

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 exponent
5 )
6 {
7     BN_CTX      *context;
8     BIGNUM      *bnD;
9     BIGNUM      *bnN;
10    BIGNUM      *bnP;
11    BIGNUM      *bnE;
12    BIGNUM      *bnPhi;
13    BIGNUM      *bnQ;
14    BIGNUM      *bnQr;
15    UINT32      fill;
16
17    CRYPT_RESULT    retVal = CRYPT_SUCCESS;    // Assume success
18
19    pAssert(key != NULL && key->privateKey != NULL && key->publicKey != NULL);
20
21    context = BN_CTX_new();
22    if(context == NULL)
23        FAIL(FATAL_ERROR_ALLOCATION);
```

```

24     BN_CTX_start(context);
25     bnE = BN_CTX_get(context);
26     bnD = BN_CTX_get(context);
27     bnN = BN_CTX_get(context);
28     bnP = BN_CTX_get(context);
29     bnPhi = BN_CTX_get(context);
30     bnQ = BN_CTX_get(context);
31     bnQr = BN_CTX_get(context);
32
33     if(bnQr == NULL)
34         FAIL(FATAL_ERROR_ALLOCATION);
35
36     // Assume the size of the public key value is within range
37     pAssert(key->publicKey->size <= MAX_RSA_KEY_BYTES);
38
39     if( BN_bin2bn(key->publicKey->buffer, key->publicKey->size, bnN) == NULL
40         || BN_bin2bn(key->privateKey->buffer, key->privateKey->size, bnP) == NULL)
41
42         FAIL(FATAL_ERROR_INTERNAL);
43
44     // If P size is not 1/2 of n size, then this is not a valid value for this
45     // implementation. This will also catch the case were P is input as zero.
46     // This generates a return rather than an assert because the key being loaded
47     // might be SW generated and wrong.
48     if(BN_num_bits(bnP) < BN_num_bits(bnN)/2)
49     {
50         retVal = CRYPT_PARAMETER;
51         goto Cleanup;
52     }
53     // Get q = n/p;
54     if (BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
55         FAIL(FATAL_ERROR_INTERNAL);
56
57     // If there is a remainder, then this is not a valid n
58     if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
59     {
60         retVal = CRYPT_PARAMETER;        // problem may be recoverable
61         goto Cleanup;
62     }
63     // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
64     if( BN_copy(bnPhi, bnN) == NULL
65         || !BN_sub(bnPhi, bnPhi, bnP)
66         || !BN_sub(bnPhi, bnPhi, bnQ)
67         || !BN_add_word(bnPhi, 1))
68         FAIL(FATAL_ERROR_INTERNAL);
69
70     // Compute the multiplicative inverse
71     BN_set_word(bnE, key->exponent);
72     if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
73     {
74         // Going to assume that the error is caused by a bad
75         // set of parameters. Specifically, an exponent that is
76         // not compatible with the primes. In an implementation that
77         // has better visibility to the error codes, this might be
78         // refined so that failures in the library would return
79         // a more informative value. Should not assume here that
80         // the error codes will remain unchanged.
81
82         retVal = CRYPT_PARAMETER;
83         goto Cleanup;
84     }
85
86     fill = key->publicKey->size - BN_num_bytes(bnD);
87     BN_bn2bin(bnD, &key->privateKey->buffer[fill]);
88     memset(key->privateKey->buffer, 0, fill);
89

```



```

90     // Change the size of the private key so that it is known to contain
91     // a private exponent rather than a prime.
92     key->privateKey->size = key->publicKey->size;
93
94 Cleanup:
95     BN_CTX_end(context);
96     BN_CTX_free(context);
97     return retVal;
98 }

```

### B.9.1.3.2. `_cpri__TestKeyRSA()`

This function computes the private exponent  $de = 1 \bmod (p-1)(q-1)$ . 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.

Return Value	Meaning
<code>CRYPT_SUCCESS</code>	private exponent ( $d$ ) was generated
<code>CRYPT_PARAMETER</code>	one or more parameters are invalid

```

99 CRYPT_RESULT
100 _cpri__TestKeyRSA(
101     TPM2B      *d,           // OUT: the address to receive the private exponent
102     UINT32     exponent,    // IN: the public modulus
103     TPM2B      *publicKey,  // IN/OUT: an input if only one prime is provided.
104                     //          an output if both primes are provided
105     TPM2B      *prime1,     // IN: a first prime
106     TPM2B      *prime2     // IN: an optional second prime
107 )
108 {
109     BN_CTX      *context;
110     BIGNUM      *bnD;
111     BIGNUM      *bnN;
112     BIGNUM      *bnP;
113     BIGNUM      *bnE;
114     BIGNUM      *bnPhi;
115     BIGNUM      *bnQ;
116     BIGNUM      *bnQr;
117     UINT32      fill;
118
119     CRYPT_RESULT  retVal = CRYPT_SUCCESS;    // Assume success
120
121     pAssert(publicKey != NULL && prime1 != NULL);
122     // Make sure that the sizes are within range
123     pAssert( prime1->size <= MAX_RSA_KEY_BYTES/2
124             && publicKey->size <= MAX_RSA_KEY_BYTES);
125     pAssert( prime2 == NULL || prime2->size < MAX_RSA_KEY_BYTES/2);
126
127     if(publicKey->size/2 != prime1->size)
128         return CRYPT_PARAMETER;
129
130     context = BN_CTX_new();
131     if(context == NULL)
132         FAIL(FATAL_ERROR_ALLOCATION);
133     BN_CTX_start(context);
134     bnE = BN_CTX_get(context);    // public exponent (e)
135     bnD = BN_CTX_get(context);    // private exponent (d)
136     bnN = BN_CTX_get(context);    // public modulus (n)
137     bnP = BN_CTX_get(context);    // prime1 (p)

```

```

138     bnPhi = BN_CTX_get(context);    // (p-1)(q-1)
139     bnQ = BN_CTX_get(context);     // prime2 (q)
140     bnQr = BN_CTX_get(context);    // n mod p
141
142     if(bnQr == NULL)
143         FAIL(FATAL_ERROR_ALLOCATION);
144
145     if(BN_bin2bn(prime1->buffer, prime1->size, bnP) == NULL)
146         FAIL(FATAL_ERROR_INTERNAL);
147
148     // If prime2 is provided, then compute n
149     if(prime2 != NULL)
150     {
151         // Two primes provided so use them to compute n
152         if(BN_bin2bn(prime2->buffer, prime2->size, bnQ) == NULL)
153             FAIL(FATAL_ERROR_INTERNAL);
154
155         // Make sure that the sizes of the primes are compatible
156         if(BN_num_bits(bnQ) != BN_num_bits(bnP))
157         {
158             retVal = CRYPT_PARAMETER;
159             goto Cleanup;
160         }
161         // Multiply the primes to get the public modulus
162
163         if(BN_mul(bnN, bnP, bnQ, context) != 1)
164             FAIL(FATAL_ERROR_INTERNAL);
165
166         // if the space provided for the public modulus is large enough,
167         // save the created value
168         if(BN_num_bits(bnN) != (publicKey->size * 8))
169         {
170             retVal = CRYPT_PARAMETER;
171             goto Cleanup;
172         }
173         BN_bn2bin(bnN, publicKey->buffer);
174     }
175     else
176     {
177         // One prime provided so find the second prime by division
178         BN_bin2bn(publicKey->buffer, publicKey->size, bnN);
179
180         // Get q = n/p;
181         if(BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
182             FAIL(FATAL_ERROR_INTERNAL);
183
184         // If there is a remainder, then this is not a valid n
185         if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
186         {
187             retVal = CRYPT_PARAMETER;    // problem may be recoverable
188             goto Cleanup;
189         }
190     }
191     // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
192     BN_copy(bnPhi, bnN);
193     BN_sub(bnPhi, bnPhi, bnP);
194     BN_sub(bnPhi, bnPhi, bnQ);
195     BN_add_word(bnPhi, 1);
196     // Compute the multiplicative inverse
197     BN_set_word(bnE, exponent);
198     if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
199     {
200         // Going to assume that the error is caused by a bad set of parameters.
201         // Specifically, an exponent that is not compatible with the primes.
202         // In an implementation that has better visibility to the error codes,
203         // this might be refined so that failures in the library would return

```

```

204     // a more informative value.
205     // Do not assume that the error codes will remain unchanged.
206     retVal = CRYPT_PARAMETER;
207     goto Cleanup;
208 }
209 // Return the private exponent.
210 // Make sure it is normalized to have the correct size.
211 d->size = publicKey->size;
212 fill = d->size - BN_num_bytes(bnD);
213 BN_bn2bin(bnD, &d->buffer[fill]);
214 memset(d->buffer, 0, fill);
215 Cleanup:
216     BN_CTX_end(context);
217     BN_CTX_free(context);
218     return retVal;
219 }

```

### B.9.1.3.3. 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 ( $n$ ).

Return Value	Meaning
CRYPT_SUCCESS	encryption complete
CRYPT_PARAMETER	number to exponentiate is larger than the modulus

```

220 static CRYPT_RESULT
221 RSAEP (
222     UINT32     dInOutSize, // OUT size of the encrypted block
223     BYTE      *dInOut,    // OUT: the encrypted data
224     RSA_KEY   *key        // IN: the key to use
225 )
226 {
227     UINT32     e;
228     BYTE      exponent[4];
229     CRYPT_RESULT retVal;
230
231     e = key->exponent;
232     if(e == 0)
233         e = RSA_DEFAULT_PUBLIC_EXPONENT;
234     UINT32_TO_BYTE_ARRAY(e, exponent);
235
236     //!!! Can put check for test of RSA here
237
238     retVal = _math_ModExp(dInOutSize, dInOut, dInOutSize, dInOut, 4, exponent,
239                         key->publicKey->size, key->publicKey->buffer);
240
241     // Exponentiation result is stored in-place, thus no space shortage is possible.
242     pAssert(retVal != CRYPT_UNDERFLOW);
243
244     return retVal;
245 }

```

### B.9.1.3.4. 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.

Return Value	Meaning
CRYPT_SUCCESS	decryption succeeded
CRYPT_PARAMETER	the value to decrypt is larger than the modulus

```

246 static CRYPT_RESULT
247 RSADP (
248     UINT32      dInOutSize,    // IN/OUT: size of decrypted data
249     BYTE        *dInOut,      // IN/OUT: the decrypted data
250     RSA_KEY     *key          // IN: the key
251 )
252 {
253     CRYPT_RESULT retVal;
254
255     //!!! Can put check for RSA tested here
256
257     // Make sure that the pointers are provided and that the private key is present
258     // If the private key is present it is assumed to have been created by
259     // so is presumed good_cpri_PrivateExponent
260     pAssert(key != NULL && dInOut != NULL &&
261             key->publicKey->size == key->publicKey->size);
262
263     // make sure that the value to be decrypted is smaller than the modulus
264     // note: this check is redundant as is also performed by _math_ModExp()
265     // which is optimized for use in RSA operations
266     if(_math_uComp(key->publicKey->size, key->publicKey->buffer,
267                  dInOutSize, dInOut) <= 0)
268         return CRYPT_PARAMETER;
269
270     // _math_ModExp can return CRYPT_PARAMETER or CRYPT_UNDERFLOW but actual
271     // underflow is not possible because everything is in the same buffer.
272     retVal = _math_ModExp(dInOutSize, dInOut, dInOutSize, dInOut,
273                          key->privateKey->size, key->privateKey->buffer,
274                          key->publicKey->size, key->publicKey->buffer);
275
276     // Exponentiation result is stored in-place, thus no space shortage is possible.
277     pAssert(retVal != CRYPT_UNDERFLOW);
278
279     return retVal;
280 }

```

#### B.9.1.3.5. OaepEncode()

This function performs OAEP padding. The size of the buffer to receive the OAEP padded data must equal the size of the modulus

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	<i>hashAlg</i> is not valid
CRYPT_FAIL	message size is too large

```

281 static CRYPT_RESULT
282 OaepEncode (
283     UINT32      paddedSize,    // IN: pad value size
284     BYTE        *padded,      // OUT: the pad data
285     TPM_ALG_ID hashAlg,      // IN: algorithm to use for padding
286     const char *label,       // IN: null-terminated string (may be NULL)
287     UINT32      messageSize,  // IN: the message size
288     BYTE        *message     // IN: the message being padded
289 #ifdef TEST_RSA
290     //

```

```

290     , BYTE      *testSeed    // IN: optional seed used for testing.
291 #endif // TEST_RSA          //
292 )
293 {
294     UINT32      padLen;
295     UINT32      dbSize;
296     UINT32      i;
297     BYTE        mySeed[MAX_DIGEST_SIZE];
298     BYTE        *seed = mySeed;
299     INT32       hLen = _cpri_GetDigestSize(hashAlg);
300     BYTE        mask[MAX_RSA_KEY_BYTES];
301     BYTE        *pp;
302     BYTE        *pm;
303     UINT32      lSize = 0;
304     CRYPT_RESULT retVal = CRYPT_SUCCESS;
305
306
307     pAssert(padded != NULL && message != NULL);
308
309     // A value of zero is not allowed because the KDF can't produce a result
310     // if the digest size is zero.
311     if(hLen <= 0)
312         return CRYPT_PARAMETER;
313
314     // If a label is provided, get the length of the string, including the
315     // terminator
316     if(label != NULL)
317         lSize = (UINT32)strlen(label) + 1;
318
319     // Basic size check
320     // messageSize <= k * 2hLen * 2
321     if(messageSize > paddedSize - 2 * hLen - 2)
322         return CRYPT_FAIL;
323
324     // Hash L even if it is null
325     // Offset into padded leaving room for masked seed and byte of zero
326     pp = &padded[hLen + 1];
327     retVal = _cpri_HashBlock(hashAlg, lSize, (BYTE *)label, hLen, pp);
328
329     // concatenate PS of k * mLLen * 2hLen * 2
330     padLen = paddedSize - messageSize - (2 * hLen) - 2;
331     memset(&pp[hLen], 0, padLen);
332     pp[hLen+padLen] = 0x01;
333     padLen += 1;
334     memcpy(&pp[hLen+padLen], message, messageSize);
335
336     // The total size of db = hLen + pad + mSize;
337     dbSize = hLen+padLen+messageSize;
338
339     // If testing, then use the provided seed. Otherwise, use values
340     // from the RNG
341 #ifdef TEST_RSA
342     if(testSeed != NULL)
343         seed = testSeed;
344     else
345 #endif // TEST_RSA
346         _cpri_GenerateRandom(hLen, mySeed);
347
348     // mask = MGF1 (seed, nSize * hLen * 1)
349     if((retVal = _cpri_MGF1(dbSize, mask, hashAlg, hLen, seed)) < 0)
350         return retVal; // Don't expect an error because hash size is not zero
351                       // was detected in the call to _cpri_HashBlock() above.
352
353     // Create the masked db
354     pm = mask;

```

```

355     for(i = dbSize; i > 0; i--)
356         *pp++ ^= *pm++;
357     pp = &padded[hLen + 1];
358
359     // Run the masked data through MGF1
360     if((retVal = _cpri_MGF1(hLen, &padded[1], hashAlg, dbSize, pp)) < 0)
361         return retVal; // Don't expect zero here as the only case for zero
362                        // was detected in the call to _cpri_HashBlock() above.
363
364     // Now XOR the seed to create masked seed
365     pp = &padded[1];
366     pm = seed;
367     for(i = hLen; i > 0; i--)
368         *pp++ ^= *pm++;
369
370     // Set the first byte to zero
371     *padded = 0x00;
372     return CRYPT_SUCCESS;
373 }

```

#### B.9.1.3.6. 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.

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

```

374 static CRYPT_RESULT
375 OaepDecode (
376     UINT32     *dataOutSize, // IN/OUT: the recovered data size
377     BYTE       *dataOut,    // OUT: the recovered data
378     TPM_ALG_ID hashAlg,    // IN: algorithm to use for padding
379     const char *label,     // IN: null-terminated string (may be NULL)
380     UINT32     paddedSize, // IN: the size of the padded data
381     BYTE       *padded     // IN: the padded data
382 )
383 {
384     UINT32     dSizeSave;
385     UINT32     i;
386     BYTE       seedMask[MAX_DIGEST_SIZE];
387     INT32     hLen = _cpri_GetDigestSize(hashAlg);
388
389     BYTE       mask[MAX_RSA_KEY_BYTES];
390     BYTE       *pp;
391     BYTE       *pm;
392     UINT32     lSize = 0;
393     CRYPT_RESULT retVal = CRYPT_SUCCESS;
394
395     // Unknown hash
396     pAssert(hLen > 0 && dataOutSize != NULL && dataOut != NULL && padded != NULL);
397
398     // If there is a label, get its size including the terminating 0x00
399     if(label != NULL)
400         lSize = (UINT32)strlen(label) + 1;
401
402     // Set the return size to zero so that it doesn't have to be done on each

```

```

403     // failure
404     dSizeSave = *dataOutSize;
405     *dataOutSize = 0;
406
407     // Strange size (anything smaller can't be an OAEP padded block)
408     // Also check for no leading 0
409     if(paddedSize < (<K>unsigned)((2 * hLen) + 2) || *padded != 0)
410         return CRYPT_FAIL;
411
412     // Use the hash size to determine what to put through MGF1 in order
413     // to recover the seedMask
414     if((retVal = _cpri__MGF1(hLen, seedMask, hashAlg,
415                             paddedSize-hLen-1, &padded[hLen+1])) < 0)
416         return retVal;
417
418     // Recover the seed into seedMask
419     pp = &padded[1];
420     pm = seedMask;
421     for(i = hLen; i > 0; i--)
422         *pm++ ^= *pp++;
423
424     // Use the seed to generate the data mask
425     if((retVal = _cpri__MGF1(paddedSize-hLen-1, mask, hashAlg,
426                             hLen, seedMask)) < 0)
427         return retVal;
428
429     // Use the mask generated from seed to recover the padded data
430     pp = &padded[hLen+1];
431     pm = mask;
432     for(i = paddedSize-hLen-1; i > 0; i--)
433         *pm++ ^= *pp++;
434
435     // Make sure that the recovered data has the hash of the label
436     // Put trial value in the seed mask
437     if((retVal=_cpri__HashBlock(hashAlg, lSize, (BYTE *)label, hLen, seedMask)) < 0)
438         return retVal;
439
440     if(memcmp(seedMask, mask, hLen) != 0)
441         return CRYPT_FAIL;
442
443
444     // find the start of the data
445     pm = &mask[hLen];
446     for(i = paddedSize-(2*hLen)-1; i > 0; i--)
447     {
448         if(*pm++ != 0)
449             break;
450     }
451     if(i == 0)
452         return CRYPT_PARAMETER;
453
454     // pm should be pointing at the first part of the data
455     // and i is one greater than the number of bytes to move
456     i--;
457     if(i > dSizeSave)
458     {
459         // Restore dSize
460         *dataOutSize = dSizeSave;
461         return CRYPT_FAIL;
462     }
463     memcpy(dataOut, pm, i);
464     *dataOutSize = i;
465     return CRYPT_SUCCESS;
466 }

```

**B.9.1.3.7. PKSC1v1\_5Encode()**

This function performs the encoding for RSAES-PKCS1-V1\_5-ENCRYPT as defined in PKCS#1V2. 1

Return Value	Meaning
CRYPT_SUCCESS	data encoded
CRYPT_PARAMETER	message size is too large

```

467 static CRYPT_RESULT
468 RSAES_PKSC1v1_5Encode(
469     UINT32    paddedSize,    // IN: pad value size
470     BYTE      *padded,      // OUT: the pad data
471     UINT32    messageSize,  // IN: the message size
472     BYTE      *message      // IN: the message being padded
473 )
474 {
475     UINT32    ps = paddedSize - messageSize - 3;
476     if(messageSize > paddedSize - 11)
477         return CRYPT_PARAMETER;
478
479     // move the message to the end of the buffer
480     memcpy(&padded[paddedSize - messageSize], message, messageSize);
481
482     // Set the first byte to 0x00 and the second to 0x02
483     *padded = 0;
484     padded[1] = 2;
485
486     // Fill with random bytes
487     _cpri__GenerateRandom(ps, &padded[2]);
488
489     // Set the delimiter for the random field to 0
490     padded[2+ps] = 0;
491
492     // Now, the only messy part. Make sure that all the ps bytes are non-zero
493     // In this implementation, use the value of the current index
494     for(ps++; ps > 1; ps--)
495     {
496         if(padded[ps] == 0)
497             padded[ps] = 0x55;    // In the < 0.5% of the cases that the random
498                                   // value is 0, just pick a value to put into
499                                   // the spot.
500     }
501     return CRYPT_SUCCESS;
502 }

```

**B.9.1.3.8. RSAES\_Decode()**

This function performs the decoding for RSAES-PKCS1-V1\_5-ENCRYPT as defined in PKCS#1V2. 1

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decoding error or results would no fit into provided buffer

```

503 static CRYPT_RESULT
504 RSAES_Decode(
505     UINT32    *messageSize,  // IN/OUT: recovered message size
506     BYTE      *message,      // OUT: the recovered message
507     UINT32    codedSize,    // IN: the encoded message size
508     BYTE      *coded        // IN: the encoded message
509 )

```



```

510 {
511     BOOL        fail = FALSE;
512     UINT32      ps;
513
514     fail = (codedSize < 11);
515     fail |= (coded[0] != 0x00) || (coded[1] != 0x02);
516     for(ps = 2; ps < codedSize; ps++)
517     {
518         if(coded[ps] == 0)
519             break;
520     }
521     ps++;
522
523     // Make sure that ps has not gone over the end and that there are at least 8
524     // bytes of pad data.
525     fail |= ((ps >= codedSize) || ((ps-2) < 8));
526     if((*messageSize < codedSize - ps) || fail)
527         return CRYPT_FAIL;
528
529     *messageSize = codedSize - ps;
530     memcpy(message, &coded[ps], codedSize - ps);
531     return CRYPT_SUCCESS;
532 }

```

### B.9.1.3.9. 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.

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	hashAlg is not a supported hash algorithm

```

533 static CRYPT_RESULT
534 PssEncode (
535     UINT32      eOutSize,           // IN: size of the encode data buffer
536     BYTE        *eOut,             // OUT: encoded data buffer
537     TPM_ALG_ID  hashAlg,          // IN: hash algorithm to use for the encoding
538     UINT32      hashInSize,       // IN: size of digest to encode
539     BYTE        *hashIn,          // IN: the digest
540 #ifdef TEST_RSA
541     , BYTE        *saltIn         // IN: optional parameter for testing
542 #endif // TEST_RSA
543 )
544 {
545     INT32      hLen = _cpri_GetDigestSize(hashAlg);
546     BYTE      salt[MAX_RSA_KEY_BYTES - 1];
547     UINT16    saltSize;
548     BYTE      *ps = salt;
549     CRYPT_RESULT retVal;
550     UINT16    mLen;
551     CPRI_HASH_STATE hashState;
552
553     // These are fatal errors indicating bad TPM firmware
554     pAssert(eOut != NULL && hLen > 0 && hashIn != NULL );
555
556     // Get the size of the mask
557     mLen = (UINT16)(eOutSize - hLen - 1);
558
559     // Use the maximum salt size
560     saltSize = mLen - 1;
561
562 }

```

```

562 //using eOut for scratch space
563 // Set the first 8 bytes to zero
564 memset(eOut, 0, 8);
565
566
567 // Get set the salt
568 #ifndef TEST_RSA
569 if(saltIn != NULL)
570 {
571     saltSize = hLen;
572     memcpy(salt, saltIn, hLen);
573 }
574 else
575 #endif // TEST_RSA
576     _cpri__GenerateRandom(saltSize, salt);
577
578 // Create the hash of the pad || input hash || salt
579 _cpri__StartHash(hashAlg, FALSE, &hashState);
580 _cpri__UpdateHash(&hashState, 8, eOut);
581 _cpri__UpdateHash(&hashState, hashInSize, hashIn);
582 _cpri__UpdateHash(&hashState, saltSize, salt);
583 _cpri__CompleteHash(&hashState, hLen, &eOut[eOutSize - hLen - 1]);
584
585 // Create a mask
586 if((retVal = _cpri__MGF1(mLen, eOut, hashAlg, hLen, &eOut[mLen])) < 0)
587 {
588     // Currently _cpri__MGF1 is not expected to return a CRYPT_RESULT error.
589     pAssert(0);
590     return retVal;
591 }
592 // Since this implementation uses key sizes that are all even multiples of
593 // 8, just need to make sure that the most significant bit is CLEAR
594 eOut[0] &= 0x7f;
595
596 // Before we mess up the eOut value, set the last byte to 0xbc
597 eOut[eOutSize - 1] = 0xbc;
598
599 // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
600 eOut = &eOut[mLen - saltSize - 1];
601 *eOut++ ^= 0x01;
602
603 // XOR the salt data into the buffer
604 for(; saltSize > 0; saltSize--)
605     *eOut++ ^= *ps++;
606
607 // and we are done
608 return CRYPT_SUCCESS;
609 }

```

#### B.9.1.3.10. 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 enforced by this implementation. This implementation does allow for a variable size salt value to have been used by the creator of the signature.

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_SCHEME	<i>hashAlg</i> is not a supported hash algorithm
CRYPT_FAIL	decode operation failed

```

610 static CRYPT_RESULT
611 PssDecode (
612     TPM_ALG_ID    hashAlg,           // IN: hash algorithm to use for the encoding
613     UINT32        dInSize,          // IN: size of the digest to compare
614     BYTE          *dIn,             // IN: the digest to compare
615     UINT32        eInSize,          // IN: size of the encoded data
616     BYTE          *eIn,             // IN: the encoded data
617     UINT32        saltSize          // IN: the expected size of the salt
618 )
619 {
620     INT32          hLen = _cpri_GetDigestSize(hashAlg);
621     BYTE          mask[MAX_RSA_KEY_BYTES];
622     BYTE          *pm = mask;
623     BYTE          pad[8] = {0};
624     UINT32        i;
625     UINT32        mLen;
626     BOOL          fail = FALSE;
627     CRYPT_RESULT  retVal;
628     CPRI_HASH_STATE hashState;
629
630     // These errors are indicative of failures due to programmer error
631     pAssert(dIn != NULL && eIn != NULL);
632
633     // check the hash scheme
634     if(hLen == 0)
635         return CRYPT_SCHEME;
636
637     // most significant bit must be zero
638     fail = ((eIn[0] & 0x80) != 0);
639
640     // last byte must be 0xbc
641     fail |= (eIn[eInSize - 1] != 0xbc);
642
643     // Use the hLen bytes at the end of the buffer to generate a mask
644     // Doesn't start at the end which is a flag byte
645     mLen = eInSize - hLen - 1;
646     if((retVal = _cpri_MGF1(mLen, mask, hashAlg, hLen, &eIn[mLen])) < 0)
647         return retVal;
648     if(retVal == 0)
649         return CRYPT_FAIL;
650
651     // Clear the MS0 of the mask to make it consistent with the encoding.
652     mask[0] &= 0x7F;
653
654     // XOR the data into the mask to recover the salt. This sequence
655     // advances eIn so that it will end up pointing to the seed data
656     // which is the hash of the signature data
657     for(i = mLen; i > 0; i--)
658         *pm++ ^= *eIn++;
659
660     // Find the first byte of 0x01 after a string of all 0x00
661     for(pm = mask, i = mLen; i > 0; i--)
662     {
663         if(*pm == 0x01)
664             break;
665         else
666             fail |= (*pm++ != 0);
667     }

```

```

668     fail |= (i == 0);
669
670     // if we have failed, will continue using the entire mask as the salt value so
671     // that the timing attacks will not disclose anything (I don't think that this
672     // is a problem for TPM applications but, usually, we don't fail so this
673     // doesn't cost anything).
674     if(fail)
675     {
676         i = mLen;
677         pm = mask;
678     }
679     else
680     {
681         pm++;
682         i--;
683     }
684     // If the salt size was provided, then the recovered size must match
685     fail |= (saltSize != 0 && i != saltSize);
686
687     // i contains the salt size and pm points to the salt. Going to use the input
688     // hash and the seed to recreate the hash in the lower portion of eIn.
689     _cpri__StartHash(hashAlg, FALSE, &hashState);
690
691     // add the pad of 8 zeros
692     _cpri__UpdateHash(&hashState, 8, pad);
693
694     // add the provided digest value
695     _cpri__UpdateHash(&hashState, dInSize, dIn);
696
697     // and the salt
698     _cpri__UpdateHash(&hashState, i, pm);
699
700     // get the result
701     retVal = _cpri__CompleteHash(&hashState, MAX_DIGEST_SIZE, mask);
702
703     // retVal will be the size of the digest or zero. If not equal to the indicated
704     // digest size, then the signature doesn't match
705     fail |= (retVal != hLen);
706     fail |= (memcmp(mask, eIn, hLen) != 0);
707     if(fail)
708         return CRYPT_FAIL;
709     else
710         return CRYPT_SUCCESS;
711 }

```

#### B.9.1.3.11. PKSC1v1\_5SignEncode()

Encode a message using *PKCS1v1(). 5* method.

Return Value	Meaning
CRYPT_SUCCESS	encode complete
CRYPT_SCHEME	<i>hashAlg</i> is not a supported hash algorithm
CRYPT_PARAMETER	<i>eOutSize</i> is not large enough or <i>hInSize</i> does not match the digest size of <i>hashAlg</i>

```

712 static CRYPT_RESULT
713 RSASSA_Encode(
714     UINT32     eOutSize,      // IN: the size of the resulting block
715     BYTE      *eOut,         // OUT: the encoded block
716     TPM_ALG_ID hashAlg,     // IN: hash algorithm for PKSC1v1_5
717     UINT32     hInSize,     // IN: size of hash to be signed
718     BYTE      *hIn          // IN: hash buffer

```

```

719 )
720 {
721     BYTE            *der;
722     INT32           derSize = _cpri__GetHashDER(hashAlg, &der);
723     INT32           fillSize;
724
725     pAssert(eOut != NULL && hIn != NULL);
726
727     // Can't use this scheme if the algorithm doesn't have a DER string defined.
728     if(derSize == 0 )
729         return CRYPT_SCHEME;
730
731     // If the digest size of 'hashAlg' doesn't match the input digest size, then
732     // the DER will misidentify the digest so return an error
733     if((unsigned)_cpri__GetDigestSize(hashAlg) != hInSize)
734         return CRYPT_PARAMETER;
735
736     fillSize = eOutSize - derSize - hInSize - 3;
737
738     // Make sure that this combination will fit in the provided space
739     if(fillSize < 8)
740         return CRYPT_PARAMETER;
741     // Start filling
742     *eOut++ = 0; // initial byte of zero
743     *eOut++ = 1; // byte of 0x01
744     for(; fillSize > 0; fillSize--)
745         *eOut++ = 0xff; // bunch of 0xff
746     *eOut++ = 0; // another 0
747     for(; derSize > 0; derSize--)
748         *eOut++ = *der++; // copy the DER
749     for(; hInSize > 0; hInSize--)
750         *eOut++ = *hIn++; // copy the hash
751     return CRYPT_SUCCESS;
752 }

```

#### B.9.1.3.12. RSASSA\_Decode()

This function performs the RSASSA decoding of a signature.

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decode unsuccessful
CRYPT_SCHEME	<i>hashAlg</i> is not supported

```

753 static CRYPT_RESULT
754 RSASSA_Decode(
755     TPM_ALG_ID      hashAlg,           // IN: hash algorithm to use for the encoding
756     UINT32           hInSize,          // IN: size of the digest to compare
757     BYTE            *hIn,              // IN: the digest to compare
758     UINT32           eInSize,          // IN: size of the encoded data
759     BYTE            *eIn,              // IN: the encoded data
760 )
761 {
762     BOOL            fail = FALSE;
763     BYTE            *der;
764     INT32           derSize = _cpri__GetHashDER(hashAlg, &der);
765     INT32           hashSize = _cpri__GetDigestSize(hashAlg);
766     INT32           fillSize;
767
768     pAssert(hIn != NULL && eIn != NULL);
769
770     // Can't use this scheme if the algorithm doesn't have a DER string

```

```

771     // defined or if the provided hash isn't the right size
772     if(derSize == 0 || (unsigned)hashSize != hInSize)
773         return CRYPT_SCHEME;
774
775     // Make sure that this combination will fit in the provided space
776     // Since no data movement takes place, can just walk through this
777     // and accept nearly random values. This can only be called from
778     // _cpri__ValidateSignature() so eInSize is known to be in range.
779     fillSize = eInSize - derSize - hashSize - 3;
780
781     // Start checking
782     fail |= (*eIn++ != 0); // initial byte of zero
783     fail |= (*eIn++ != 1); // byte of 0x01
784     for(; fillSize > 0; fillSize--)
785         fail |= (*eIn++ != 0xff); // bunch of 0xff
786     fail |= (*eIn++ != 0); // another 0
787     for(; derSize > 0; derSize--)
788         fail |= (*eIn++ != *der++); // match the DER
789     for(; hInSize > 0; hInSize--)
790         fail |= (*eIn++ != *hIn++); // match the hash
791     if(fail)
792         return CRYPT_FAIL;
793     return CRYPT_SUCCESS;
794 }

```

#### B.9.1.4. Externally Accessible Functions

##### B.9.1.4.1. \_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.

```

795     BOOL
796     _cpri__RsaStartup(
797         void
798     )
799     {
800         return TRUE;
801     }

```

##### B.9.1.4.2. \_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 *RSAPublicEncrypt*. 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.

Return Value	Meaning
CRYPT_SUCCESS	encryption complete
CRYPT_PARAMETER	<i>cOutSize</i> is too small (must be the size of the modulus)
CRYPT_SCHEME	<i>padType</i> is not a supported scheme

```

802  CRYPT_RESULT
803  _cpri_EncryptRSA(
804      UINT32      *cOutSize,      // OUT: the size of the encrypted data
805      BYTE        *cOut,         // OUT: the encrypted data
806      RSA_KEY     *key,          // IN: the key to use for encryption
807      TPM_ALG_ID  padType,       // IN: the type of padding
808      UINT32      dInSize,       // IN: the amount of data to encrypt
809      BYTE        *dIn,          // IN: the data to encrypt
810      TPM_ALG_ID  hashAlg,       // IN: in case this is needed
811      const char  *label         // IN: in case it is needed
812  )
813  {
814      CRYPT_RESULT  retVal = CRYPT_SUCCESS;
815
816      pAssert(cOutSize != NULL);
817
818      // All encryption schemes return the same size of data
819      if(*cOutSize < key->publicKey->size)
820          return CRYPT_PARAMETER;
821      *cOutSize = key->publicKey->size;
822
823      switch (padType)
824      {
825      case TPM_ALG_NULL: // 'raw' encryption
826          {
827              // dIn can have more bytes than cOut as long as the extra bytes
828              // are zero
829              for(; dInSize > *cOutSize; dInSize--)
830              {
831                  if(*dIn++ != 0)
832                      return CRYPT_PARAMETER;
833              }
834              // If dIn is smaller than cOut, fill cOut with zeros
835              if(dInSize < *cOutSize)
836                  memset(cOut, 0, *cOutSize - dInSize);
837
838              // Copy the rest of the value
839              memcpy(&cOut[*cOutSize-dInSize], dIn, dInSize);
840              // If the size of dIn is the same as cOut dIn could be larger than
841              // the modulus. If it is, then RSAEP() will catch it.
842          }
843          break;
844      case TPM_ALG_RSAES:
845          retVal = RSAES_PKSC1v1_5Encode(*cOutSize, cOut, dInSize, dIn);
846          break;
847      case TPM_ALG_OAEP:
848          retVal = OaepEncode(*cOutSize, cOut, hashAlg, label, dInSize, dIn
849  #ifdef TEST_RSA
850          ,NULL
851  #endif
852          );
853          break;
854      default:
855          return CRYPT_SCHEME;
856      }
857      // All the schemes that do padding will come here for the encryption step
858      // Check that the Encoding worked

```

```

860     if(retval != CRYPT_SUCCESS)
861         return retVal;
862
863     // Padding OK so do the encryption
864     return RSAEP(*cOutSize, cOut, key);
865 }

```

### B.9.1.4.3. `_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.

Return Value	Meaning
CRYPT_SUCCESS	successful completion
CRYPT_PARAMETER	<i>cInSize</i> is not the same as the size of the public modulus of <i>key</i> , or numeric value of the encrypted data is greater than the modulus
CRYPT_FAIL	<i>dOutSize</i> is not large enough for the result
CRYPT_SCHEME	<i>padType</i> is not supported

```

866 CRYPT_RESULT
867 _cpri__DecryptRSA(
868     UINT32      *dOutSize,      // OUT: the size of the decrypted data
869     BYTE        *dOut,         // OUT: the decrypted data
870     RSA_KEY     *key,          // IN: the key to use for decryption
871     TPM_ALG_ID  padType,       // IN: the type of padding
872     UINT32      cInSize,       // IN: the amount of data to decrypt
873     BYTE        *cIn,          // IN: the data to decrypt
874     TPM_ALG_ID  hashAlg,       // IN: in case this is needed for the scheme
875     const char  *label         // IN: in case it is needed for the scheme
876 )
877 {
878     CRYPT_RESULT  retVal;
879
880     // Make sure that the necessary parameters are provided
881     pAssert(cIn != NULL && dOut != NULL && dOutSize != NULL && key != NULL);
882
883     // Size is checked to make sure that the decryption works properly
884     if(cInSize != key->publicKey->size)
885         return CRYPT_PARAMETER;
886
887     // For others that do padding, do the decryption in place and then
888     // go handle the decoding.
889     if((retVal = RSADP(cInSize, cIn, key)) != CRYPT_SUCCESS)
890         return retVal;      // Decryption failed
891
892     // Remove padding
893     switch (padType)
894     {
895     case TPM_ALG_NULL:
896         if(*dOutSize < key->publicKey->size)
897             return CRYPT_FAIL;
898         *dOutSize = key->publicKey->size;
899         memcpy(dOut, cIn, *dOutSize);
900         return CRYPT_SUCCESS;
901     case TPM_ALG_RSAES:
902         return RSAES_Decode(dOutSize, dOut, cInSize, cIn);
903         break;
904     case TPM_ALG_OAEP:
905         return OaepDecode(dOutSize, dOut, hashAlg, label, cInSize, cIn);
906         break;
907     default:

```



```

908     return CRYPT_SCHEME;
909     break;
910 }
911 }

```

#### B.9.1.4.4. `_cpri__SignRSA()`

This function is used to generate an RSA signature of the type indicated in *scheme*.

Return Value	Meaning
CRYPT_SUCCESS	sign operation completed normally
CRYPT_SCHEME	<i>scheme</i> or <i>hashAlg</i> are not supported
CRYPT_PARAMETER	<i>hInSize</i> does not match <i>hashAlg</i> (for RSASSA)

```

912 CRYPT_RESULT
913 _cpri__SignRSA(
914     UINT32      *sigOutSize,    // OUT: size of signature
915     BYTE        *sigOut,        // OUT: signature
916     RSA_KEY     *key,           // IN: key to use
917     TPM_ALG_ID  scheme,        // IN: the scheme to use
918     TPM_ALG_ID  hashAlg,       // IN: hash algorithm for PKSC1v1_5
919     UINT32      hInSize,       // IN: size of digest to be signed
920     BYTE        *hIn           // IN: digest buffer
921 )
922 {
923     CRYPT_RESULT  retVal;
924
925     // Parameter checks
926     pAssert(sigOutSize != NULL && sigOut != NULL && key != NULL && hIn != NULL);
927
928
929     // For all signatures the size is the size of the key modulus
930     *sigOutSize = key->publicKey->size;
931     switch (scheme)
932     {
933     case TPM_ALG_NULL:
934         *sigOutSize = 0;
935         return CRYPT_SUCCESS;
936     case TPM_ALG_RSAPSS:
937         // PssEncode can return CRYPT_PARAMETER
938         retVal = PssEncode(*sigOutSize, sigOut, hashAlg, hInSize, hIn
939 #ifdef TEST_RSA
940             , NULL
941 #endif
942             );
943         break;
944     case TPM_ALG_RSASSA:
945         // RSASSA_Encode can return CRYPT_PARAMETER or CRYPT_SCHEME
946         retVal = RSASSA_Encode(*sigOutSize, sigOut, hashAlg, hInSize, hIn);
947         break;
948     default:
949         return CRYPT_SCHEME;
950     }
951     if(retVal != CRYPT_SUCCESS)
952         return retVal;
953     // Do the encryption using the private key
954     // RSADP can return CRYPT_PARAMETER
955     return RSADP(*sigOutSize, sigOut, key);
956 }

```

**B.9.1.4.5. \_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.

Return Value	Meaning
CRYPT_SUCCESS	the signature checks
CRYPT_FAIL	the signature does not check
CRYPT_SCHEME	unsupported scheme or hash algorithm

```

957  CRYPT_RESULT
958  _cpri__ValidateSignatureRSA(
959      RSA_KEY      *key,           // IN: key to use
960      TPM_ALG_ID   scheme,        // IN: the scheme to use
961      TPM_ALG_ID   hashAlg,       // IN: hash algorithm
962      UINT32       hInSize,       // IN: size of digest to be checked
963      BYTE         *hIn,          // IN: digest buffer
964      UINT32       sigInSize,     // IN: size of signature
965      BYTE         *sigIn,        // IN: signature
966      UINT16       saltSize       // IN: salt size for PSS
967  )
968  {
969      CRYPT_RESULT  retVal;
970
971      // Fatal programming errors
972      pAssert(key != NULL && sigIn != NULL && hIn != NULL);
973
974      // Errors that might be caused by calling parameters
975      if(sigInSize != key->publicKey->size)
976          return CRYPT_FAIL;
977      // Decrypt the block
978      if((retVal = RSAEP(sigInSize, sigIn, key)) != CRYPT_SUCCESS)
979          return CRYPT_FAIL;
980      switch (scheme)
981      {
982      case TPM_ALG_NULL:
983          return CRYPT_SCHEME;
984          break;
985      case TPM_ALG_RSAPSS:
986          return PssDecode(hashAlg, hInSize, hIn, sigInSize, sigIn, saltSize);
987          break;
988      case TPM_ALG_RSASSA:
989          return RSASSA_Decode(hashAlg, hInSize, hIn, sigInSize, sigIn);
990          break;
991      default:
992          break;
993      }
994      return CRYPT_SCHEME;
995  }
996  #ifndef RSA_KEY_SIEVE        //%
```

**B.9.1.4.6. \_cpri\_\_GenerateKeyRSA()**

Generate an RSA key from a provided seed

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 cancelled

```

997 CRYPT_RESULT
998 _cpri_GenerateKeyRSA(
999     TPM2B          *n,           // OUT: The public modulus
1000    TPM2B          *p,           // OUT: One of the prime factors of n
1001    UINT16         keySizeInBits, // IN: Size of the public modulus in bits
1002    UINT32         e,           // IN: The public exponent
1003    TPM_ALG_ID     hashAlg,      // IN: hash algorithm to use in the key
1004                                     // generation process
1005    TPM2B          *seed,        // IN: the seed to use
1006    const char     *label,       // IN: A label for the generation process.
1007    TPM2B          *extra,       // IN: Party 1 data for the KDF
1008    UINT32         *counter      // IN/OUT: Counter value to allow KFD iteration
1009                                     // to be propagated across multiple
1010                                     // routines
1011 )
1012 {
1013     UINT32         lLen;         // length of the label
1014                                     // (counting the terminating 0);
1015     UINT16         digestSize = _cpri_GetDigestSize(hashAlg);
1016
1017     TPM2B_HASH_BLOCK  oPadKey;
1018
1019     UINT32         outer;
1020     UINT32         inner;
1021     BYTE          swapped[4];
1022
1023     CRYPT_RESULT   retVal;
1024     int            i, fill;
1025     const static char defaultLabel[] = "RSA key";
1026     BYTE          *pb;
1027
1028
1029     BYTE          h1[MAX_HASH_STATE_SIZE]; // contains the hash of the
1030                                     // HMAC key w/ iPad
1031     BYTE          h2[MAX_HASH_STATE_SIZE]; // contains the hash of the
1032                                     // HMAC key w/ oPad
1033     BYTE          h[MAX_HASH_STATE_SIZE]; // the working hash context
1034
1035     BIGNUM        *bnP;
1036     BIGNUM        *bnQ;
1037     BIGNUM        *bnT;
1038     BIGNUM        *bnE;
1039     BIGNUM        *bnN;
1040     BN_CTX        *context;
1041     UINT32        rem;
1042
1043     // Make sure that hashAlg is valid hash
1044     pAssert(digestSize != 0);
1045
1046     // if present, use externally provided counter
1047     if(counter != NULL)
1048         outer = *counter;
1049     else
1050         outer = 1;
1051
1052     // Validate exponent
1053     UINT32_TO_BYTE_ARRAY(e, swapped);
1054

```

```

1055 // Need to check that the exponent is prime and not less than 3
1056 if( e != 0 && (e < 3 || !_math__IsPrime(4, swapped)))
1057     return CRYPT_FAIL;
1058
1059 // Get structures for the big number representations
1060 context = BN_CTX_new();
1061 if(context == NULL)
1062     FAIL(FATAL_ERROR_ALLOCATION);
1063 BN_CTX_start(context);
1064 bnP = BN_CTX_get(context);
1065 bnQ = BN_CTX_get(context);
1066 bnT = BN_CTX_get(context);
1067 bnE = BN_CTX_get(context);
1068 bnN = BN_CTX_get(context);
1069 if(bnN == NULL)
1070     FAIL(FATAL_ERROR_INTERNAL);
1071
1072 // Set Q to zero. This is used as a flag. The prime is computed in P. When a
1073 // new prime is found, Q is checked to see if it is zero. If so, P is copied
1074 // to Q and a new P is found. When both P and Q are non-zero, the modulus and
1075 // private exponent are computed and a trial encryption/decryption is
1076 // performed. If the encrypt/decrypt fails, assume that at least one of the
1077 // primes is composite. Since we don't know which one, set Q to zero and start
1078 // over and find a new pair of primes.
1079 BN_zero(bnQ);
1080
1081 // Need to have some label
1082 if(label == NULL)
1083     label = (const char *)&defaultLabel;
1084 // Get the label size
1085 for(lLen = 0; label[lLen++] != 0);
1086
1087
1088 // Start the hash using the seed and get the intermediate hash value
1089 _cpri__StartHMAC(hashAlg, FALSE, &h1, seed->size, seed->buffer, &oPadKey.b);
1090 _cpri__StartHash(hashAlg, FALSE, &h2);
1091 _cpri__UpdateHash(&h2, oPadKey.b.size, oPadKey.b.buffer);
1092
1093 n->size = keySizeInBits/8;
1094 pAssert(n->size <= MAX_RSA_KEY_BYTES);
1095 p->size = n->size / 2;
1096 if(e == 0)
1097     e = RSA_DEFAULT_PUBLIC_EXPONENT;
1098
1099 BN_set_word(bnE, e);
1100
1101 // The first test will increment the counter from zero.
1102 for(outer += 1; outer != 0; outer++)
1103 {
1104     if(_plat__IsCanceled())
1105     {
1106         retVal = CRYPT_CANCEL;
1107         goto Cleanup;
1108     }
1109
1110     // Need to fill in the candidate with the hash
1111     fill = digestSize;
1112     pb = p->buffer;
1113
1114     // Reset the inner counter
1115     inner = 0;
1116     for(i = p->size; i > 0; i -= digestSize)
1117     {
1118         inner++;
1119         // Initialize the HMAC with saved state
1120         _cpri__CopyHashState(&h, &h1);

```

```

1121
1122 // Hash the inner counter (the one that changes on each HMAC iteration)
1123 UINT32_TO_BYTE_ARRAY(inner, swapped);
1124 _cpri_UpdateHash(&h, 4, swapped);
1125 _cpri_UpdateHash(&h, lLen, (BYTE *)label);
1126
1127 // Is there any party 1 data
1128 if(extra != NULL)
1129     _cpri_UpdateHash(&h, extra->size, extra->buffer);
1130
1131 // Include the outer counter (the one that changes on each prime
1132 // prime candidate generation
1133 UINT32_TO_BYTE_ARRAY(outer, swapped);
1134 _cpri_UpdateHash(&h, 4, swapped);
1135 _cpri_UpdateHash(&h, 2, (BYTE *)&keySizeInBits);
1136 if(i < fill)
1137     fill = i;
1138 _cpri_CompleteHash(&h, fill, pb);
1139
1140 // Restart the oPad hash
1141 _cpri_CopyHashState(&h, &h2);
1142
1143 // Add the last hashed data
1144 _cpri_UpdateHash(&h, fill, pb);
1145
1146 // gives a completed HMAC
1147 _cpri_CompleteHash(&h, fill, pb);
1148 pb += fill;
1149 }
1150 // Set the Most significant 2 bits and the low bit of the candidate
1151 p->buffer[0] |= 0xC0;
1152 p->buffer[p->size - 1] |= 1;
1153
1154 // Convert the candidate to a BN
1155 BN_bin2bn(p->buffer, p->size, bnP);
1156
1157 // If this is the second prime, make sure that it differs from the
1158 // first prime by at least 2^100
1159 if(!BN_is_zero(bnQ))
1160 {
1161     // bnQ is non-zero if we already found it
1162     if(BN_ucmp(bnP, bnQ) < 0)
1163         BN_sub(bnT, bnQ, bnP);
1164     else
1165         BN_sub(bnT, bnP, bnQ);
1166     if(BN_num_bits(bnT) < 100) <Q>// Difference has to be at least 100 bits
1167         continue;
1168 }
1169 // Make sure that the prime candidate (p) is not divisible by the exponent
1170 // and that (p-1) is not divisible by the exponent
1171 // Get the remainder after dividing by the modulus
1172 rem = BN_mod_word(bnP, e);
1173 if(rem == 0) // evenly divisible so add two keeping the number odd and
1174     // making sure that 1 != p mod e
1175     BN_add_word(bnP, 2);
1176 else if(rem == 1) // leaves a remainder of 1 so subtract two keeping the
1177     // number odd and making (e-1) = p mod e
1178     BN_sub_word(bnP, 2);
1179
1180 // Have a candidate, check for primality
1181 if((retVal = (CRYPT_RESULT)BN_is_prime_ex(bnP,
1182     BN_prime_checks, NULL, NULL)) < 0)
1183     FAIL(FATAL_ERROR_INTERNAL);
1184
1185 if(retVal != 1)
1186     continue;

```

```

1187
1188 // Found a prime, is this the first or second.
1189 if(BN_is_zero(bnQ))
1190 {
1191     // copy p to q and compute another prime in p
1192     BN_copy(bnQ, bnP);
1193     continue;
1194 }
1195 //Form the public modulus
1196 BN_mul(bnN, bnP, bnQ, context);
1197 if(BN_num_bits(bnN) != keySizeInBits)
1198     FAIL(FATAL_ERROR_INTERNAL);
1199
1200 // Save the public modulus
1201 BnTo2B(n, bnN, 0); // Fills the buffer with the correct size
1202
1203 // And one prime
1204 BnTo2B(p, bnP, 0);
1205
1206 // Finish by making sure that we can form the modular inverse of PHI
1207 // with respect to the public exponent
1208 // Compute PHI = (p - 1)(q - 1) = n - p - q + 1
1209 // Make sure that we can form the modular inverse
1210 BN_sub(bnT, bnN, bnP);
1211 BN_sub(bnT, bnT, bnQ);
1212 BN_add_word(bnT, 1);
1213
1214 // find d such that (Phi * d) mod e ==1
1215 // If there isn't then we are broken because we took the step
1216 // of making sure that the prime != 1 mod e so the modular inverse
1217 // must exist
1218 if(BN_mod_inverse(bnT, bnE, bnT, context) == NULL || BN_is_zero(bnT))
1219     FAIL(FATAL_ERROR_INTERNAL);
1220
1221 // Do a trial encryption and decryption of the seed to see if this
1222 // gives a valid result
1223 BN_bin2bn(seed->buffer, (n->size)-1, bnP);
1224 BN_copy(bnQ, bnP);
1225 BN_mod_exp(bnQ, bnQ, bnE, bnN, context);
1226 BN_mod_exp(bnQ, bnQ, bnT, bnN, context);
1227 if(BN_cmp(bnP, bnQ) != 0) // Trial encrypt decrypt failed. Start
1228     // over with new primes
1229 {
1230     BN_zero(bnQ);
1231     continue;
1232 }
1233 retVal = CRYPT_SUCCESS;
1234 goto Cleanup;
1235 }
1236 retVal = CRYPT_FAIL;
1237
1238
1239 Cleanup:
1240 // Close out the hash sessions
1241 _cpri__CompleteHash(&h2, 0, NULL);
1242 _cpri__CompleteHash(&h1, 0, NULL);
1243
1244 // Free up allocated BN values
1245 BN_CTX_end(context);
1246 BN_CTX_free(context);
1247 if(counter != NULL)
1248     *counter = outer;
1249 return retVal;
1250 }
1251 #endif // RSA_KEY_SIEVE //%
```

## B.9.2. Alternative RSA Key Generation

### B.9.2.1. Introduction

The files in this clause implement an alternative RSA key generation method that is about an order of magnitude faster than the regular method in B.9.1 and is provided simply to speed testing of the test functions. The method implemented in this clause uses a sieve rather than choosing prime candidates at random and testing for primeness. In this alternative, the sieve field 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.

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.9.2.2. RSAKeySieve.h

This header file is used to for parameterization of the Sieve and RNG used by the RSA module

```
1 #ifndef RSA_H
2 #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.

```
3 extern 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  $2^{16}$ .

```
4 #define PRIME_DIFF_TABLE_512_BYTE_PAGES 13
```

This set of macros used the value above to set the table size.

```
5 #ifndef PRIME_DIFF_TABLE_512_BYTE_PAGES
6 #   define PRIME_DIFF_TABLE_512_BYTE_PAGES 4
7 #endif
8 #ifdef PRIME_DIFF_TABLE_512_BYTE_PAGES
9 #   if PRIME_DIFF_TABLE_512_BYTE_PAGES > 12
10 #       define PRIME_DIFF_TABLE_BYTES 6542
11 #   else
12 #       if PRIME_DIFF_TABLE_512_BYTE_PAGES <= 0
13 #           define PRIME_DIFF_TABLE_BYTES 512
14 #       else
15 #           define PRIME_DIFF_TABLE_BYTES (PRIME_DIFF_TABLE_512_BYTE_PAGES * 512)
16 #       endif
17 #   endif
18 #endif
19 extern BYTE primeDiffTable[PRIME_DIFF_TABLE_BYTES];
```

This determines the number of bits in the sieve field This must be a power of two.

```

20 #define FIELD_POWER      14 // This is the only value in this group that should be
21 // changed
22 #define FIELD_BITS      (1 << FIELD_POWER)
23 #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.

```

24 #define SEED_VALUES_SIZE      105
25 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.

```

26 const extern BYTE            bitsInByte[256];

```

This is the iterator structure for accessing the compressed prime number table. The expectation is that values will need to be accessed sequentially. This tries to save some data access.

```

27 typedef struct {
28     UINT16      lastPrime;
29     UINT16      index;
30     UINT16      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

```

### B.9.2.3. RSAKeySieve.c

```

1  /*(Copyright)
2     Microsoft Copyright 2009, 2010, 2011, 2012
3     Microsoft Confidential Contribution to a TCG Specification or Design Guide
4     under Article 15 of "The Bylaws of the Trusted Computing Group" as Amended
5     through March 20, 2003
6
7  */
8
9  /** Introduction
10
11
12  #include "CryptoEngine.h"
13  #ifdef RSA_KEY_SIEVE //%
```



```

14
15 // This next line will show up in the header file for this code. It will
16 // make the local functions public when debugging.
17 // #ifdef  RSA_DEBUG
18
19 /** Bit Manipulation Functions
20 **** Introduction
21 // These functions operate on a bit array. A bit array is an array of
22 // bytes with the 0th byte being the byte with the lowest memory address.
23 // Within the byte, bit 0 is the least significant bit.
24
25 **** ClearBit()
26 // This function will CLEAR a bit in a bit array.
27 void
28 ClearBit(
29     unsigned char    *a,           // IN: A pointer to an array of bytes
30     int              i             // IN: the number of the bit to CLEAR
31 )
32 {
33     a[i >> 3] &= 0xff ^ (1 << (i & 7));
34 }

```

#### B.9.2.3.1.1. SetBit()

Function to SET a bit in a bit array.

```

35 void
36 SetBit(
37     unsigned char    *a,           // IN: A pointer to an array of bytes
38     int              i             // IN: the number of the bit to SET
39 )
40 {
41     a[i >> 3] |= (1 << (i & 7));
42 }

```

#### B.9.2.3.1.2. IsBitSet()

Function to test if a bit in a bit array is SET.

Return Value	Meaning
0	bit is CLEAR
1	bit is SET

```

43 UINT32
44 IsBitSet(
45     unsigned char    *a,           // IN: A pointer to an array of bytes
46     int              i             // IN: the number of the bit to test
47 )
48 {
49     return ((a[i >> 3] & (1 << (i & 7))) != 0);
50 }

```

#### B.9.2.3.1.3. BitsInArray()

This function counts the number of bits set in an array of bytes.

```

51 int
52 BitsInArray(
53     unsigned char    *a,           // IN: A pointer to an array of bytes

```

```

54     int          i                // IN: the number of bytes to sum
55     )
56 {
57     int          j = 0;
58     for(; i ; i--)
59         j += bitsInByte[*a++];
60     return j;
61 }

```

#### B.9.2.3.1.4. 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

```

62  UINT32
63  FindNthSetBit(
64      const UINT16    aSize,      // IN: the size of the array to check
65      const BYTE      *a,         // IN: the array to check
66      const UINT32    n           // IN, the number of the SET bit
67  )
68  {
69      UINT32          i;
70      const BYTE      *pA = a;
71      UINT32          retValue;
72      BYTE            sel;
73
74      //find the bit
75      for(i = 0; i < n; i += bitsInByte[*pA++]);
76
77      // The chosen bit is in the byte that was just accessed
78      // Compute the offset to the start of that byte
79      pA--;
80      retValue = (pA - a) * 8;
81
82      // Subtract the bits in the last byte added.
83      i -= bitsInByte[*pA];
84
85      // Now process the byte, one bit at a time.
86      for(sel = *pA; sel != 0 ; sel = sel >> 1)
87      {
88          if(sel & 1)
89          {
90              i += 1;
91              if(i == n)
92                  return retValue;
93          }
94          retValue += 1;
95      }
96      FAIL(FATAL_ERROR_INTERNAL);
97      return 0; // This is just to keep the compiler from complaining
98  }

```

#### B.9.2.3.2. Miscellaneous Functions

##### B.9.2.3.2.1. RandomForRsa()

This function uses a special form of *KDFa()* to produce a pseudo random sequence. Its 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 *ctx.out* will not wrap to zero if incremented. If so, the function returns *FALSE*. Otherwise, the *ctx.out* is incremented before each number is generated.

```

99  void
100 RandomForRsa(
101     KDFa_CONTEXT      *ktx,          // IN: a context for the KDF
102     const char        *label,       // IN: a use qualifying label
103     TPM2B             *p,           // OUT: the pseudo random result
104 )
105 {
106     INT16             i;
107     UINT32            inner;
108     BYTE              swapped[4];
109     UINT16            fill;
110     BYTE              *pb;
111     UINT16            lLen = 0;
112     UINT16            digestSize = _cpri__GetDigestSize(ktx->hashAlg);
113     BYTE              h[MAX_HASH_STATE_SIZE]; // the working hash context
114
115     if(label != NULL)
116         for(lLen = 0; label[lLen++]);
117     fill = digestSize;
118     pb = p->buffer;
119     inner = 0;
120     *(ktx->outer) += 1;
121     for(i = p->size; i > 0; i -= digestSize)
122     {
123         inner++;
124
125         // Initialize the HMAC with saved state
126         _cpri__CopyHashState(&h, &(ktx->iPadCtx));
127
128         // Hash the inner counter (the one that changes on each HMAC iteration)
129         UINT32_TO_BYTE_ARRAY(inner, swapped);
130         _cpri__UpdateHash(&h, 4, swapped);
131         if(lLen != 0)
132             _cpri__UpdateHash(&h, lLen, (BYTE *)label);
133
134         // Is there any party 1 data
135         if(ktx->extra != NULL)
136             _cpri__UpdateHash(&h, ktx->extra->size, ktx->extra->buffer);
137
138         // Include the outer counter (the one that changes on each prime
139         // prime candidate generation
140         UINT32_TO_BYTE_ARRAY(*(ktx->outer), swapped);
141         _cpri__UpdateHash(&h, 4, swapped);
142         _cpri__UpdateHash(&h, 2, (BYTE *)&ktx->keySizeInBits);
143         if(i < fill)
144             fill = i;
145         _cpri__CompleteHash(&h, fill, pb);
146
147         // Restart the oPad hash
148         _cpri__CopyHashState(&h, &(ktx->oPadCtx));
149
150         // Add the last hashed data
151         _cpri__UpdateHash(&h, fill, pb);
152
153         // gives a completed HMAC
154         _cpri__CompleteHash(&h, fill, pb);
155         pb += fill;
156     }
157     return;
158 }

```

**B.9.2.3.2.2. 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.

```

159  UINT32
160  MillerRabinRounds(
161      UINT32      bits           // IN: Number of bits in the RSA prime
162  )
163  {
164      if(bits < 511) <K>return 8;    // don't really expect this
165      if(bits < 1536) <K>return 5;  // for 512 and 1K primes
166      return 4;                    // for 3K public modulus and greater
167  }
```

**B.9.2.3.2.3. MillerRabin()**

This function performs a Miller-Rabin test from FIPS 186-3. It does *iterations* trials on the number. If 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.

Return Value	Meaning
TRUE	probably prime
FALSE	composite

```

168  BOOL
169  MillerRabin(
170      BIGNUM          *bnW,
171      int             iterations,
172      KDFa_CONTEXT   *ktx,
173      BN_CTX         *context
174  )
175  {
176      BIGNUM          *bnWm1;
177      BIGNUM          *bnM;
178      BIGNUM          *bnB;
179      BIGNUM          *bnZ;
180      BOOL            ret = FALSE;    // Assumed composite for easy exit
181      TPM2B_TYPE(MAX_PRIME, MAX_RSA_KEY_BYTES/2);
182      TPM2B_MAX_PRIME  b;
183      int              a;
184      int              j;
185      int              wLen;
186      int              i;
187
188      pAssert(BN_is_bit_set(bnW, 0));
189      INSTRUMENT_INC(MillerRabinTrials); // Instrumentation
190
191      BN_CTX_start(context);
192      bnWm1 = BN_CTX_get(context);
193      bnB = BN_CTX_get(context);
194      bnZ = BN_CTX_get(context);
195      bnM = BN_CTX_get(context);
196      if(bnM == NULL)
197          FAIL(FATAL_ERROR_ALLOCATION);
198
199      // Let a be the largest integer such that 2^a divides w-1.
200      BN_copy(bnWm1, bnW);
201      BN_sub_word(bnWm1, 1);
```

```

202     // Since w is odd (w-1) is even so start at bit number 1 rather than 0
203     for(a = 1; !BN_is_bit_set(bnWm1, a); a++);
204
205 // 2. m = (w-1) / 2^a
206     BN_rshift(bnM, bnWm1, a);
207
208 // 3. wlen = len (w).
209     wLen = BN_num_bits(bnW);
210     pAssert((wLen & 7) == 0);
211
212     // Set the size for the random number
213     b.b.size = (wLen + 7)/8;
214
215 // 4. For i = 1 to iterations do
216     for(i = 0; i < iterations ; i++)
217     {
218
219 // 4.1 Obtain a string b of wlen bits from an RBG.
220 step4point1:
221     // In the reference implementation, wLen is always a multiple of 8
222     if(ktx != NULL)
223         RandomForRsa(ktx, "Miller-Rabin witness", &b.b);
224     else
225         _cpri__GenerateRandom(b.t.size, b.t.buffer);
226
227     if(BN_bin2bn(b.t.buffer, b.t.size, bnB) == NULL)
228         FAIL(FATAL_ERROR_ALLOCATION);
229
230 // 4.2 If ((b ≤ 1) or (b ≥ w-1)), then go to step 4.1.
231     if(BN_is_zero(bnB))
232         goto step4point1;
233     if(BN_is_one(bnB))
234         goto step4point1;
235     if(BN_ucmp(bnB, bnWm1) >= 0)
236         goto step4point1;
237
238 // 4.3 z = b^m mod w.
239     if(BN_mod_exp(bnZ, bnB, bnM, bnW, context) != 1)
240         FAIL(FATAL_ERROR_ALLOCATION);
241
242 // 4.4 If ((z = 1) or (z = w - 1)), then go to step 4.7.
243     if(BN_is_one(bnZ) || BN_ucmp(bnZ, bnWm1) == 0)
244         goto step4point7;
245
246 // 4.5 For j = 1 to a - 1 do.
247     for(j = 1; j < a; j++)
248     {
249 // 4.5.1 z = z^2 mod w.
250         if(BN_mod_mul(bnZ, bnZ, bnZ, bnW, context) != 1)
251             FAIL(FATAL_ERROR_ALLOCATION);
252
253 // 4.5.2 If (z = w-1), then go to step 4.7.
254         if(BN_ucmp(bnZ, bnWm1) == 0)
255             goto step4point7;
256
257 // 4.5.3 If (z = 1), then go to step 4.6.
258         if(BN_is_one(bnZ))
259             goto step4point6;
260     }
261 // 4.6 Return COMPOSITE.
262 step4point6:
263     if(i > 9)
264         INSTRUMENT_INC(failedAtIteration[9]);
265     else
266         INSTRUMENT_INC(failedAtIteration[i]);
267     goto end;

```

```

268
269 // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
270 step4point7:
271     continue;
272 }
273 // 5. Return PROBABLY PRIME
274     ret = TRUE;
275
276 end:
277     BN_CTX_end(context);
278     return ret;
279 }

```

#### B.9.2.3.2.4. NextPrime()

This function is used to access the next prime number in the sequence of primes. It requires a pre-initialized iterator.

```

280 UINT16
281 NextPrime(
282     PRIME_ITERATOR    *iter
283 )
284 {
285     if(iter->index >= iter->final)
286         return (iter->lastPrime = 0);
287     return (iter->lastPrime += primeDiffTable[iter->index++]);
288 }

```

#### B.9.2.3.2.5. AdjustNumberOfPrimes()

Modifies the input parameter to be a valid value for the number of primes. The adjusted value is either the input value rounded up to the next 512 bytes boundary or the maximum value of the implementation. If the input is 0, the return is set to the maximum.

```

289 UINT16
290 AdjustNumberOfPrimes(
291     UINT16    p
292 )
293 {
294     p = ((p + 511) / 512) * 512;
295     if(p == 0 || p > PRIME_DIFF_TABLE_BYTES)
296         p = PRIME_DIFF_TABLE_BYTES;
297     return p;
298 }

```

#### B.9.2.3.2.6. PrimeInit()

This function is used to initialize the prime sequence generator iterator. The iterator is initialized and returns the first prime that is equal to the requested starting value. If the starting value is no a prime, then the iterator is initialized to the next higher prime number.

```

299 UINT16
300 PrimeInit(
301     UINT16    first,           // IN: the initial prime
302     PRIME_ITERATOR *iter,     // IN/OUT: the iterator structure
303     UINT16    primes         // IN: the table length
304 )
305 {
306
307     iter->lastPrime = 1;

```

```

308     iter->index = 0;
309     iter->final = AdjustNumberOfPrimes(primes);
310     while(iter->lastPrime < first)
311         NextPrime(iter);
312     return iter->lastPrime;
313 }

```

#### B.9.2.3.2.7. SetDefaultNumberOfPrimes()

This macro sets the default number of primes to the indicated value.

```

314 // #define SetDefaultNumberOfPrimes(p) (primeTableBytes = AdjustNumberOfPrimes(p))

```

#### B.9.2.3.2.8. IsPrimeWord()

Checks to see if a UINT32 is prime

Return Value	Meaning
TRUE	number is prime
FAIL	number is not prime

```

315 BOOL
316 IsPrimeWord(
317     UINT32    p    // IN: number to test
318 )
319 {
320     #if defined RSA_KEY_SIEVE && (PRIME_DIFF_TABLE_BYTES >= 6542)
321
322         UINT32    test;
323         UINT32    index;
324         UINT32    stop;
325
326         if((p & 1) == 0)
327             return FALSE;
328         if(p == 1 || p == 3)
329             return TRUE;
330
331         // Get a high value for the stopping point
332         for(index = p, stop = 0; index; index >>= 2)
333             stop = (stop << 1) + 1;
334         stop++;
335
336         // If the full prime difference value table is present, can check here
337
338         test = 3;
339         for(index = 1; index < PRIME_DIFF_TABLE_BYTES; index += 1)
340         {
341             if((p % test) == 0)
342                 return (p == test);
343             if(test > stop)
344                 return TRUE;
345             test += primeDiffTable[index];
346         }
347         return TRUE;
348     #else
349     #else
350
351         BYTE        b[4];
352         if(p = RSA_DEFAULT_PUBLIC_EXPONENT || p == 1 || p == 3 )
353             return TRUE;
354         if((p & 1) == 0)

```

```

355     return FALSE;
356     UINT32_TO_BYTE_ARRAY(p,b);
357     return _math__IsPrime(4, b);
358 #endif
359 }

```

#### B.9.2.3.2.9. SetDefaultSieveFieldSize()

This function sets the default sieve field size to the indicated value which should be a power of two. If not, the value is rounded down

```

360 void
361 SetDefaultSieveFieldSize(
362     UINT16    f    // IN: the size of the sieve field. This should be
363                 //     a power of two. The actual field size will be one
364                 //     byte larger.
365 )
366 {
367     UINT16    i;
368     if(f == 0 || f > MAX_FIELD_SIZE)
369         defaultFieldSize = MAX_FIELD_SIZE;
370     else
371     {
372         for(i = 1, f >>= 1; f != 0; f >>= 1, i <<= 1);
373         defaultFieldSize = i + 1;
374     }
375 }
376 typedef struct {
377     UINT16    prime;
378     UINT16    count;
379 } SIEVE_MARKS;
380 const SIEVE_MARKS sieveMarks[5] = {
381     {31, 7}, {73, 5}, {241, 4}, {1621, 3}, {UINT16_MAX, 2}};

```

#### B.9.2.3.2.10. PrimeSieve()

This function does a prime sieve over the input *field* which has as its starting address the value in *bnN*. Since this initializes the Sieve using a pre-computed field with the bits associated with 3, 5 and 7 already turned off, the value of *pnN* may need to be adjusted by a few counts to allow the pre-computed field to be used without modification. The *fieldSize* parameter must be  $2^N + 1$  and is probably not useful if it is less than 129 bytes (1024 bits).

```

382 UINT16
383 PrimeSieve(
384     BIGNUM    *bnN,        // IN/OUT: number to sieve
385     UINT16    fieldSize,  // IN: size of the field area in bytes
386     BYTE      *field,     // IN: field
387     UINT16    primes      // IN: the number of primes to use
388 )
389 {
390     UINT16    i;
391     UINT32    j;
392     UINT16    fieldBits = fieldSize * 8;
393     UINT32    r;
394     const BYTE *p1;
395     BYTE      *p2;
396     PRIME_ITERATOR iter;
397     UINT32    adjust;
398     UINT32    mark = 0;
399     UINT16    count = sieveMarks[0].count;
400     UINT16    stop = sieveMarks[0].prime;
401     UINT32    composite;

```



```

402
403 //   UINT64           test;           //DEBUG
404
405 pAssert(field != NULL && bnN != NULL);
406 // Need to have a field that has a size of 2^n + 1 bytes
407 pAssert(BitsInArray((BYTE *)&fieldSize, 2) == 2);
408
409 primes = AdjustNumberOfPrimes(primes);
410
411 // If the remainder is odd, then subtracting the value
412 // will give an even number, but we want an odd number,
413 // so subtract the 105+rem. Otherwise, just subtract
414 // the even remainder.
415 adjust = BN_mod_word(bnN,105);
416 if(adjust & 1)
417     adjust += 105;
418
419 // seed the field
420 // This starts the pointer at the nearest byte to the input value
421 p1 = &seedValues[adjust/16];
422
423 // Reduce the number of bytes to transfer by the amount skipped
424 j = sizeof(seedValues) - adjust/16;
425 adjust = adjust % 16;
426 BN_sub_word(bnN, adjust);
427 adjust >>= 1;
428
429 // This offsets the field
430 p2 = field;
431 for(i = fieldSize; i > 0; i--)
432 {
433     *p2++ = *p1++;
434     if(--j == 0)
435     {
436         j = sizeof(seedValues);
437         p1 = seedValues;
438     }
439 }
440 // Mask the first bits in the field and the last byte in order to eliminate
441 // bytes not in the field from consideration.
442 field[0] &= 0xff << adjust;
443 field[fieldSize-1] &= 0xff >> (8 - adjust);
444
445 // Cycle through the primes, clearing bits
446 // Have already done 3, 5, and 7
447 PrimeInit(7, &iter, primes);
448
449 // Get the next N primes where N is determined by the mark in the sieveMarks
450 while(composite = NextPrime(&iter))
451 {
452     UINT16 pList[8];
453     UINT16 next = 0;
454     i = count;
455     pList[i--] = composite;
456     for(; i > 0; i--)
457     {
458         next = NextPrime(&iter);
459         pList[i] = next;
460         if(next != 0)
461             composite *= next;
462     }
463     composite = BN_mod_word(bnN, composite);
464     for(i = count; i > 0; i--)
465     {
466         next = pList[i];
467         if(next == 0)

```

```

468         goto done;
469         r = composite % next;
470         if(r & 1)           j = (next - r)/2;
471         else if(r == 0)     j = 0;
472         else                j = next - r/2;
473         for(; j < fieldBits; j += next)
474             ClearBit(field, j);
475     }
476     if(next >= stop)
477     {
478         mark++;
479         count = sieveMarks[mark].count;
480         stop = sieveMarks[mark].prime;
481     }
482 }
483 done:
484 INSTRUMENT_INC(totalFieldsSieved);
485 i = BitsInArray(field, fieldSize);
486 if(i == 0) INSTRUMENT_INC(emptyFieldsSieved);
487 return i;
488 }

```

#### B.9.2.3.2.11. PrimeSelectWithSieve()

This function will sieve the field around the input prime candidate. If the sieve field is not empty, one of the one bits in the field is chosen for testing with Miller-Rabin. If the value is prime, *pnP* is updated with this value and the function returns success. If this value is not prime, another pseudo-random candidate is chosen and tested. This process repeats until all values in the field have been checked. If all bits in the field have been checked and none is prime, the function returns FALSE and a new random value needs to be chosen.

```

489 BOOL
490 PrimeSelectWithSieve(
491     BIGNUM          *bnP,           // IN/OUT: The candidate to filter
492     KDFa_CONTEXT    *ktx,         // IN: KDFa iterator structure
493     UINT32          e,             // IN: the exponent
494     BN_CTX          *context       // IN: the big number context to play in
495 #ifdef RSA_DEBUG           // %
496     ,UINT16         fieldSize,     // IN: number of bytes in the field, as
497                               // determined by the caller
498     UINT16          primes        // IN: number of primes to use.
499 #endif                     // %
500 )
501 {
502     BYTE            field[MAX_FIELD_SIZE];
503     UINT32          first;
504     UINT32          ones;
505     INT32           chosen;
506     UINT16          rounds = MillerRabinRounds(BN_num_bits(bnP));
507 #ifdef RSA_DEBUG
508     UINT16          primes;
509     UINT32          fieldSize;
510     // Adjust the field size and prime table list to fit the size of the prime
511     // being tested.
512     primes = BN_num_bits(bnP);
513     if(primes <= 512)
514     {
515         primes = AdjustNumberOfPrimes(2048);
516         fieldSize = 65;
517     }
518     else if(primes <= 1024)
519     {
520         primes = AdjustNumberOfPrimes(4096);

```

```

521     fieldSize = 129;
522 }
523 else
524 {
525     primes = AdjustNumberOfPrimes(0); // Set to the maximum
526     fieldSize = MAX_FIELD_SIZE;
527 }
528 if(fieldSize > MAX_FIELD_SIZE)
529     fieldSize = MAX_FIELD_SIZE;
530 #endif
531
532 // Save the low-order word to use as a search generator and make sure that
533 // it has some interesting range to it
534 first = bnP->d[0] | 0x80000000;
535
536 // Align to field boundary
537 bnP->d[0] &= ~((UINT32)(fieldSize-3));
538 pAssert(BN_is_bit_set(bnP, 0));
539 bnP->d[0] &= (UINT32_MAX << (FIELD_POWER + 1)) + 1;
540 ones = PrimeSieve(bnP, fieldSize, field, primes);
541 #ifndef RSA_FILTER_DEBUG
542 pAssert(ones == BitsInArray(field, defaultFieldSize));
543 #endif
544 for(; ones > 0; ones--)
545 {
546 #ifndef RSA_FILTER_DEBUG
547     if(ones != BitsInArray(field, defaultFieldSize))
548         FAIL(FATAL_ERROR_INTERNAL);
549 #endif
550 // Decide which bit to look at and find its offset
551 if(ones == 1)
552     ones = ones;
553 chosen = FindNthSetBit(defaultFieldSize, field, ((first % ones) + 1));
554 if(chosen >= ((defaultFieldSize) * 8))
555     FAIL(FATAL_ERROR_INTERNAL);
556
557
558 // Set this as the trial prime
559 BN_add_word(bnP, chosen * 2);
560
561 // Use MR to see if this is prime
562 if(MillerRabin(bnP, rounds, ktx, context))
563 {
564     // Final check is to make sure that 0 != (p-1) mod e
565     // This is the same as -1 != p mod e ; or
566     // (e - 1) != p mod e
567     if((e <= 3) || (BN_mod_word(bnP, e) != (e-1)))
568         return TRUE;
569 }
570 // Back out the bit number
571 BN_sub_word(bnP, chosen * 2);
572
573 // Clear the bit just tested
574 ClearBit(field, chosen);
575 }
576 // Ran out of bits and couldn't find a prime in this field
577 INSTRUMENT_INC(noPrimeFields);
578 return FALSE;
579 }

```

#### B.9.2.3.2.12. AdjustPrimeCandidate()

This function adjusts the candidate prime so that it is odd and  $> \text{root}(2)/2$ . This allows the product of these two numbers to be . 5, which, in fixed point notation means that the most significant bit is 1. For this

routine, the  $\sqrt{2}/2$  is approximated with  $0xB505$  which is, in fixed point is 0.7071075439453125 or an error of 0.0001%. Just setting the upper two bits would give a value  $> 0.75$  which is an error of  $> 6\%$ . Given the amount of time all the other computations take, reducing the error is not much of a cost, but it isn't totally required either.

The function also puts the number on a field boundary.

```

580 void
581 AdjustPrimeCandidate(
582     BYTE      *a,
583     UINT16    len
584 )
585 {
586     UINT16    highBytes;
587
588     highBytes = BYTE_ARRAY_TO_UINT16(a);
589     // This is fixed point arithmetic on 16-bit values
590     highBytes = ((UINT32)highBytes * (UINT32)0x4AFB) >> 16;
591     highBytes += 0xB505;
592     UINT16_TO_BYTE_ARRAY(highBytes, a);
593     a[len-1] |= 1;
594 }

```

#### B.9.2.3.2.13. GenerateRandomPrime()

```

595 void
596 GenerateRandomPrime(
597     TPM2B     *p,
598     BN_CTX    *ctx
599 #ifdef RSA_DEBUG    //%
600     ,UINT16    field,
601     UINT16    primes
602 #endif    //%
603 )
604 {
605     BIGNUM    *bnP;
606     BN_CTX    *context;
607
608     if(ctx == NULL) context = BN_CTX_new();
609     else context = ctx;
610     if(context == NULL)
611         FAIL(FATAL_ERROR_ALLOCATION);
612     BN_CTX_start(context);
613     bnP = BN_CTX_get(context);
614
615     while(TRUE)
616     {
617         _cpri_GenerateRandom(p->size, p->buffer);
618         p->buffer[p->size-1] |= 1;
619         p->buffer[0] |= 0x80;
620         BN_bin2bn(p->buffer, p->size, bnP);
621 #ifdef RSA_DEBUG
622         if(PrimeSelectWithSieve(bnP, NULL, 0, context, field, primes))
623 #else
624         if(PrimeSelectWithSieve(bnP, NULL, 0, context))
625 #endif
626             break;
627     }
628     BnTo2B(p, bnP, BN_num_bytes(bnP));
629     BN_CTX_end(context);
630     if(ctx == NULL)
631         BN_CTX_free(context);
632     return;
633 }

```

```

634 KDFa_CONTEXT *
635 KDFaContextStart(
636     KDFa_CONTEXT      *ktx,           // IN/OUT: the context structure to
637                               // initialize
638     TPM2B              *seed,         // IN: the seed for the digest process
639     TPM2B              hashAlg,       // IN: the hash algorithm
640     TPM2B              *extra,        // IN: the extra data
641     UINT32             *outer,        // IN: the outer iteration counter
642     UINT16             keySizeInBits
643 )
644 {
645     UINT16             digestSize = _cpri__GetDigestSize(hashAlg);
646     TPM2B_HASH_BLOCK  oPadKey;
647
648     if(seed == NULL)
649         return NULL;
650
651     pAssert(ktx != NULL && outer != NULL && digestSize != 0);
652
653     // Start the hash using the seed and get the intermediate hash value
654     _cpri__StartHMAC(hashAlg, FALSE, &(ktx->iPadCtx), seed->size, seed->buffer,
655                       &oPadKey.b);
656     _cpri__StartHash(hashAlg, FALSE, &(ktx->oPadCtx));
657     _cpri__UpdateHash(&(ktx->oPadCtx), oPadKey.b.size, oPadKey.b.buffer);
658     ktx->extra = extra;
659     ktx->hashAlg = hashAlg;
660     ktx->outer = outer;
661     ktx->keySizeInBits = keySizeInBits;
662     return ktx;
663 }
664 void
665 KDFaContextEnd(
666     KDFa_CONTEXT      *ktx           // IN/OUT: the context structure to close
667 )
668 {
669     if(ktx != NULL)
670     {
671         // Close out the hash sessions
672         _cpri__CompleteHash(&(ktx->iPadCtx), 0, NULL);
673         _cpri__CompleteHash(&(ktx->oPadCtx), 0, NULL);
674     }
675 }
676 //#endif

```

### B.9.2.3.3. Public Function

#### B.9.2.3.3.1. Introduction

This is the external entry for this replacement function. All this file provides is the substitute function to generate an RSA key. If the compiler settings are set appropriately, this this function will be used instead of the similarly named function in CpriRSA.c.

#### B.9.2.3.3.2. `_cpri__GenerateKeyRSA()`

Generate an RSA key from a provided seed

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 cancelled

```

677 CRYPT_RESULT
678 _cpri_GenerateKeyRSA(
679     TPM2B      *n,           // OUT: The public modulus
680     TPM2B      *p,           // OUT: One of the prime factors of n
681     UINT16     keySizeInBits, // IN: Size of the public modulus in bits
682     UINT32     e,           // IN: The public exponent
683     TPM_ALG_ID hashAlg,     // IN: hash algorithm to use in the key
684                     // generation process
685     TPM2B      *seed,       // IN: the seed to use
686     const char *label,     // IN: A label for the generation process.
687     TPM2B      *extra,      // IN: Party 1 data for the KDF
688     UINT32     *counter     // IN/OUT: Counter value to allow KDF
689                     // iteration to be propagated across
690                     // multiple routines
691     #ifdef RSA_DEBUG        //%
692     ,UINT16     primes,     // IN: number of primes to test
693     UINT16     fieldSize   // IN: the field size to use
694     #endif                //%
695 )
696 {
697     CRYPT_RESULT      retVal;
698     UINT32            myCounter = 0;
699     UINT32            *pCtr = (counter == NULL) ? &myCounter : counter;
700
701     KDFa_CONTEXT      ktx;
702     KDFa_CONTEXT      *ktxPtr;
703     UINT32            i;
704     BIGNUM            *bnP;
705     BIGNUM            *bnQ;
706     BIGNUM            *bnT;
707     BIGNUM            *bnE;
708     BIGNUM            *bnN;
709     BN_CTX            *context;
710
711
712     // Make sure that the required pointers are provided
713     pAssert(n != NULL && p != NULL);
714
715     // If the seed is provided, then use KDFa for generation of the 'random'
716     // values
717     ktxPtr = KDFaContextStart(&ktx, seed, hashAlg, extra, pCtr, keySizeInBits);
718
719     n->size = keySizeInBits/8;
720     p->size = n->size / 2;
721
722     // Validate exponent
723     if(e == 0 || e == RSA_DEFAULT_PUBLIC_EXPONENT)
724         e = RSA_DEFAULT_PUBLIC_EXPONENT;
725     else
726         if(!IsPrimeWord(e))
727             return CRYPT_FAIL;
728
729     // Get structures for the big number representations
730     context = BN_CTX_new();
731     BN_CTX_start(context);
732     bnP = BN_CTX_get(context);
733     bnQ = BN_CTX_get(context);
734     bnT = BN_CTX_get(context);

```

```

735     bnE = BN_CTX_get(context);
736     bnN = BN_CTX_get(context);
737     if(bnN == NULL)
738         FAIL(FATAL_ERROR_INTERNAL);
739
740     // Set Q to zero. This is used as a flag. The prime is computed in P. When a
741     // new prime is found, Q is checked to see if it is zero. If so, P is copied
742     // to Q and a new P is found. When both P and Q are non-zero, the modulus and
743     // private exponent are computed and a trial encryption/decryption is
744     // performed. If the encrypt/decrypt fails, assume that at least one of the
745     // primes is composite. Since we don't know which one, set Q to zero and start
746     // over and find a new pair of primes.
747     BN_zero(bnQ);
748     BN_set_word(bnE, e);
749
750     // Each call to generate a random value will increment ktx.outer
751     // it doesn't matter if ktx.outer wraps. This lets the caller
752     // use the initial value of the counter for additional entropy.
753     for(i = 0; i < UINT32_MAX; i++)
754     {
755         if(_plat_IsCanceled())
756         {
757             retVal = CRYPT_CANCEL;
758             goto end;
759         }
760         // Get a random prime candidate.
761         if(seed == NULL)
762             _cpri_GenerateRandom(p->size, p->buffer);
763         else
764             RandomForRsa(&ktx, label, p);
765         AdjustPrimeCandidate(p->buffer, p->size);
766
767         // Convert the candidate to a BN
768         if(BN_bin2bn(p->buffer, p->size, bnP) == NULL)
769             FAIL(FATAL_ERROR_INTERNAL);
770         // If this is the second prime, make sure that it differs from the
771         // first prime by at least 2^100. Since BIGNUMS use words, the check
772         // below will make sure they are different by at least 128 bits
773         if(!BN_is_zero(bnQ))
774         { // bnQ is non-zero, we have a first value
775             UINT32 *pP = (UINT32 *)(&bnP->d[4]);
776             UINT32 *pQ = (UINT32 *)(&bnQ->d[4]);
777             INT32 k = ((INT32)bnP->top) - 4;
778             for(;k > 0; k--)
779                 if(*pP++ != *pQ++)
780                     break;
781             // Didn't find any difference so go get a new value
782             if(k == 0)
783                 continue;
784         }
785         // If PrimeSelectWithSieve returns success, bnP is a prime,
786 #ifdef RSA_DEBUG
787         if(!PrimeSelectWithSieve(bnP, ktxPtr, e, context, fieldSize, primes))
788 #else
789         if(!PrimeSelectWithSieve(bnP, ktxPtr, e, context))
790 #endif
791             continue; // If not, get another
792
793         // Found a prime, is this the first or second.
794         if(BN_is_zero(bnQ))
795         { // copy p to q and compute another prime in p
796             BN_copy(bnQ, bnP);
797             continue;
798         }
799         //Form the public modulus
800         if( BN_mul(bnN, bnP, bnQ, context) != 1

```

```

801     || BN_num_bits(bnN) != keySizeInBits)
802     FAIL(FATAL_ERROR_INTERNAL);
803     // Save the public modulus
804     BnTo2B(n, bnN, n->size);
805     // And one prime
806     BnTo2B(p, bnP, p->size);
807
808 #ifdef EXTENDED_CHECKS
809     // Finish by making sure that we can form the modular inverse of PHI
810     // with respect to the public exponent
811     // Compute PHI = (p - 1)(q - 1) = n - p - q + 1
812     // Make sure that we can form the modular inverse
813     if( BN_sub(bnT, bnN, bnP) != 1
814        || BN_sub(bnT, bnT, bnQ) != 1
815        || BN_add_word(bnT, 1) != 1)
816         FAIL(FATAL_ERROR_INTERNAL);
817
818     // find d such that (Phi * d) mod e ==1
819     // If there isn't then we are broken because we took the step
820     // of making sure that the prime != 1 mod e so the modular inverse
821     // must exist
822     if( BN_mod_inverse(bnT, bnE, bnT, context) == NULL
823        || BN_is_zero(bnT))
824         FAIL(FATAL_ERROR_INTERNAL);
825
826     // And, finally, do a trial encryption decryption
827     {
828         TPM2B_TYPE(RSA_KEY, MAX_RSA_KEY_BYTES);
829         TPM2B_RSA_KEY r;
830         r.t.size = sizeof(r.t.buffer);
831         // If we are using a seed, then results must be reproducible on each
832         // call. Otherwise, just get a random number
833         if(seed == NULL)
834             _cpri_GenerateRandom(keySizeInBits/8, r.t.buffer);
835         else
836             RandomForRsa(&ktx, label, &r.b);
837
838         // Make sure that the number is smaller than the public modulus
839         r.t.buffer[0] &= 0x7F;
840         // Convert
841         if( BN_bin2bn(r.t.buffer, r.t.size, bnP) == NULL
842            // Encrypt with the public exponent
843            || BN_mod_exp(bnQ, bnP, bnE, bnN, context) != 1
844            // Decrypt with the private exponent
845            || BN_mod_exp(bnQ, bnQ, bnT, bnN, context) != 1)
846             FAIL(FATAL_ERROR_INTERNAL);
847         // If the starting and ending values are not the same, start over -);
848         if(BN_ucmp(bnP, bnQ) != 0)
849             {
850                 BN_zero(bnQ);
851                 continue;
852             }
853     }
854 #endif // EXTENDED_CHECKS
855     retVal = CRYPT_SUCCESS;
856     goto end;
857 }
858 retVal = CRYPT_FAIL;
859
860 end:
861     KDFaContextEnd(&ktx);
862
863     // Free up allocated BN values
864     BN_CTX_end(context);
865     BN_CTX_free(context);
866     return retVal;

```



```

867 }
868 #else
869 static void noFuntion(void)
870 {
871     pAssert(1);
872 }
873 #endif          //%
```

#### B.9.2.4. RSADData.c

```

1  #include "CryptoEngine.h"
2  #ifdef  RSA_KEY_SIEVE
3  #ifdef  RSA_DEBUG
4  UINT16 defaultFieldSize = MAX_FIELD_SIZE;
5  #endif
```

This table contains a pre-sieved table. It has the bits for 3, 5, and 7 removed. Because of the factors, it needs to be aligned to 105 and has a repeat of 105.

```

6  const BYTE  seedValues[SEED_VALUES_SIZE] = {
7      0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c,
8      0x99, 0x96, 0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96,
9      0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
10     0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
11     0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
12     0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
13     0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
14     0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0x69, 0xc9, 0x34,
15     0x25, 0xda, 0x22, 0x65, 0x99, 0xb4, 0x4c, 0x1b,
16     0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
17     0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6,
18     0x25, 0xd3, 0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2,
19     0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
20     0xd1};
21  const BYTE bitsInByte[256] = {
22     0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
23     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
24     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
25     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
26     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
27     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
28     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
29     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
30     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
31     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
32     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
33     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
34     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
35     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
36     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
37     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
38     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
39     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
40     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
41     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
42     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
43     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
44     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
45     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
46     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
47     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
48     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
49     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
50     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
```

```

51     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
52     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
53     0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08
54 };

```

Following table contains a byte that is the difference between two successive primes. This reduces the table size by a factor of two. It is optimized for sequential access to the prime table which is the most common case.

When the table size is at its max, the table will have all primes less than  $2^{16}$ . This is 6542 primes in 6542 bytes.

```

55 const UINT16     primeTableBytes = PRIME_DIFF_TABLE_BYTES;
56 #if PRIME_DIFF_TABLE_BYTES > 0
57 const BYTE primeDiffTable [PRIME_DIFF_TABLE_BYTES] = {
58     0x02,0x02,0x02,0x04,0x02,0x04,0x02,0x04,0x06,0x02,0x06,0x04,0x02,0x04,0x06,0x06,
59     0x02,0x06,0x04,0x02,0x06,0x04,0x06,0x08,0x04,0x02,0x04,0x02,0x04,0x0E,0x04,0x06,
60     0x02,0x0A,0x02,0x06,0x06,0x04,0x06,0x06,0x02,0x0A,0x02,0x04,0x02,0x0C,0x0C,0x04,
61     0x02,0x04,0x06,0x02,0x0A,0x06,0x06,0x06,0x02,0x06,0x04,0x02,0x0A,0x0E,0x04,0x02,
62     0x04,0x0E,0x06,0x0A,0x02,0x04,0x06,0x08,0x06,0x06,0x06,0x04,0x06,0x08,0x04,0x08,0x0A,
63     0x02,0x0A,0x02,0x06,0x04,0x06,0x08,0x04,0x02,0x04,0x0C,0x08,0x04,0x08,0x04,0x08,0x0A,
64     0x0C,0x02,0x12,0x06,0x0A,0x06,0x06,0x02,0x06,0x0A,0x06,0x06,0x02,0x06,0x06,0x06,0x04,
65     0x02,0x0C,0x0A,0x02,0x04,0x06,0x06,0x02,0x0C,0x04,0x06,0x08,0x0A,0x08,0x0A,0x08,
66     0x06,0x06,0x04,0x08,0x06,0x04,0x08,0x04,0x0E,0x0A,0x0C,0x02,0x0A,0x02,0x04,0x02,
67     0x0A,0x0E,0x04,0x02,0x04,0x0E,0x04,0x02,0x04,0x14,0x04,0x08,0x0A,0x08,0x04,0x06,
68     0x06,0x0E,0x04,0x06,0x06,0x08,0x06,0x0C,0x04,0x06,0x02,0x0A,0x02,0x06,0x0A,0x02,
69     0x0A,0x02,0x06,0x12,0x04,0x02,0x04,0x06,0x06,0x06,0x08,0x06,0x06,0x16,0x02,0x0A,0x08,
70     0x0A,0x06,0x06,0x08,0x0C,0x04,0x06,0x06,0x06,0x02,0x0C,0x0A,0x12,0x02,0x04,0x06,
71     0x02,0x06,0x04,0x02,0x04,0x0C,0x02,0x06,0x22,0x06,0x06,0x08,0x12,0x0A,0x0E,0x04,
72     0x02,0x04,0x06,0x08,0x04,0x02,0x06,0x0C,0x0A,0x02,0x04,0x02,0x04,0x06,0x0C,0x0C,
73     0x08,0x0C,0x06,0x04,0x06,0x08,0x04,0x08,0x04,0x0E,0x04,0x06,0x02,0x04,0x06,0x02
74 #endif
75 // 256
76 #if PRIME_DIFF_TABLE_BYTES > 256
77     ,0x06,0x0A,0x14,0x06,0x04,0x02,0x18,0x04,0x02,0x0A,0x0C,0x02,0x0A,0x08,0x06,0x06,
78     0x06,0x12,0x06,0x04,0x02,0x0C,0x0A,0x0C,0x08,0x10,0x0E,0x06,0x04,0x02,0x04,0x02,
79     0x0A,0x0C,0x06,0x06,0x12,0x02,0x10,0x02,0x16,0x06,0x08,0x06,0x04,0x02,0x04,0x08,
80     0x06,0x0A,0x02,0x0A,0x0E,0x0A,0x06,0x0C,0x02,0x04,0x02,0x0A,0x0C,0x02,0x10,0x02,
81     0x06,0x04,0x02,0x0A,0x08,0x12,0x18,0x04,0x06,0x08,0x10,0x02,0x04,0x08,0x10,0x02,
82     0x04,0x08,0x06,0x06,0x04,0x0C,0x02,0x16,0x06,0x02,0x06,0x04,0x06,0x0E,0x06,0x04,
83     0x02,0x06,0x04,0x06,0x06,0x06,0x06,0x0E,0x04,0x06,0x0C,0x08,0x06,0x04,0x1A,0x12,
84     0x0A,0x08,0x04,0x06,0x02,0x06,0x16,0x0C,0x02,0x10,0x08,0x04,0x0C,0x0E,0x0A,0x02,
85     0x04,0x08,0x06,0x06,0x04,0x02,0x04,0x06,0x08,0x04,0x02,0x06,0x0A,0x02,0x0A,0x08,
86     0x04,0x0E,0x0A,0x0C,0x02,0x06,0x04,0x02,0x10,0x0E,0x04,0x06,0x08,0x06,0x04,0x12,
87     0x08,0x0A,0x06,0x06,0x08,0x0A,0x0C,0x0E,0x04,0x06,0x06,0x02,0x1C,0x02,0x0A,0x08,
88     0x04,0x0E,0x04,0x08,0x0C,0x06,0x0C,0x04,0x06,0x14,0x0A,0x02,0x10,0x1A,0x04,0x02,
89     0x0C,0x06,0x04,0x0C,0x06,0x08,0x04,0x08,0x16,0x02,0x04,0x02,0x0C,0x1C,0x02,0x06,
90     0x16,0x06,0x04,0x06,0x02,0x0C,0x04,0x0C,0x02,0x0A,0x02,0x10,0x02,0x10,0x06,0x14,
91     0x10,0x08,0x04,0x02,0x04,0x02,0x16,0x08,0x0C,0x06,0x0A,0x02,0x04,0x06,0x02,0x06,
92     0x0A,0x02,0x0C,0x0A,0x02,0x0A,0x0E,0x06,0x04,0x06,0x08,0x06,0x06,0x10,0x0C,0x02
93 #endif
94 // 512
95 #if PRIME_DIFF_TABLE_BYTES > 512
96     ,0x04,0x0E,0x06,0x04,0x08,0x0A,0x08,0x06,0x06,0x16,0x06,0x02,0x0A,0x0E,0x04,0x06,
97     0x12,0x02,0x0A,0x0E,0x04,0x02,0x0A,0x0E,0x04,0x08,0x12,0x04,0x06,0x02,0x04,0x06,
98     0x02,0x0C,0x04,0x14,0x16,0x0C,0x02,0x04,0x06,0x06,0x02,0x06,0x16,0x02,0x06,0x10,
99     0x06,0x0C,0x02,0x06,0x0C,0x10,0x02,0x04,0x06,0x0E,0x04,0x02,0x12,0x18,0x0A,0x06,
100    0x02,0x0A,0x02,0x0A,0x02,0x0A,0x06,0x02,0x0A,0x02,0x0A,0x06,0x08,0x1E,0x0A,0x02,
101    0x0A,0x08,0x06,0x0A,0x12,0x06,0x0C,0x0C,0x02,0x12,0x06,0x04,0x06,0x06,0x12,0x02,
102    0x0A,0x0E,0x06,0x04,0x02,0x04,0x18,0x02,0x0C,0x06,0x10,0x08,0x06,0x06,0x12,0x10,
103    0x02,0x04,0x06,0x02,0x06,0x06,0x0A,0x06,0x0C,0x0C,0x12,0x02,0x06,0x04,0x12,0x08,
104    0x18,0x04,0x02,0x04,0x06,0x02,0x0C,0x04,0x0E,0x1E,0x0A,0x06,0x0C,0x0E,0x06,0x0A,
105    0x0C,0x02,0x04,0x06,0x08,0x06,0x0A,0x02,0x04,0x0E,0x06,0x06,0x04,0x06,0x02,0x0A,
106    0x02,0x10,0x0C,0x08,0x12,0x04,0x06,0x0C,0x02,0x06,0x06,0x06,0x1C,0x06,0x0E,0x04,
107    0x08,0x0A,0x08,0x0C,0x12,0x04,0x02,0x04,0x18,0x0C,0x06,0x02,0x10,0x06,0x06,0x0E,

```

```

108     0x0A,0x0E,0x04,0x1E,0x06,0x06,0x06,0x08,0x06,0x04,0x02,0x0C,0x06,0x04,0x02,0x06,
109     0x16,0x06,0x02,0x04,0x12,0x02,0x04,0x0C,0x02,0x06,0x04,0x1A,0x06,0x06,0x04,0x08,
110     0x0A,0x20,0x10,0x02,0x06,0x04,0x02,0x04,0x02,0x0A,0x0E,0x06,0x04,0x08,0x0A,0x06,
111     0x14,0x04,0x02,0x06,0x1E,0x04,0x08,0x0A,0x06,0x08,0x06,0x0C,0x04,0x06,0x02
112 #endif
113 // 768
114 #if PRIME_DIFF_TABLE_BYTES > 768
115     ,0x06,0x04,0x06,0x02,0x0A,0x02,0x10,0x06,0x14,0x04,0x0C,0x0E,0x1C,0x06,0x14,0x04,
116     0x12,0x08,0x06,0x04,0x06,0x0E,0x06,0x06,0x0A,0x02,0x0A,0x0C,0x08,0x0A,0x02,0x0A,
117     0x08,0x0C,0x0A,0x18,0x02,0x04,0x08,0x06,0x04,0x08,0x12,0x0A,0x06,0x06,0x02,0x06,
118     0x0A,0x0C,0x02,0x0A,0x06,0x06,0x06,0x08,0x06,0x0A,0x06,0x02,0x06,0x06,0x06,0x0A,
119     0x08,0x18,0x06,0x16,0x02,0x12,0x04,0x08,0x0A,0x1E,0x08,0x12,0x04,0x02,0x0A,0x06,
120     0x02,0x06,0x04,0x12,0x08,0x0C,0x12,0x10,0x06,0x02,0x0C,0x06,0x0A,0x02,0x0A,0x02,
121     0x06,0x0A,0x0E,0x04,0x18,0x02,0x10,0x02,0x0A,0x02,0x0A,0x14,0x04,0x02,0x04,0x08,
122     0x10,0x06,0x06,0x02,0x0C,0x10,0x08,0x04,0x06,0x1E,0x02,0x0A,0x02,0x06,0x04,0x06,
123     0x06,0x08,0x06,0x04,0x0C,0x06,0x08,0x0C,0x04,0x0E,0x0C,0x0A,0x18,0x06,0x0C,0x06,
124     0x02,0x16,0x08,0x12,0x0A,0x06,0x0E,0x04,0x02,0x06,0x0A,0x08,0x06,0x04,0x06,0x1E,
125     0x0E,0x0A,0x02,0x0C,0x0A,0x02,0x10,0x02,0x12,0x18,0x12,0x06,0x10,0x12,0x06,0x02,
126     0x12,0x04,0x06,0x02,0x0A,0x08,0x0A,0x06,0x06,0x08,0x04,0x06,0x02,0x0A,0x02,0x0C,
127     0x04,0x06,0x06,0x02,0x0C,0x04,0x0E,0x12,0x04,0x06,0x14,0x04,0x08,0x06,0x04,0x08,
128     0x04,0x0E,0x06,0x04,0x0E,0x0C,0x04,0x02,0x1E,0x04,0x18,0x06,0x06,0x0C,0x0C,0x0E,
129     0x06,0x04,0x02,0x04,0x12,0x06,0x0C,0x08,0x06,0x04,0x0C,0x02,0x0C,0x1E,0x10,0x02,
130     0x06,0x16,0x0E,0x06,0x0A,0x0C,0x06,0x02,0x04,0x08,0x0A,0x06,0x06,0x18,0x0E,0x06
131 #endif
132 // 1024
133 #if PRIME_DIFF_TABLE_BYTES > 1024
134     ,0x04,0x08,0x0C,0x12,0x0A,0x02,0x0A,0x02,0x04,0x06,0x14,0x06,0x04,0x0E,0x04,0x02,
135     0x04,0x0E,0x06,0x0C,0x18,0x0A,0x06,0x08,0x0A,0x02,0x1E,0x04,0x06,0x02,0x0C,0x04,
136     0x0E,0x06,0x22,0x0C,0x08,0x06,0x0A,0x02,0x04,0x14,0x0A,0x08,0x10,0x02,0x0A,0x0E,
137     0x04,0x02,0x0C,0x06,0x10,0x06,0x08,0x04,0x08,0x04,0x06,0x08,0x10,0x06,0x0C,0x06,
138     0x04,0x06,0x06,0x08,0x12,0x04,0x14,0x04,0x0C,0x02,0x0A,0x06,0x02,0x0A,0x0C,0x02,
139     0x04,0x14,0x06,0x1E,0x06,0x04,0x08,0x0A,0x0C,0x06,0x02,0x1C,0x02,0x06,0x04,0x02,
140     0x10,0x0C,0x02,0x06,0x0A,0x08,0x18,0x0C,0x06,0x12,0x06,0x04,0x0E,0x06,0x04,0x0C,
141     0x08,0x06,0x0C,0x04,0x06,0x0C,0x06,0x0C,0x02,0x10,0x14,0x04,0x02,0x0A,0x12,0x08,
142     0x04,0x0E,0x04,0x02,0x06,0x16,0x06,0x0E,0x06,0x06,0x0A,0x06,0x02,0x0A,0x02,0x04,
143     0x02,0x16,0x02,0x04,0x06,0x06,0x0C,0x06,0x0C,0x0E,0x0A,0x0C,0x06,0x08,0x04,0x24,0x0E,
144     0x0C,0x06,0x04,0x06,0x10,0x0C,0x06,0x0C,0x10,0x02,0x0A,0x08,0x16,0x02,0x0C,0x06,
145     0x04,0x06,0x12,0x02,0x0C,0x06,0x04,0x0C,0x08,0x06,0x0C,0x04,0x06,0x0C,0x06,0x02,
146     0x0C,0x0C,0x04,0x0E,0x06,0x10,0x06,0x02,0x0A,0x08,0x12,0x06,0x22,0x02,0x1C,0x02,
147     0x16,0x06,0x02,0x0A,0x0C,0x02,0x06,0x04,0x08,0x16,0x06,0x02,0x0A,0x08,0x04,0x06,
148     0x08,0x04,0x0C,0x12,0x0C,0x14,0x04,0x06,0x06,0x08,0x04,0x02,0x10,0x0C,0x02,0x0A,
149     0x08,0x0A,0x02,0x04,0x06,0x0E,0x0C,0x16,0x08,0x1C,0x02,0x04,0x14,0x04,0x02,0x04
150 #endif
151 // 1280
152 #if PRIME_DIFF_TABLE_BYTES > 1280
153     ,0x0E,0x0A,0x0C,0x02,0x0C,0x10,0x02,0x1C,0x08,0x16,0x08,0x04,0x06,0x06,0x0E,0x04,
154     0x08,0x0C,0x06,0x06,0x04,0x14,0x04,0x12,0x02,0x0C,0x06,0x04,0x06,0x0E,0x12,0x0A,
155     0x08,0x0A,0x20,0x06,0x0A,0x06,0x06,0x02,0x06,0x10,0x06,0x02,0x0C,0x06,0x1C,0x02,
156     0x0A,0x08,0x10,0x06,0x08,0x06,0x0A,0x18,0x14,0x0A,0x02,0x0A,0x02,0x0C,0x04,0x06,
157     0x14,0x04,0x02,0x0C,0x12,0x0A,0x02,0x0A,0x02,0x04,0x14,0x1A,0x10,0x04,0x08,0x06,
158     0x04,0x0C,0x06,0x08,0x0C,0x0C,0x06,0x04,0x08,0x16,0x02,0x10,0x0E,0x0A,0x06,0x0C,
159     0x0C,0x0E,0x06,0x04,0x14,0x04,0x0C,0x06,0x02,0x06,0x06,0x10,0x08,0x16,0x02,0x1C,
160     0x08,0x06,0x04,0x14,0x04,0x0C,0x18,0x14,0x04,0x08,0x0A,0x02,0x10,0x02,0x0C,0x0C,
161     0x22,0x02,0x04,0x06,0x0C,0x06,0x06,0x08,0x06,0x04,0x02,0x06,0x18,0x04,0x14,0x0A,
162     0x06,0x06,0x0E,0x04,0x06,0x06,0x02,0x0C,0x06,0x0A,0x02,0x0A,0x06,0x14,0x04,0x1A,
163     0x04,0x02,0x06,0x16,0x02,0x18,0x04,0x06,0x02,0x04,0x06,0x18,0x06,0x08,0x04,0x02,
164     0x22,0x06,0x08,0x10,0x0C,0x02,0x0A,0x02,0x0A,0x06,0x08,0x04,0x08,0x0C,0x16,0x06,
165     0x0E,0x04,0x1A,0x04,0x02,0x0C,0x0A,0x08,0x04,0x08,0x0C,0x04,0x0E,0x06,0x10,0x06,
166     0x08,0x04,0x06,0x06,0x08,0x06,0x0A,0x0C,0x02,0x06,0x06,0x10,0x08,0x06,0x06,0x0C,
167     0x0A,0x02,0x06,0x12,0x04,0x06,0x06,0x06,0x0C,0x12,0x08,0x06,0x0A,0x08,0x12,0x04,
168     0x0E,0x06,0x12,0x0A,0x08,0x0A,0x0C,0x02,0x06,0x0C,0x0C,0x24,0x04,0x06,0x08,0x04
169 #endif
170 // 1536
171 #if PRIME_DIFF_TABLE_BYTES > 1536
172     ,0x06,0x02,0x04,0x12,0x0C,0x06,0x08,0x06,0x06,0x04,0x12,0x02,0x04,0x02,0x18,0x04,
173     0x06,0x06,0x0E,0x1E,0x06,0x04,0x06,0x0C,0x06,0x14,0x04,0x08,0x04,0x08,0x06,0x06,

```

```

174     0x04, 0x1E, 0x02, 0x0A, 0x0C, 0x08, 0x0A, 0x08, 0x18, 0x06, 0x0C, 0x04, 0x0E, 0x04, 0x06, 0x02,
175     0x1C, 0x0E, 0x10, 0x02, 0x0C, 0x06, 0x04, 0x14, 0x0A, 0x06, 0x06, 0x06, 0x08, 0x0A, 0x0C, 0x0E,
176     0x0A, 0x0E, 0x10, 0x0E, 0x0A, 0x0E, 0x06, 0x10, 0x06, 0x08, 0x06, 0x10, 0x14, 0x0A, 0x02, 0x06,
177     0x04, 0x02, 0x04, 0x0C, 0x02, 0x0A, 0x02, 0x06, 0x16, 0x06, 0x02, 0x04, 0x12, 0x08, 0x0A, 0x08,
178     0x16, 0x02, 0x0A, 0x12, 0x0E, 0x04, 0x02, 0x04, 0x12, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x02, 0x1E,
179     0x04, 0x1E, 0x02, 0x0A, 0x02, 0x12, 0x04, 0x12, 0x06, 0x0E, 0x0A, 0x02, 0x04, 0x14, 0x24, 0x06,
180     0x04, 0x06, 0x0E, 0x04, 0x14, 0x0A, 0x0E, 0x16, 0x06, 0x02, 0x1E, 0x0C, 0x0A, 0x12, 0x02, 0x04,
181     0x0E, 0x06, 0x16, 0x12, 0x02, 0x0C, 0x06, 0x04, 0x08, 0x04, 0x08, 0x06, 0x0A, 0x02, 0x0C, 0x12,
182     0x0A, 0x0E, 0x10, 0x0E, 0x04, 0x06, 0x06, 0x02, 0x06, 0x04, 0x02, 0x1C, 0x02, 0x1C, 0x06, 0x02,
183     0x04, 0x06, 0x0E, 0x04, 0x0C, 0x0E, 0x10, 0x0E, 0x04, 0x06, 0x08, 0x06, 0x04, 0x06, 0x06, 0x06,
184     0x08, 0x04, 0x08, 0x04, 0x0E, 0x10, 0x08, 0x06, 0x04, 0x0C, 0x08, 0x10, 0x02, 0x0A, 0x08, 0x04,
185     0x06, 0x1A, 0x06, 0x0A, 0x08, 0x04, 0x06, 0x0C, 0x0E, 0x1E, 0x04, 0x0E, 0x16, 0x08, 0x0C, 0x04,
186     0x06, 0x08, 0x0A, 0x06, 0x0E, 0x0A, 0x06, 0x02, 0x0A, 0x0C, 0x0C, 0x0E, 0x06, 0x06, 0x12, 0x0A,
187     0x06, 0x08, 0x12, 0x04, 0x06, 0x02, 0x06, 0x0A, 0x02, 0x0A, 0x08, 0x06, 0x06, 0x0A, 0x02, 0x12
188 #endif
189 // 1792
190 #if PRIME_DIFF_TABLE_BYTES > 1792
191     , 0x0A, 0x02, 0x0C, 0x04, 0x06, 0x08, 0x0A, 0x0C, 0x0E, 0x0C, 0x04, 0x08, 0x0A, 0x06, 0x06, 0x14,
192     0x04, 0x0E, 0x10, 0x0E, 0x0A, 0x08, 0x0A, 0x0C, 0x02, 0x12, 0x06, 0x0C, 0x0A, 0x0C, 0x02, 0x04,
193     0x02, 0x0C, 0x06, 0x04, 0x08, 0x04, 0x2C, 0x04, 0x02, 0x04, 0x02, 0x0A, 0x0C, 0x06, 0x06, 0x0E,
194     0x04, 0x06, 0x06, 0x06, 0x08, 0x06, 0x24, 0x12, 0x04, 0x06, 0x02, 0x0C, 0x06, 0x06, 0x06, 0x04,
195     0x0E, 0x16, 0x0C, 0x02, 0x12, 0x0A, 0x06, 0x1A, 0x18, 0x04, 0x02, 0x04, 0x02, 0x04, 0x0E, 0x04,
196     0x06, 0x06, 0x08, 0x10, 0x0C, 0x02, 0x2A, 0x04, 0x02, 0x04, 0x18, 0x06, 0x06, 0x02, 0x12, 0x04,
197     0x0E, 0x06, 0x1C, 0x12, 0x0E, 0x06, 0x0A, 0x0C, 0x02, 0x06, 0x0C, 0x1E, 0x06, 0x04, 0x06, 0x06,
198     0x0E, 0x04, 0x02, 0x18, 0x04, 0x06, 0x06, 0x1A, 0x0A, 0x12, 0x06, 0x08, 0x06, 0x06, 0x1E, 0x04,
199     0x0C, 0x0C, 0x02, 0x10, 0x02, 0x06, 0x04, 0x0C, 0x12, 0x02, 0x06, 0x04, 0x1A, 0x0C, 0x06, 0x0C,
200     0x04, 0x18, 0x18, 0x0C, 0x06, 0x02, 0x0C, 0x1C, 0x08, 0x04, 0x06, 0x0C, 0x02, 0x12, 0x06, 0x04,
201     0x06, 0x06, 0x14, 0x10, 0x02, 0x06, 0x06, 0x12, 0x0A, 0x06, 0x02, 0x04, 0x08, 0x06, 0x06, 0x18,
202     0x10, 0x06, 0x08, 0x0A, 0x06, 0x0E, 0x16, 0x08, 0x10, 0x06, 0x02, 0x0C, 0x04, 0x02, 0x16, 0x08,
203     0x12, 0x22, 0x02, 0x06, 0x12, 0x04, 0x06, 0x06, 0x08, 0x0A, 0x08, 0x12, 0x06, 0x04, 0x02, 0x04,
204     0x08, 0x10, 0x02, 0x0C, 0x0C, 0x06, 0x12, 0x04, 0x06, 0x06, 0x06, 0x02, 0x06, 0x0C, 0x0A, 0x14,
205     0x0C, 0x12, 0x04, 0x06, 0x02, 0x10, 0x02, 0x0A, 0x0E, 0x04, 0x1E, 0x02, 0x0A, 0x0C, 0x02, 0x18,
206     0x06, 0x10, 0x08, 0x0A, 0x02, 0x0C, 0x16, 0x06, 0x02, 0x10, 0x14, 0x0A, 0x02, 0x0C, 0x0C, 0x00
207 #endif
208 // 2048
209 #if PRIME_DIFF_TABLE_BYTES > 2048
210     , 0x12, 0x0A, 0x0C, 0x06, 0x02, 0x0A, 0x02, 0x06, 0x0A, 0x12, 0x02, 0x0C, 0x06, 0x04, 0x06, 0x02,
211     0x18, 0x1C, 0x02, 0x04, 0x02, 0x0A, 0x02, 0x10, 0x0C, 0x08, 0x16, 0x02, 0x06, 0x04, 0x02, 0x0A,
212     0x06, 0x14, 0x0C, 0x0A, 0x08, 0x0C, 0x06, 0x06, 0x06, 0x04, 0x12, 0x02, 0x04, 0x0C, 0x12, 0x02,
213     0x0C, 0x06, 0x04, 0x02, 0x10, 0x0C, 0x0C, 0x0E, 0x04, 0x08, 0x12, 0x04, 0x0C, 0x0E, 0x06, 0x06,
214     0x04, 0x08, 0x06, 0x04, 0x14, 0x0C, 0x0A, 0x0E, 0x04, 0x02, 0x10, 0x02, 0x0C, 0x1E, 0x04, 0x06,
215     0x18, 0x14, 0x18, 0x0A, 0x08, 0x0C, 0x0A, 0x0C, 0x06, 0x0C, 0x0C, 0x06, 0x08, 0x10, 0x0E, 0x06,
216     0x04, 0x06, 0x24, 0x14, 0x0A, 0x1E, 0x0C, 0x02, 0x04, 0x02, 0x1C, 0x0C, 0x0E, 0x06, 0x16, 0x08,
217     0x04, 0x12, 0x06, 0x0E, 0x12, 0x04, 0x06, 0x02, 0x06, 0x22, 0x12, 0x02, 0x10, 0x06, 0x12, 0x02,
218     0x18, 0x04, 0x02, 0x06, 0x0C, 0x06, 0x0C, 0x0A, 0x08, 0x06, 0x10, 0x0C, 0x08, 0x0A, 0x0E, 0x28,
219     0x06, 0x02, 0x06, 0x04, 0x0C, 0x0E, 0x04, 0x02, 0x04, 0x02, 0x04, 0x08, 0x06, 0x0A, 0x06, 0x06,
220     0x02, 0x06, 0x06, 0x06, 0x0C, 0x06, 0x18, 0x0A, 0x02, 0x0A, 0x06, 0x0C, 0x06, 0x06, 0x0E, 0x06,
221     0x06, 0x34, 0x14, 0x06, 0x0A, 0x02, 0x0A, 0x08, 0x0A, 0x0C, 0x02, 0x06, 0x04, 0x0E, 0x10,
222     0x08, 0x0C, 0x06, 0x16, 0x02, 0x0A, 0x08, 0x06, 0x16, 0x02, 0x16, 0x06, 0x08, 0x0A, 0x0C, 0x0C,
223     0x02, 0x0A, 0x06, 0x0C, 0x02, 0x04, 0x0E, 0x0A, 0x02, 0x06, 0x12, 0x04, 0x0C, 0x08, 0x12, 0x0C,
224     0x06, 0x06, 0x04, 0x06, 0x06, 0x0E, 0x04, 0x02, 0x0C, 0x0C, 0x04, 0x06, 0x12, 0x12, 0x0C, 0x02,
225     0x10, 0x0C, 0x08, 0x12, 0x0A, 0x1A, 0x04, 0x06, 0x08, 0x06, 0x06, 0x04, 0x02, 0x0A, 0x14, 0x04
226 #endif
227 // 2304
228 #if PRIME_DIFF_TABLE_BYTES > 2304
229     , 0x06, 0x08, 0x04, 0x14, 0x0A, 0x02, 0x22, 0x02, 0x04, 0x18, 0x02, 0x0C, 0x0C, 0x0A, 0x06, 0x02,
230     0x0C, 0x1E, 0x06, 0x0C, 0x10, 0x0C, 0x02, 0x16, 0x12, 0x0C, 0x0E, 0x0A, 0x02, 0x0C, 0x0C, 0x04,
231     0x02, 0x04, 0x06, 0x0C, 0x02, 0x10, 0x12, 0x02, 0x28, 0x08, 0x10, 0x06, 0x08, 0x0A, 0x02, 0x04,
232     0x12, 0x08, 0x0A, 0x08, 0x0C, 0x04, 0x12, 0x02, 0x12, 0x0A, 0x02, 0x04, 0x02, 0x04, 0x08, 0x1C,
233     0x02, 0x06, 0x16, 0x0C, 0x06, 0x0E, 0x12, 0x04, 0x06, 0x08, 0x06, 0x06, 0x0A, 0x08, 0x04, 0x02,
234     0x12, 0x0A, 0x06, 0x14, 0x16, 0x08, 0x06, 0x1E, 0x04, 0x02, 0x04, 0x12, 0x06, 0x1E, 0x02, 0x04,
235     0x08, 0x06, 0x04, 0x06, 0x0C, 0x0E, 0x22, 0x0E, 0x06, 0x04, 0x02, 0x06, 0x04, 0x02, 0x04, 0x0E, 0x04, 0x02,
236     0x06, 0x1C, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x02, 0x0A, 0x02, 0x0A, 0x02, 0x0A, 0x02, 0x1E, 0x02, 0x0C,
237     0x0C, 0x0A, 0x12, 0x0C, 0x0E, 0x0A, 0x02, 0x0C, 0x06, 0x0A, 0x06, 0x0E, 0x0C, 0x04, 0x0E, 0x04,
238     0x12, 0x02, 0x0A, 0x08, 0x04, 0x08, 0x0A, 0x0C, 0x12, 0x12, 0x08, 0x06, 0x12, 0x10, 0x0E, 0x06,
239     0x06, 0x0A, 0x0E, 0x04, 0x06, 0x02, 0x0C, 0x0C, 0x04, 0x06, 0x06, 0x0C, 0x02, 0x10, 0x02, 0x0C,

```

```
240     0x06,0x04,0x0E,0x06,0x04,0x02,0x0C,0x12,0x04,0x24,0x12,0x0C,0x0C,0x02,0x04,0x02,
241     0x04,0x08,0x0C,0x04,0x24,0x06,0x12,0x02,0x0C,0x0A,0x06,0x0C,0x18,0x08,0x06,0x06,
242     0x10,0x0C,0x02,0x12,0x0A,0x14,0x0A,0x02,0x06,0x12,0x04,0x02,0x28,0x06,0x02,0x10,
243     0x02,0x04,0x08,0x12,0x0A,0x0C,0x06,0x02,0x0A,0x08,0x04,0x06,0x0C,0x02,0x0A,0x12,
244     0x08,0x06,0x04,0x14,0x04,0x06,0x24,0x06,0x02,0x0A,0x06,0x18,0x06,0x0E,0x10,0x06
245 #endif
246 // 2560
247 #if PRIME_DIFF_TABLE_BYTES > 2560
248     ,0x12,0x02,0x0A,0x14,0x0A,0x08,0x06,0x04,0x06,0x02,0x0A,0x02,0x0C,0x04,0x02,0x04,
249     0x08,0x0A,0x06,0x0C,0x12,0x0E,0x0C,0x10,0x08,0x06,0x10,0x08,0x04,0x02,0x06,0x12,
250     0x18,0x12,0x0A,0x0C,0x02,0x04,0x0E,0x0A,0x06,0x06,0x06,0x12,0x0C,0x02,0x1C,0x12,
251     0x0E,0x10,0x0C,0x0E,0x18,0x0C,0x16,0x06,0x02,0x0A,0x08,0x04,0x02,0x04,0x0E,0x0C,
252     0x06,0x04,0x06,0x0E,0x04,0x02,0x04,0x1E,0x06,0x02,0x06,0x0A,0x02,0x1E,0x16,0x02,
253     0x04,0x06,0x08,0x06,0x06,0x10,0x0C,0x0C,0x06,0x08,0x04,0x02,0x18,0x0C,0x04,0x06,
254     0x08,0x06,0x06,0x0A,0x02,0x06,0x0C,0x1C,0x0E,0x06,0x04,0x0C,0x08,0x06,0x0C,0x04,
255     0x06,0x0E,0x06,0x0C,0x0A,0x06,0x06,0x08,0x06,0x06,0x04,0x04,0x08,0x0C,0x0C,0x04,
256     0x0E,0x12,0x0A,0x02,0x10,0x06,0x14,0x06,0x0A,0x08,0x04,0x1E,0x24,0x0C,0x08,0x16,
257     0x0C,0x02,0x06,0x0C,0x10,0x06,0x06,0x02,0x12,0x04,0x1A,0x04,0x08,0x12,0x0A,0x08,
258     0x0A,0x06,0x0E,0x04,0x14,0x16,0x12,0x0C,0x08,0x1C,0x0C,0x06,0x06,0x08,0x06,0x0C,
259     0x18,0x10,0x0E,0x04,0x0E,0x0C,0x06,0x0A,0x0C,0x14,0x06,0x04,0x08,0x12,0x0C,0x12,
260     0x0A,0x02,0x04,0x14,0x0A,0x0E,0x04,0x06,0x02,0x0A,0x18,0x12,0x02,0x04,0x14,0x10,
261     0x0E,0x0A,0x0E,0x06,0x04,0x06,0x14,0x06,0x0A,0x06,0x02,0x0C,0x06,0x1E,0x0A,0x08,
262     0x06,0x04,0x06,0x08,0x28,0x02,0x04,0x02,0x0C,0x12,0x04,0x06,0x08,0x0A,0x06,0x12,
263     0x12,0x02,0x0C,0x10,0x08,0x06,0x04,0x06,0x02,0x34,0x0E,0x04,0x14,0x10,0x02
264 #endif
265 // 2816
266 #if PRIME_DIFF_TABLE_BYTES > 2816
267     ,0x04,0x06,0x0C,0x02,0x06,0x0C,0x0C,0x06,0x04,0x0E,0x0A,0x06,0x06,0x0E,0x0A,0x0E,
268     0x10,0x08,0x06,0x0C,0x04,0x08,0x16,0x06,0x02,0x12,0x16,0x06,0x02,0x12,0x06,0x10,
269     0x0E,0x0A,0x06,0x0C,0x02,0x06,0x04,0x08,0x12,0x0C,0x10,0x02,0x04,0x0E,0x04,0x08,
270     0x0C,0x0C,0x1E,0x10,0x08,0x04,0x02,0x06,0x16,0x0C,0x08,0x0A,0x06,0x06,0x06,0x0E,
271     0x06,0x12,0x0A,0x0C,0x02,0x0A,0x02,0x04,0x1A,0x04,0x0C,0x08,0x04,0x12,0x08,0x0A,
272     0x0E,0x10,0x06,0x06,0x08,0x0A,0x06,0x08,0x06,0x0C,0x0A,0x14,0x0A,0x08,0x04,0x0C,
273     0x1A,0x12,0x04,0x0C,0x12,0x06,0x1E,0x06,0x08,0x06,0x16,0x0C,0x02,0x04,0x06,0x06,
274     0x02,0x0A,0x02,0x04,0x06,0x06,0x02,0x06,0x16,0x12,0x06,0x12,0x0C,0x08,0x0C,0x06,
275     0x0A,0x0C,0x02,0x10,0x02,0x0A,0x02,0x0A,0x12,0x06,0x14,0x04,0x02,0x06,0x16,0x06,
276     0x06,0x12,0x06,0x0E,0x0C,0x10,0x02,0x06,0x06,0x04,0x0E,0x0C,0x04,0x02,0x12,0x10,
277     0x24,0x0C,0x06,0x0E,0x1C,0x02,0x0C,0x06,0x0C,0x06,0x04,0x02,0x10,0x1E,0x08,0x18,
278     0x06,0x1E,0x0A,0x02,0x12,0x04,0x06,0x0C,0x08,0x16,0x02,0x06,0x16,0x12,0x02,0x0A,
279     0x02,0x0A,0x1E,0x02,0x1C,0x06,0x0E,0x10,0x06,0x14,0x10,0x02,0x06,0x04,0x20,0x04,
280     0x02,0x04,0x06,0x02,0x0C,0x04,0x06,0x06,0x0C,0x02,0x06,0x04,0x06,0x08,0x06,0x04,
281     0x14,0x04,0x20,0x0A,0x08,0x10,0x02,0x16,0x02,0x04,0x06,0x08,0x06,0x10,0x0E,0x04,
282     0x12,0x08,0x04,0x14,0x06,0x0C,0x0C,0x06,0x0A,0x02,0x0A,0x02,0x0C,0x1C,0x0C,0x12
283 #endif
284 // 3072
285 #if PRIME_DIFF_TABLE_BYTES > 3072
286     ,0x02,0x12,0x0A,0x08,0x0A,0x30,0x02,0x04,0x06,0x08,0x0A,0x02,0x0A,0x1E,0x02,0x24,
287     0x06,0x0A,0x06,0x02,0x12,0x04,0x06,0x08,0x10,0x0E,0x10,0x06,0x0E,0x04,0x14,0x04,
288     0x06,0x02,0x0A,0x0C,0x02,0x06,0x0C,0x06,0x06,0x04,0x0C,0x02,0x06,0x04,0x0C,0x06,
289     0x08,0x04,0x02,0x06,0x12,0x0A,0x06,0x08,0x0C,0x06,0x16,0x02,0x06,0x0C,0x12,0x04,
290     0x0E,0x06,0x04,0x14,0x06,0x10,0x08,0x04,0x08,0x16,0x08,0x0C,0x06,0x06,0x10,0x0C,
291     0x12,0x1E,0x08,0x04,0x02,0x04,0x06,0x1A,0x04,0x0E,0x18,0x16,0x06,0x02,0x06,0x0A,
292     0x06,0x0E,0x06,0x06,0x0C,0x0A,0x06,0x02,0x0C,0x0A,0x0C,0x08,0x12,0x12,0x0A,0x06,
293     0x08,0x10,0x06,0x06,0x08,0x10,0x14,0x04,0x02,0x0A,0x02,0x0A,0x0C,0x06,0x08,0x06,
294     0x0A,0x14,0x0A,0x12,0x1A,0x04,0x06,0x1E,0x02,0x04,0x08,0x06,0x0C,0x0C,0x12,0x04,
295     0x08,0x16,0x06,0x02,0x0C,0x22,0x06,0x12,0x0C,0x06,0x02,0x1C,0x0E,0x10,0x0E,0x04,
296     0x0E,0x0C,0x04,0x06,0x06,0x02,0x24,0x04,0x06,0x14,0x0C,0x18,0x06,0x16,0x02,0x10,
297     0x12,0x0C,0x0C,0x12,0x02,0x06,0x06,0x06,0x04,0x06,0x0E,0x04,0x02,0x16,0x08,0x0C,
298     0x06,0x0A,0x06,0x08,0x0C,0x12,0x0C,0x06,0x0A,0x02,0x16,0x0E,0x06,0x06,0x04,0x12,
299     0x06,0x14,0x16,0x02,0x0C,0x18,0x04,0x12,0x12,0x02,0x16,0x02,0x04,0x0C,0x08,0x0C,
300     0x0A,0x0E,0x04,0x02,0x12,0x10,0x26,0x06,0x06,0x06,0x0C,0x0A,0x06,0x0C,0x08,0x06,
301     0x04,0x06,0x0E,0x1E,0x06,0x0A,0x08,0x16,0x06,0x08,0x0C,0x0A,0x02,0x0A,0x02,0x06
302 #endif
303 // 3328
304 #if PRIME_DIFF_TABLE_BYTES > 3328
305     ,0x0A,0x02,0x0A,0x0C,0x12,0x14,0x06,0x04,0x08,0x16,0x06,0x06,0x1E,0x06,0x0E,0x06,
```

```
306     0x0C,0x0C,0x06,0x0A,0x02,0x0A,0x1E,0x02,0x10,0x08,0x04,0x02,0x06,0x12,0x04,0x02,
307     0x06,0x04,0x1A,0x04,0x08,0x06,0x0A,0x02,0x04,0x06,0x08,0x04,0x06,0x1E,0x0C,0x02,
308     0x06,0x06,0x04,0x14,0x16,0x08,0x04,0x02,0x04,0x48,0x08,0x04,0x02,0x16,0x02,0x04,
309     0x0E,0x0A,0x02,0x04,0x14,0x06,0x0A,0x12,0x06,0x14,0x10,0x06,0x08,0x06,0x04,0x14,
310     0x0C,0x16,0x02,0x04,0x02,0x0C,0x0A,0x12,0x02,0x16,0x06,0x12,0x1E,0x02,0x0A,0x0E,
311     0x0A,0x08,0x10,0x32,0x06,0x0A,0x08,0x0A,0x0C,0x06,0x12,0x02,0x16,0x06,0x02,0x04,
312     0x06,0x08,0x06,0x06,0x0A,0x12,0x02,0x16,0x02,0x10,0x0E,0x0A,0x06,0x02,0x0C,0x0A,
313     0x14,0x04,0x0E,0x06,0x04,0x24,0x02,0x04,0x06,0x0C,0x02,0x04,0x0E,0x0C,0x06,0x04,
314     0x06,0x02,0x06,0x04,0x14,0x0A,0x02,0x0A,0x06,0x0C,0x02,0x18,0x0C,0x0C,0x06,0x06,
315     0x04,0x18,0x02,0x04,0x18,0x02,0x06,0x04,0x06,0x08,0x10,0x06,0x02,0x0A,0x0C,0x0E,
316     0x06,0x22,0x06,0x0E,0x06,0x04,0x02,0x1E,0x16,0x08,0x04,0x06,0x08,0x04,0x02,0x1C,
317     0x02,0x06,0x04,0x1A,0x12,0x16,0x02,0x06,0x10,0x06,0x02,0x10,0x0C,0x02,0x0C,0x04,
318     0x06,0x06,0x0E,0x0A,0x06,0x08,0x0C,0x04,0x12,0x02,0x0A,0x08,0x10,0x06,0x06,0x1E,
319     0x02,0x0A,0x12,0x02,0x0A,0x08,0x04,0x08,0x0C,0x18,0x28,0x02,0x0C,0x0A,0x06,0x0C,
320     0x02,0x0C,0x04,0x02,0x04,0x06,0x12,0x0E,0x0C,0x06,0x04,0x0E,0x1E,0x04,0x08,0x0A
321 #endif
322 // 3584
323 #if PRIME_DIFF_TABLE_BYTES > 3584
324     ,0x08,0x06,0x0A,0x12,0x08,0x04,0x0E,0x10,0x06,0x08,0x04,0x06,0x02,0x0A,0x02,0x0C,
325     0x04,0x02,0x04,0x06,0x08,0x04,0x06,0x20,0x18,0x0A,0x08,0x12,0x0A,0x02,0x06,0x0A,
326     0x02,0x04,0x12,0x06,0x0C,0x02,0x10,0x02,0x16,0x06,0x06,0x08,0x12,0x04,0x12,0x0C,
327     0x08,0x06,0x04,0x14,0x06,0x1E,0x16,0x0C,0x02,0x06,0x12,0x04,0x3E,0x04,0x02,0x0C,
328     0x06,0x0A,0x02,0x0C,0x0C,0x1C,0x02,0x04,0x0E,0x16,0x06,0x02,0x06,0x06,0x0A,0x0E,
329     0x04,0x02,0x0A,0x06,0x08,0x0A,0x0E,0x0A,0x06,0x02,0x0C,0x16,0x12,0x08,0x0A,0x12,
330     0x0C,0x02,0x0C,0x04,0x0C,0x02,0x0A,0x02,0x06,0x12,0x06,0x06,0x22,0x06,0x02,0x0C,
331     0x04,0x06,0x12,0x12,0x02,0x10,0x06,0x06,0x08,0x06,0x0A,0x12,0x08,0x0A,0x08,0x0A,
332     0x02,0x04,0x12,0x1A,0x0C,0x16,0x02,0x04,0x02,0x16,0x06,0x06,0x0E,0x10,0x06,0x14,
333     0x0A,0x0C,0x02,0x12,0x2A,0x04,0x18,0x02,0x06,0x0A,0x0C,0x02,0x06,0x0A,0x08,0x04,
334     0x06,0x0C,0x0C,0x08,0x04,0x06,0x0C,0x1E,0x14,0x06,0x18,0x06,0x0A,0x0C,0x02,0x0A,
335     0x14,0x06,0x06,0x04,0x0C,0x0E,0x0A,0x12,0x0C,0x08,0x06,0x0C,0x04,0x0E,0x0A,0x02,
336     0x0C,0x1E,0x10,0x02,0x0C,0x06,0x04,0x02,0x04,0x06,0x1A,0x04,0x12,0x02,0x04,0x06,
337     0x0E,0x36,0x06,0x34,0x02,0x10,0x06,0x06,0x0C,0x1A,0x04,0x02,0x06,0x16,0x06,0x02,
338     0x0C,0x0C,0x06,0x0A,0x12,0x02,0x0C,0x0C,0x0A,0x12,0x0C,0x06,0x08,0x06,0x0A,0x06,
339     0x08,0x04,0x02,0x04,0x14,0x18,0x06,0x06,0x0A,0x0E,0x0A,0x02,0x16,0x06,0x0E,0x0A
340 #endif
341 // 3840
342 #if PRIME_DIFF_TABLE_BYTES > 3840
343     ,0x1A,0x04,0x12,0x08,0x0C,0x0C,0x0A,0x0C,0x06,0x08,0x10,0x06,0x08,0x06,0x06,0x16,
344     0x02,0x0A,0x14,0x0A,0x06,0x2C,0x12,0x06,0x0A,0x02,0x04,0x06,0x0E,0x04,0x1A,0x04,
345     0x02,0x0C,0x0A,0x08,0x04,0x08,0x0C,0x04,0x0C,0x08,0x16,0x08,0x06,0x0A,0x12,0x06,
346     0x06,0x08,0x06,0x0C,0x04,0x08,0x12,0x0A,0x0C,0x06,0x0C,0x02,0x06,0x04,0x02,0x10,
347     0x0C,0x0C,0x0E,0x0A,0x0E,0x06,0x0A,0x0C,0x02,0x0C,0x06,0x04,0x06,0x02,0x0C,0x04,
348     0x1A,0x06,0x12,0x06,0x0A,0x06,0x02,0x12,0x0A,0x08,0x04,0x1A,0x0A,0x14,0x06,0x10,
349     0x14,0x0C,0x0A,0x08,0x0A,0x02,0x10,0x06,0x14,0x0A,0x14,0x04,0x1E,0x02,0x04,0x08,
350     0x10,0x02,0x12,0x04,0x02,0x06,0x0A,0x12,0x0C,0x0E,0x12,0x06,0x10,0x14,0x06,0x04,
351     0x08,0x06,0x04,0x06,0x0C,0x08,0x0A,0x02,0x0C,0x06,0x04,0x02,0x06,0x0A,0x02,0x10,
352     0x0C,0x0E,0x0A,0x06,0x08,0x06,0x1C,0x02,0x06,0x12,0x1E,0x22,0x02,0x10,0x0C,0x02,
353     0x12,0x10,0x06,0x08,0x0A,0x08,0x0A,0x08,0x0A,0x2C,0x06,0x06,0x04,0x14,0x04,0x02,
354     0x04,0x0E,0x1C,0x08,0x06,0x10,0x0E,0x1E,0x06,0x1E,0x04,0x0E,0x0A,0x06,0x06,0x08,
355     0x04,0x12,0x0C,0x06,0x02,0x16,0x0C,0x08,0x06,0x0C,0x04,0x0E,0x04,0x06,0x02,0x04,
356     0x12,0x14,0x06,0x10,0x26,0x10,0x02,0x04,0x06,0x02,0x28,0x2A,0x0E,0x04,0x06,0x02,
357     0x18,0x0A,0x06,0x02,0x12,0x0A,0x0C,0x02,0x10,0x02,0x06,0x10,0x06,0x08,0x04,0x02,
358     0x0A,0x06,0x08,0x0A,0x02,0x12,0x10,0x08,0x0C,0x12,0x0C,0x06,0x0C,0x0A,0x06,0x06
359 #endif
360 // 4096
361 #if PRIME_DIFF_TABLE_BYTES > 4096
362     ,0x12,0x0C,0x0E,0x04,0x02,0x0A,0x14,0x06,0x0C,0x06,0x10,0x1A,0x04,0x12,0x02,0x04,
363     0x20,0x0A,0x08,0x06,0x04,0x06,0x06,0x0E,0x06,0x12,0x04,0x02,0x12,0x0A,0x08,0x0A,
364     0x08,0x0A,0x02,0x04,0x06,0x02,0x0A,0x2A,0x08,0x0C,0x04,0x06,0x12,0x02,0x10,0x08,
365     0x04,0x02,0x0A,0x0E,0x0C,0x0A,0x14,0x04,0x08,0x0A,0x26,0x04,0x06,0x02,0x0A,0x14,
366     0x0A,0x0C,0x06,0x0C,0x1A,0x0C,0x04,0x08,0x1C,0x08,0x04,0x08,0x18,0x06,0x0A,0x08,
367     0x06,0x10,0x0C,0x08,0x0A,0x0C,0x08,0x16,0x06,0x02,0x0A,0x02,0x06,0x0A,0x06,0x06,
368     0x08,0x06,0x04,0x0E,0x1C,0x08,0x10,0x12,0x08,0x04,0x06,0x14,0x04,0x12,0x06,0x02,
369     0x18,0x18,0x06,0x06,0x0C,0x0C,0x04,0x02,0x16,0x02,0x0A,0x06,0x08,0x0C,0x04,0x14,
370     0x12,0x06,0x04,0x0C,0x18,0x06,0x06,0x36,0x08,0x06,0x04,0x1A,0x24,0x04,0x02,0x04,
371     0x1A,0x0C,0x0C,0x04,0x06,0x06,0x08,0x0C,0x0A,0x02,0x0C,0x10,0x12,0x06,0x08,0x06,
```

```
372     0x0C,0x12,0x0A,0x02,0x36,0x04,0x02,0x0A,0x1E,0x0C,0x08,0x04,0x08,0x10,0x0E,0x0C,
373     0x06,0x04,0x06,0x0C,0x06,0x02,0x04,0x0E,0x0C,0x04,0x0E,0x06,0x18,0x06,0x06,0x0A,
374     0x0C,0x0C,0x14,0x12,0x06,0x06,0x10,0x08,0x04,0x06,0x14,0x04,0x20,0x04,0x0E,0x0A,
375     0x02,0x06,0x0C,0x10,0x02,0x04,0x06,0x0C,0x02,0x0A,0x08,0x06,0x04,0x02,0x0A,0x0E,
376     0x06,0x06,0x0C,0x12,0x22,0x08,0x0A,0x06,0x18,0x06,0x02,0x0A,0x0C,0x02,0x1E,0x0A,
377     0x0E,0x0C,0x0C,0x10,0x06,0x06,0x02,0x12,0x04,0x06,0x1E,0x0E,0x04,0x06,0x06,0x02
378 #endif
379 // 4352
380 #if PRIME_DIFF_TABLE_BYTES > 4352
381     ,0x06,0x04,0x06,0x0E,0x06,0x04,0x08,0x0A,0x0C,0x06,0x20,0x0A,0x08,0x16,0x02,0x0A,
382     0x06,0x18,0x08,0x04,0x1E,0x06,0x02,0x0C,0x10,0x08,0x06,0x04,0x06,0x08,0x10,0x0E,
383     0x06,0x06,0x04,0x02,0x0A,0x0C,0x02,0x10,0x0E,0x04,0x02,0x04,0x14,0x12,0x0A,0x02,
384     0x0A,0x06,0x0C,0x1E,0x08,0x12,0x0C,0x0A,0x02,0x06,0x06,0x04,0x0C,0x0C,0x02,0x04,
385     0x0C,0x12,0x18,0x02,0x0A,0x06,0x08,0x10,0x08,0x06,0x0C,0x0A,0x0E,0x06,0x0C,0x06,
386     0x06,0x04,0x02,0x18,0x04,0x06,0x08,0x06,0x04,0x02,0x04,0x06,0x0E,0x04,0x08,0x0A,
387     0x18,0x18,0x0C,0x02,0x06,0x0C,0x16,0x1E,0x02,0x06,0x12,0x0A,0x06,0x06,0x08,0x04,
388     0x02,0x06,0x0A,0x08,0x0A,0x06,0x06,0x10,0x06,0x0E,0x06,0x04,0x18,0x08,0x0A,0x02,
389     0x0C,0x06,0x04,0x24,0x02,0x16,0x06,0x08,0x06,0x0A,0x08,0x06,0x0C,0x0A,0x0E,0x0A,
390     0x06,0x12,0x0C,0x02,0x0C,0x04,0x1A,0x0A,0x0E,0x10,0x12,0x08,0x12,0x0C,0x0C,0x06,
391     0x10,0x0E,0x18,0x0A,0x0C,0x08,0x16,0x06,0x02,0x0A,0x3C,0x06,0x02,0x04,0x08,0x10,
392     0x0E,0x0A,0x06,0x18,0x06,0x0C,0x12,0x18,0x02,0x1E,0x04,0x02,0x0C,0x06,0x0A,0x02,
393     0x04,0x0E,0x06,0x10,0x02,0x0A,0x08,0x16,0x14,0x06,0x04,0x20,0x06,0x12,0x04,0x02,
394     0x04,0x02,0x04,0x08,0x34,0x0E,0x16,0x02,0x16,0x14,0x0A,0x08,0x0A,0x02,0x06,0x04,
395     0x0E,0x04,0x06,0x14,0x04,0x06,0x02,0x0C,0x0C,0x06,0x0C,0x10,0x02,0x0C,0x0A,0x08,
396     0x04,0x06,0x02,0x1C,0x0C,0x08,0x0A,0x0C,0x02,0x04,0x0E,0x1C,0x08,0x06,0x04,0x02
397 #endif
398 // 4608
399 #if PRIME_DIFF_TABLE_BYTES > 4608
400     ,0x04,0x06,0x02,0x0C,0x3A,0x06,0x0E,0x0A,0x02,0x06,0x1C,0x20,0x04,0x1E,0x08,0x06,
401     0x04,0x06,0x0C,0x0C,0x02,0x04,0x06,0x06,0x0E,0x10,0x08,0x1E,0x04,0x02,0x0A,0x08,
402     0x06,0x04,0x06,0x1A,0x04,0x0C,0x02,0x0A,0x12,0x0C,0x0C,0x12,0x02,0x04,0x0C,0x08,
403     0x0C,0x0A,0x14,0x04,0x08,0x10,0x0C,0x08,0x06,0x10,0x08,0x0A,0x0C,0x0E,0x06,0x04,
404     0x08,0x0C,0x04,0x14,0x06,0x28,0x08,0x10,0x06,0x24,0x02,0x06,0x04,0x06,0x02,0x16,
405     0x12,0x02,0x0A,0x06,0x24,0x0E,0x0C,0x04,0x12,0x08,0x04,0x0E,0x0A,0x02,0x0A,0x08,
406     0x04,0x02,0x12,0x10,0x0C,0x0E,0x0A,0x0E,0x06,0x06,0x2A,0x0A,0x06,0x06,0x14,0x0A,
407     0x08,0x0C,0x04,0x0C,0x12,0x02,0x0A,0x0E,0x12,0x0A,0x12,0x08,0x06,0x04,0x0E,0x06,
408     0x0A,0x1E,0x0E,0x06,0x06,0x04,0x0C,0x26,0x04,0x02,0x04,0x06,0x08,0x0C,0x0A,0x06,
409     0x12,0x06,0x32,0x06,0x04,0x06,0x0C,0x08,0x0A,0x20,0x06,0x16,0x02,0x0A,0x0C,0x12,
410     0x02,0x06,0x04,0x1E,0x08,0x06,0x06,0x12,0x0A,0x02,0x04,0x0C,0x14,0x0A,0x08,0x18,
411     0x0A,0x02,0x06,0x16,0x06,0x02,0x12,0x0A,0x0C,0x02,0x1E,0x12,0x0C,0x1C,0x02,0x06,
412     0x04,0x06,0x0E,0x06,0x0C,0x0A,0x08,0x04,0x0C,0x1A,0x0A,0x08,0x06,0x10,0x02,0x0A,
413     0x12,0x0E,0x06,0x04,0x06,0x0E,0x10,0x02,0x06,0x04,0x0C,0x14,0x04,0x14,0x04,0x06,
414     0x0C,0x02,0x24,0x04,0x06,0x02,0x0A,0x02,0x16,0x08,0x06,0x0A,0x0C,0x0C,0x12,0x0E,
415     0x18,0x24,0x04,0x14,0x18,0x0A,0x06,0x02,0x1C,0x06,0x12,0x08,0x04,0x06,0x08,0x06
416 #endif
417 // 4864
418 #if PRIME_DIFF_TABLE_BYTES > 4864
419     ,0x04,0x02,0x0C,0x1C,0x12,0x0E,0x10,0x0E,0x12,0x0A,0x08,0x06,0x04,0x06,0x06,0x08,
420     0x16,0x0C,0x02,0x0A,0x12,0x06,0x02,0x12,0x0A,0x02,0x0C,0x0A,0x12,0x20,0x06,0x04,
421     0x06,0x06,0x08,0x06,0x06,0x0A,0x14,0x06,0x0C,0x0A,0x08,0x0A,0x0E,0x06,0x0A,0x0E,
422     0x04,0x02,0x16,0x12,0x02,0x0A,0x02,0x04,0x14,0x04,0x02,0x22,0x02,0x0C,0x06,0x0A,
423     0x02,0x0A,0x12,0x06,0x0E,0x0C,0x0C,0x16,0x08,0x06,0x10,0x06,0x08,0x04,0x0C,0x06,
424     0x08,0x04,0x24,0x06,0x06,0x14,0x18,0x06,0x0C,0x12,0x0A,0x02,0x0A,0x1A,0x06,0x10,
425     0x08,0x06,0x04,0x18,0x12,0x08,0x0C,0x0C,0x0A,0x12,0x0C,0x02,0x18,0x04,0x0C,0x12,
426     0x0C,0x0E,0x0A,0x02,0x04,0x18,0x0C,0x0E,0x0A,0x06,0x02,0x06,0x04,0x06,0x1A,0x04,
427     0x06,0x06,0x02,0x16,0x08,0x12,0x04,0x12,0x08,0x04,0x18,0x02,0x0C,0x0C,0x04,0x02,
428     0x34,0x02,0x12,0x06,0x04,0x06,0x0C,0x02,0x06,0x0C,0x0A,0x08,0x04,0x02,0x18,0x0A,
429     0x02,0x0A,0x02,0x0C,0x06,0x12,0x28,0x06,0x14,0x10,0x02,0x0C,0x06,0x0A,0x0C,0x02,
430     0x04,0x06,0x0E,0x0C,0x0C,0x16,0x06,0x08,0x04,0x02,0x10,0x12,0x0C,0x02,0x06,0x10,
431     0x06,0x02,0x06,0x04,0x0C,0x1E,0x08,0x10,0x02,0x12,0x0A,0x18,0x02,0x06,0x18,0x04,
432     0x02,0x16,0x02,0x10,0x02,0x06,0x0C,0x04,0x12,0x08,0x04,0x0E,0x04,0x12,0x18,0x06,
433     0x02,0x06,0x0A,0x02,0x0A,0x26,0x06,0x0A,0x0E,0x06,0x06,0x18,0x04,0x02,0x0C,0x10,
434     0x0E,0x10,0x0C,0x02,0x06,0x0A,0x1A,0x04,0x02,0x0C,0x06,0x04,0x0C,0x08,0x0C,0x0A
435 #endif
436 // 5120
437 #if PRIME_DIFF_TABLE_BYTES > 5120
```

```
438     ,0x12,0x06,0x0E,0x1C,0x02,0x06,0x0A,0x02,0x04,0x0E,0x22,0x02,0x06,0x16,0x02,0x0A,
439     0x0E,0x04,0x02,0x10,0x08,0x0A,0x06,0x08,0x0A,0x08,0x04,0x06,0x02,0x10,0x06,0x06,
440     0x12,0x1E,0x0E,0x06,0x04,0x18,0x1E,0x02,0x0A,0x0E,0x04,0x14,0x0A,0x08,0x04,0x08,0x12,
441     0x04,0x0E,0x06,0x04,0x18,0x06,0x06,0x12,0x12,0x02,0x24,0x06,0x0A,0x0E,0x0C,0x04,
442     0x06,0x02,0x1E,0x06,0x04,0x02,0x06,0x1C,0x14,0x04,0x14,0x0C,0x18,0x10,0x12,0x0C,
443     0x0E,0x06,0x04,0x0C,0x20,0x0C,0x06,0x0A,0x08,0x0A,0x06,0x12,0x02,0x10,0x0E,0x06,
444     0x16,0x06,0x0C,0x02,0x12,0x04,0x08,0x1E,0x0C,0x04,0x0C,0x02,0x0A,0x26,0x16,0x02,
445     0x04,0x0E,0x06,0x0C,0x18,0x04,0x02,0x04,0x0E,0x0C,0x0A,0x02,0x10,0x06,0x14,0x04,
446     0x14,0x16,0x0C,0x02,0x04,0x02,0x0C,0x16,0x18,0x06,0x06,0x02,0x06,0x04,0x06,0x02,
447     0x0A,0x0C,0x0C,0x06,0x02,0x06,0x10,0x08,0x06,0x04,0x12,0x0C,0x0C,0x0E,0x04,0x0C,
448     0x06,0x08,0x06,0x12,0x06,0x0A,0x0C,0x0E,0x06,0x04,0x08,0x16,0x06,0x02,0x1C,0x12,
449     0x02,0x12,0x0A,0x06,0x0E,0x0A,0x02,0x0A,0x0E,0x06,0x0A,0x02,0x16,0x06,0x08,0x06,
450     0x10,0x0C,0x08,0x16,0x02,0x04,0x0E,0x12,0x0C,0x06,0x18,0x06,0x0A,0x02,0x0C,0x16,
451     0x12,0x06,0x14,0x06,0x0A,0x0E,0x04,0x02,0x06,0x0C,0x16,0x0E,0x0C,0x04,0x06,0x08,
452     0x16,0x02,0x0A,0x0C,0x08,0x28,0x02,0x06,0x0A,0x08,0x04,0x2A,0x14,0x04,0x20,0x0C,
453     0x0A,0x06,0x0C,0x0C,0x02,0x0A,0x08,0x06,0x04,0x08,0x04,0x1A,0x12,0x04,0x08,0x1C
454 #endif
455 // 5376
456 #if PRIME_DIFF_TABLE_BYTES > 5376
457     ,0x06,0x12,0x06,0x0C,0x02,0x0A,0x06,0x06,0x0E,0x0A,0x0C,0x0E,0x18,0x06,0x04,0x14,
458     0x16,0x02,0x12,0x04,0x06,0x0C,0x02,0x10,0x12,0x0E,0x06,0x06,0x04,0x06,0x08,0x12,
459     0x04,0x0E,0x1E,0x04,0x12,0x08,0x0A,0x02,0x04,0x08,0x0C,0x04,0x0C,0x12,0x02,0x0C,
460     0x0A,0x02,0x10,0x08,0x04,0x1E,0x02,0x06,0x1C,0x02,0x0A,0x02,0x12,0x0A,0x0E,0x04,
461     0x1A,0x06,0x12,0x04,0x14,0x06,0x04,0x08,0x12,0x04,0x0C,0x1A,0x18,0x04,0x14,0x16,
462     0x02,0x12,0x16,0x02,0x04,0x0C,0x02,0x06,0x06,0x06,0x04,0x06,0x0E,0x04,0x18,0x0C,
463     0x06,0x12,0x02,0x0C,0x1C,0x0E,0x04,0x06,0x08,0x16,0x06,0x0C,0x12,0x08,0x04,0x14,
464     0x06,0x04,0x06,0x02,0x12,0x06,0x04,0x0C,0x0C,0x08,0x1C,0x06,0x08,0x0A,0x02,0x18,
465     0x0C,0x0A,0x18,0x08,0x0A,0x14,0x0C,0x06,0x0C,0x0C,0x04,0x0E,0x0C,0x18,0x22,0x12,
466     0x08,0x0A,0x06,0x12,0x08,0x04,0x08,0x10,0x0E,0x06,0x04,0x06,0x18,0x02,0x06,0x04,
467     0x06,0x02,0x10,0x06,0x06,0x14,0x18,0x04,0x02,0x04,0x0E,0x04,0x12,0x02,0x06,0x0C,
468     0x04,0x0E,0x04,0x02,0x12,0x10,0x06,0x06,0x02,0x10,0x14,0x06,0x06,0x1E,0x04,0x08,
469     0x06,0x18,0x10,0x06,0x06,0x08,0x0C,0x1E,0x04,0x12,0x12,0x08,0x04,0x1A,0x0A,0x02,
470     0x16,0x08,0x0A,0x0E,0x06,0x04,0x12,0x08,0x0C,0x1C,0x02,0x06,0x04,0x0C,0x06,0x18,
471     0x06,0x08,0x0A,0x14,0x10,0x08,0x1E,0x06,0x06,0x04,0x02,0x0A,0x0E,0x06,0x0A,0x20,
472     0x16,0x12,0x02,0x04,0x02,0x04,0x08,0x16,0x08,0x12,0x0C,0x1C,0x02,0x10,0x0C,0x12
473 #endif
474 // 5632
475 #if PRIME_DIFF_TABLE_BYTES > 5632
476     ,0x0E,0x0A,0x12,0x0C,0x06,0x20,0x0A,0x0E,0x06,0x0A,0x02,0x0A,0x02,0x06,0x16,0x02,
477     0x04,0x06,0x08,0x0A,0x06,0x0E,0x06,0x04,0x0C,0x1E,0x18,0x06,0x06,0x08,0x06,0x04,
478     0x02,0x04,0x06,0x08,0x06,0x06,0x16,0x12,0x08,0x04,0x02,0x12,0x06,0x04,0x02,0x10,
479     0x12,0x14,0x0A,0x06,0x06,0x1E,0x02,0x0C,0x1C,0x06,0x06,0x06,0x02,0x0C,0x0A,0x08,
480     0x12,0x12,0x04,0x08,0x12,0x0A,0x02,0x1C,0x02,0x0A,0x0E,0x04,0x02,0x1E,0x0C,0x16,
481     0x1A,0x0A,0x08,0x06,0x0A,0x08,0x10,0x0E,0x06,0x06,0x0E,0x06,0x04,0x02,0x0A,
482     0x0C,0x02,0x06,0x0A,0x08,0x04,0x02,0x0A,0x1A,0x16,0x06,0x02,0x0C,0x12,0x04,0x1A,
483     0x04,0x08,0x0A,0x06,0x0E,0x0A,0x02,0x12,0x06,0x0A,0x14,0x06,0x06,0x04,0x18,0x02,
484     0x04,0x08,0x06,0x10,0x0E,0x10,0x12,0x02,0x04,0x0C,0x02,0x0A,0x02,0x06,0x0C,0x0A,
485     0x06,0x06,0x14,0x06,0x04,0x06,0x26,0x04,0x06,0x0C,0x0E,0x04,0x0C,0x08,0x0A,0x0C,
486     0x0C,0x08,0x04,0x06,0x0E,0x0A,0x06,0x0C,0x02,0x0A,0x12,0x02,0x12,0x0A,0x08,0x0A,
487     0x02,0x0C,0x04,0x0E,0x1C,0x02,0x10,0x02,0x12,0x06,0x0A,0x06,0x08,0x10,0x0E,0x1E,
488     0x0A,0x14,0x06,0x0A,0x18,0x02,0x1C,0x02,0x0C,0x10,0x06,0x08,0x24,0x04,0x08,0x04,
489     0x0E,0x0C,0x0A,0x08,0x0C,0x04,0x06,0x08,0x04,0x06,0x0E,0x16,0x08,0x06,0x04,0x02,
490     0x0A,0x06,0x14,0x0A,0x08,0x06,0x06,0x16,0x12,0x02,0x10,0x06,0x14,0x04,0x1A,0x04,
491     0x0E,0x16,0x0E,0x04,0x0C,0x06,0x08,0x04,0x06,0x06,0x1A,0x0A,0x02,0x12,0x12,0x04
492 #endif
493 // 5888
494 #if PRIME_DIFF_TABLE_BYTES > 5888
495     ,0x02,0x10,0x02,0x12,0x04,0x06,0x08,0x04,0x06,0x0C,0x02,0x06,0x06,0x1C,0x26,0x04,
496     0x08,0x10,0x1A,0x04,0x02,0x0A,0x0C,0x02,0x0A,0x08,0x06,0x0A,0x0C,0x02,0x0A,0x02,
497     0x18,0x04,0x1E,0x1A,0x06,0x06,0x12,0x06,0x06,0x16,0x02,0x0A,0x12,0x1A,0x04,0x12,
498     0x08,0x06,0x06,0x0C,0x10,0x06,0x08,0x10,0x06,0x08,0x10,0x02,0x2A,0x3A,0x08,0x04,
499     0x06,0x02,0x04,0x08,0x10,0x06,0x14,0x04,0x0C,0x0C,0x06,0x0C,0x02,0x0A,0x02,0x06,
500     0x16,0x02,0x0A,0x06,0x08,0x06,0x0A,0x0E,0x06,0x06,0x04,0x12,0x08,0x0A,0x08,0x10,
501     0x0E,0x0A,0x02,0x0A,0x02,0x0C,0x06,0x04,0x14,0x0A,0x08,0x34,0x08,0x0A,0x06,0x02,
502     0x0A,0x08,0x0A,0x06,0x06,0x08,0x0A,0x02,0x16,0x02,0x04,0x06,0x0E,0x04,0x02,0x18,
503     0x0C,0x04,0x1A,0x12,0x04,0x06,0x0E,0x1E,0x06,0x04,0x06,0x02,0x16,0x08,0x04,0x06,
```



```

504     0x02, 0x16, 0x06, 0x08, 0x10, 0x06, 0x0E, 0x04, 0x06, 0x12, 0x08, 0x0C, 0x06, 0x0C, 0x18, 0x1E,
505     0x10, 0x08, 0x22, 0x08, 0x16, 0x06, 0x0E, 0x0A, 0x12, 0x0E, 0x04, 0x0C, 0x08, 0x04, 0x24, 0x06,
506     0x06, 0x02, 0x0A, 0x02, 0x04, 0x14, 0x06, 0x06, 0x0A, 0x0C, 0x06, 0x02, 0x28, 0x08, 0x06, 0x1C,
507     0x06, 0x02, 0x0C, 0x12, 0x04, 0x18, 0x0E, 0x06, 0x06, 0x0A, 0x14, 0x0A, 0x0E, 0x10, 0x0E, 0x10,
508     0x06, 0x08, 0x24, 0x04, 0x0C, 0x0C, 0x06, 0x0C, 0x32, 0x0C, 0x06, 0x04, 0x06, 0x06, 0x08, 0x06,
509     0x0A, 0x02, 0x0A, 0x02, 0x12, 0x0A, 0x0E, 0x10, 0x08, 0x06, 0x04, 0x14, 0x04, 0x02, 0x0A, 0x06,
510     0x0E, 0x12, 0x0A, 0x26, 0x0A, 0x12, 0x02, 0x0A, 0x02, 0x0C, 0x04, 0x02, 0x04, 0x0E, 0x06, 0x0A
511 #endif
512 // 6144
513 #if PRIME_DIFF_TABLE_BYTES > 6144
514     , 0x08, 0x28, 0x06, 0x14, 0x04, 0x0C, 0x08, 0x06, 0x22, 0x08, 0x16, 0x08, 0x0C, 0x0A, 0x02, 0x10,
515     0x2A, 0x0C, 0x08, 0x16, 0x08, 0x16, 0x08, 0x06, 0x22, 0x02, 0x06, 0x04, 0x0E, 0x06, 0x10, 0x02,
516     0x16, 0x06, 0x08, 0x18, 0x16, 0x06, 0x02, 0x0C, 0x04, 0x06, 0x0E, 0x04, 0x08, 0x18, 0x04, 0x06,
517     0x06, 0x02, 0x16, 0x14, 0x06, 0x04, 0x0E, 0x04, 0x06, 0x06, 0x08, 0x06, 0x0A, 0x06, 0x08, 0x06,
518     0x10, 0x0E, 0x06, 0x06, 0x16, 0x06, 0x18, 0x20, 0x06, 0x12, 0x06, 0x12, 0x0A, 0x08, 0x1E, 0x12,
519     0x06, 0x10, 0x0C, 0x06, 0x0C, 0x02, 0x06, 0x04, 0x0C, 0x08, 0x06, 0x16, 0x08, 0x06, 0x04, 0x0E,
520     0x0A, 0x12, 0x14, 0x0A, 0x02, 0x06, 0x04, 0x02, 0x1C, 0x12, 0x02, 0x0A, 0x06, 0x06, 0x06, 0x0E,
521     0x28, 0x18, 0x02, 0x04, 0x08, 0x0C, 0x04, 0x14, 0x04, 0x20, 0x12, 0x10, 0x06, 0x24, 0x08, 0x06,
522     0x04, 0x06, 0x0E, 0x04, 0x06, 0x1A, 0x06, 0x0A, 0x0E, 0x12, 0x0A, 0x06, 0x06, 0x0E, 0x0A, 0x06,
523     0x06, 0x0E, 0x06, 0x18, 0x04, 0x0E, 0x16, 0x08, 0x0C, 0x0A, 0x08, 0x0C, 0x12, 0x0A, 0x12, 0x08,
524     0x18, 0x0A, 0x08, 0x04, 0x18, 0x06, 0x12, 0x06, 0x02, 0x0A, 0x1E, 0x02, 0x0A, 0x02, 0x04, 0x02,
525     0x28, 0x02, 0x1C, 0x08, 0x06, 0x06, 0x12, 0x06, 0x0A, 0x0E, 0x04, 0x12, 0x1E, 0x12, 0x02, 0x0C,
526     0x1E, 0x06, 0x1E, 0x04, 0x12, 0x0C, 0x02, 0x04, 0x0E, 0x06, 0x0A, 0x06, 0x08, 0x0A, 0x0C,
527     0x02, 0x06, 0x0C, 0x0A, 0x02, 0x12, 0x04, 0x14, 0x04, 0x06, 0x0E, 0x06, 0x06, 0x16, 0x06, 0x06,
528     0x08, 0x12, 0x12, 0x0A, 0x02, 0x0A, 0x02, 0x06, 0x04, 0x06, 0x0C, 0x12, 0x02, 0x0A, 0x08, 0x04,
529     0x12, 0x02, 0x06, 0x06, 0x06, 0x0A, 0x08, 0x0A, 0x06, 0x12, 0x0C, 0x08, 0x0C, 0x06, 0x04, 0x06
530 #endif
531 // 6400
532 #if PRIME_DIFF_TABLE_BYTES > 6400
533     , 0x0E, 0x10, 0x02, 0x0C, 0x04, 0x06, 0x26, 0x06, 0x06, 0x10, 0x14, 0x1C, 0x14, 0x0A, 0x06, 0x06,
534     0x0E, 0x04, 0x1A, 0x04, 0x0E, 0x0A, 0x12, 0x0E, 0x1C, 0x02, 0x04, 0x0E, 0x10, 0x02, 0x1C, 0x06,
535     0x08, 0x06, 0x22, 0x08, 0x04, 0x12, 0x02, 0x10, 0x08, 0x06, 0x28, 0x08, 0x12, 0x04, 0x1E, 0x06,
536     0x0C, 0x02, 0x1E, 0x06, 0x0A, 0x0E, 0x28, 0x0E, 0x0A, 0x02, 0x0C, 0x0A, 0x08, 0x04, 0x08, 0x06,
537     0x06, 0x1C, 0x02, 0x04, 0x0C, 0x0E, 0x10, 0x08, 0x1E, 0x10, 0x12, 0x02, 0x0A, 0x12, 0x06, 0x20,
538     0x04, 0x12, 0x06, 0x02, 0x0C, 0x0A, 0x12, 0x02, 0x06, 0x0A, 0x0E, 0x12, 0x1C, 0x06, 0x08, 0x10,
539     0x02, 0x04, 0x14, 0x0A, 0x08, 0x12, 0x0A, 0x02, 0x0A, 0x08, 0x04, 0x06, 0x0C, 0x06, 0x14, 0x04,
540     0x02, 0x06, 0x04, 0x14, 0x0A, 0x1A, 0x12, 0x0A, 0x02, 0x12, 0x06, 0x10, 0x0E, 0x04, 0x1A, 0x04,
541     0x0E, 0x0A, 0x0C, 0x0E, 0x06, 0x06, 0x04, 0x0E, 0x0A, 0x02, 0x1E, 0x12, 0x16, 0x02
542 #endif
543 // 6542
544 #if PRIME_DIFF_TABLE_BYTES > 0
545     };
546 #endif
547 #if defined RSA_INSTRUMENT || defined RSA_DEBUG
548     UINT32 failedAtIteration[10];
549     UINT32 MillerRabinTrials;
550     UINT32 totalFields;
551     UINT32 emptyFields;
552     UINT32 noPrimeFields;
553     UINT16 lastSievePrime;
554     UINT32 primesChecked;
555 #endif

```

Only want this table when doing debug of the prime number stuff This is a table of the first 2048 primes and takes 4096 bytes

```

556 #ifdef RSA_DEBUG
557 const __int16 primes[NUM_PRIMES]=
558 {
559     3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
560     59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
561     137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
562     227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311,
563     313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
564     419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,

```

565 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613,  
566 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719,  
567 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827,  
568 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941,  
569 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021, 1031, 1033, 1039, 1049,  
570 1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129, 1151, 1153, 1163,  
571 1171, 1181, 1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237, 1249, 1259, 1277, 1279, 1283,  
572 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321, 1327, 1361, 1367, 1373, 1381, 1399, 1409, 1423,  
573 1427, 1429, 1433, 1439, 1447, 1451, 1453, 1459, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1511,  
574 1523, 1531, 1543, 1549, 1553, 1559, 1567, 1571, 1579, 1583, 1597, 1601, 1607, 1609, 1613, 1619,  
575 1621, 1627, 1637, 1657, 1663, 1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741, 1747,  
576 1753, 1759, 1777, 1783, 1787, 1789, 1801, 1811, 1823, 1831, 1847, 1861, 1867, 1871, 1873, 1877,  
577 1879, 1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951, 1973, 1979, 1987, 1993, 1997, 1999, 2003,  
578 2011, 2017, 2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083, 2087, 2089, 2099, 2111, 2113, 2129,  
579 2131, 2137, 2141, 2143, 2153, 2161, 2179, 2203, 2207, 2213, 2221, 2237, 2239, 2243, 2251, 2267,  
580 2269, 2273, 2281, 2287, 2293, 2297, 2309, 2311, 2333, 2339, 2341, 2347, 2351, 2357, 2371, 2377,  
581 2381, 2383, 2389, 2393, 2399, 2411, 2417, 2423, 2437, 2441, 2447, 2459, 2467, 2473, 2477, 2503,  
582 2521, 2531, 2539, 2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609, 2617, 2621, 2633, 2647, 2657,  
583 2659, 2663, 2671, 2677, 2683, 2687, 2689, 2693, 2699, 2707, 2711, 2713, 2719, 2729, 2731, 2741,  
584 2749, 2753, 2767, 2777, 2789, 2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843, 2851, 2857, 2861,  
585 2879, 2887, 2897, 2903, 2909, 2917, 2927, 2939, 2953, 2957, 2963, 2969, 2971, 2999, 3001, 3011,  
586 3019, 3023, 3037, 3041, 3049, 3061, 3067, 3079, 3083, 3089, 3109, 3119, 3121, 3137, 3163, 3167,  
587 3169, 3181, 3187, 3191, 3203, 3209, 3217, 3221, 3229, 3251, 3253, 3257, 3259, 3271, 3299, 3301,  
588 3307, 3313, 3319, 3323, 3329, 3331, 3343, 3347, 3359, 3361, 3371, 3373, 3389, 3391, 3407, 3413,  
589 3433, 3449, 3457, 3461, 3463, 3467, 3469, 3491, 3499, 3511, 3517, 3527, 3529, 3533, 3539, 3541,  
590 3547, 3557, 3559, 3571, 3581, 3583, 3593, 3607, 3613, 3617, 3623, 3631, 3637, 3643, 3659, 3671,  
591 3673, 3677, 3691, 3697, 3701, 3709, 3719, 3727, 3733, 3739, 3761, 3767, 3769, 3779, 3793, 3797,  
592 3803, 3821, 3823, 3833, 3847, 3851, 3853, 3863, 3877, 3881, 3889, 3907, 3911, 3917, 3919, 3923,  
593 3929, 3931, 3943, 3947, 3967, 3989, 4001, 4003, 4007, 4013, 4019, 4021, 4027, 4049, 4051, 4057,  
594 4073, 4079, 4091, 4093, 4099, 4111, 4127, 4129, 4133, 4139, 4153, 4157, 4159, 4177, 4201, 4211,  
595 4217, 4219, 4229, 4231, 4241, 4243, 4253, 4259, 4261, 4271, 4273, 4283, 4289, 4297, 4327, 4337,  
596 4339, 4349, 4357, 4363, 4373, 4391, 4397, 4409, 4421, 4423, 4441, 4447, 4451, 4457, 4463, 4481,  
597 4483, 4493, 4507, 4513, 4517, 4519, 4523, 4547, 4549, 4561, 4567, 4583, 4591, 4597, 4603, 4621,  
598 4637, 4639, 4643, 4649, 4651, 4657, 4663, 4673, 4679, 4691, 4703, 4721, 4723, 4729, 4733, 4751,  
599 4759, 4783, 4787, 4789, 4793, 4799, 4801, 4813, 4817, 4831, 4861, 4871, 4877, 4889, 4903, 4909,  
600 4919, 4931, 4933, 4937, 4943, 4951, 4957, 4967, 4969, 4973, 4987, 4993, 4999, 5003, 5009, 5011,  
601 5021, 5023, 5039, 5051, 5059, 5077, 5081, 5087, 5099, 5101, 5107, 5113, 5119, 5147, 5153, 5167,  
602 5171, 5179, 5189, 5197, 5209, 5227, 5231, 5233, 5237, 5261, 5273, 5279, 5281, 5297, 5303, 5309,  
603 5323, 5333, 5347, 5351, 5381, 5387, 5393, 5399, 5407, 5413, 5417, 5419, 5431, 5437, 5441, 5443,  
604 5449, 5471, 5477, 5479, 5483, 5501, 5503, 5507, 5519, 5521, 5527, 5531, 5557, 5563, 5569, 5573,  
605 5581, 5591, 5623, 5639, 5641, 5647, 5651, 5653, 5657, 5659, 5669, 5683, 5689, 5693, 5701, 5711,  
606 5717, 5737, 5741, 5743, 5749, 5779, 5783, 5791, 5801, 5807, 5813, 5821, 5827, 5839, 5843, 5849,  
607 5851, 5857, 5861, 5867, 5869, 5879, 5881, 5897, 5903, 5923, 5927, 5939, 5953, 5981, 5987, 6007,  
608 6011, 6029, 6037, 6043, 6047, 6053, 6067, 6073, 6079, 6089, 6091, 6101, 6113, 6121, 6131, 6133,  
609 6143, 6151, 6163, 6173, 6197, 6199, 6203, 6211, 6217, 6221, 6229, 6247, 6257, 6263, 6269, 6271,  
610 6277, 6287, 6299, 6301, 6311, 6317, 6323, 6329, 6337, 6343, 6353, 6359, 6361, 6367, 6373, 6379,  
611 6389, 6397, 6421, 6427, 6449, 6451, 6469, 6473, 6481, 6491, 6521, 6529, 6547, 6551, 6553, 6563,  
612 6569, 6571, 6577, 6581, 6599, 6607, 6619, 6637, 6653, 6659, 6661, 6673, 6679, 6689, 6691, 6701,  
613 6703, 6709, 6719, 6733, 6737, 6761, 6763, 6779, 6781, 6791, 6793, 6803, 6823, 6827, 6829, 6833,  
614 6841, 6857, 6863, 6869, 6871, 6883, 6899, 6907, 6911, 6917, 6947, 6949, 6959, 6961, 6967, 6971,  
615 6977, 6983, 6991, 6997, 7001, 7013, 7019, 7027, 7039, 7043, 7057, 7069, 7079, 7103, 7109, 7121,  
616 7127, 7129, 7151, 7159, 7177, 7187, 7193, 7207, 7211, 7213, 7219, 7229, 7237, 7243, 7247, 7253,  
617 7283, 7297, 7307, 7309, 7321, 7331, 7333, 7349, 7351, 7369, 7393, 7411, 7417, 7433, 7451, 7457,  
618 7459, 7477, 7481, 7487, 7489, 7499, 7507, 7517, 7523, 7529, 7537, 7541, 7547, 7549, 7559, 7561,  
619 7573, 7577, 7583, 7589, 7591, 7603, 7607, 7621, 7639, 7643, 7649, 7669, 7673, 7681, 7687, 7691,  
620 7699, 7703, 7717, 7723, 7727, 7741, 7753, 7757, 7759, 7789, 7793, 7817, 7823, 7829, 7841, 7853,  
621 7867, 7873, 7877, 7879, 7883, 7901, 7907, 7919, 7927, 7933, 7937, 7949, 7951, 7963, 7993, 8009,  
622 8011, 8017, 8039, 8053, 8059, 8069, 8081, 8087, 8089, 8093, 8101, 8111, 8117, 8123, 8147, 8161,  
623 8167, 8171, 8179, 8191, 8209, 8219, 8221, 8231, 8233, 8237, 8243, 8263, 8269, 8273, 8287, 8291,  
624 8293, 8297, 8311, 8317, 8329, 8353, 8363, 8369, 8377, 8387, 8389, 8419, 8423, 8429, 8431, 8443,  
625 8447, 8461, 8467, 8501, 8513, 8521, 8527, 8537, 8539, 8543, 8563, 8573, 8581, 8597, 8599, 8609,  
626 8623, 8627, 8629, 8641, 8647, 8663, 8669, 8677, 8681, 8689, 8693, 8699, 8707, 8713, 8719, 8731,  
627 8737, 8741, 8747, 8753, 8761, 8779, 8783, 8803, 8807, 8819, 8821, 8831, 8837, 8839, 8849, 8861,  
628 8863, 8867, 8887, 8893, 8923, 8929, 8933, 8941, 8951, 8963, 8969, 8971, 8999, 9001, 9007, 9011,  
629 9013, 9029, 9041, 9043, 9049, 9059, 9067, 9091, 9103, 9109, 9127, 9133, 9137, 9151, 9157, 9161,  
630 9173, 9181, 9187, 9199, 9203, 9209, 9221, 9227, 9239, 9241, 9257, 9277, 9281, 9283, 9293, 9311,

631 9319,9323,9337,9341,9343,9349,9371,9377,9391,9397,9403,9413,9419,9421,9431,9433,  
632 9437,9439,9461,9463,9467,9473,9479,9491,9497,9511,9521,9533,9539,9547,9551,9587,  
633 9601,9613,9619,9623,9629,9631,9643,9649,9661,9677,9679,9689,9697,9719,9721,9733,  
634 9739,9743,9749,9767,9769,9781,9787,9791,9803,9811,9817,9829,9833,9839,9851,9857,  
635 9859,9871,9883,9887,9901,9907,9923,9929,  
636 9931,9941,9949,9967,9973,10007,10009,10037,  
637 10039,10061,10067,10069,10079,10091,10093,10099,  
638 10103,10111,10133,10139,10141,10151,10159,10163,  
639 10169,10177,10181,10193,10211,10223,10243,10247,  
640 10253,10259,10267,10271,10273,10289,10301,10303,  
641 10313,10321,10331,10333,10337,10343,10357,10369,  
642 10391,10399,10427,10429,10433,10453,10457,10459,  
643 10463,10477,10487,10499,10501,10513,10529,10531,  
644 10559,10567,10589,10597,10601,10607,10613,10627,  
645 10631,10639,10651,10657,10663,10667,10687,10691,  
646 10709,10711,10723,10729,10733,10739,10753,10771,  
647 10781,10789,10799,10831,10837,10847,10853,10859,  
648 10861,10867,10883,10889,10891,10903,10909,10937,  
649 10939,10949,10957,10973,10979,10987,10993,11003,  
650 11027,11047,11057,11059,11069,11071,11083,11087,  
651 11093,11113,11117,11119,11131,11149,11159,11161,  
652 11171,11173,11177,11197,11213,11239,11243,11251,  
653 11257,11261,11273,11279,11287,11299,11311,11317,  
654 11321,11329,11351,11353,11369,11383,11393,11399,  
655 11411,11423,11437,11443,11447,11467,11471,11483,  
656 11489,11491,11497,11503,11519,11527,11549,11551,  
657 11579,11587,11593,11597,11617,11621,11633,11657,  
658 11677,11681,11689,11699,11701,11717,11719,11731,  
659 11743,11777,11779,11783,11789,11801,11807,11813,  
660 11821,11827,11831,11833,11839,11863,11867,11887,  
661 11897,11903,11909,11923,11927,11933,11939,11941,  
662 11953,11959,11969,11971,11981,11987,12007,12011,  
663 12037,12041,12043,12049,12071,12073,12097,12101,  
664 12107,12109,12113,12119,12143,12149,12157,12161,  
665 12163,12197,12203,12211,12227,12239,12241,12251,  
666 12253,12263,12269,12277,12281,12289,12301,12323,  
667 12329,12343,12347,12373,12377,12379,12391,12401,  
668 12409,12413,12421,12433,12437,12451,12457,12473,  
669 12479,12487,12491,12497,12503,12511,12517,12527,  
670 12539,12541,12547,12553,12569,12577,12583,12589,  
671 12601,12611,12613,12619,12637,12641,12647,12653,  
672 12659,12671,12689,12697,12703,12713,12721,12739,  
673 12743,12757,12763,12781,12791,12799,12809,12821,  
674 12823,12829,12841,12853,12889,12893,12899,12907,  
675 12911,12917,12919,12923,12941,12953,12959,12967,  
676 12973,12979,12983,13001,13003,13007,13009,13033,  
677 13037,13043,13049,13063,13093,13099,13103,13109,  
678 13121,13127,13147,13151,13159,13163,13171,13177,  
679 13183,13187,13217,13219,13229,13241,13249,13259,  
680 13267,13291,13297,13309,13313,13327,13331,13337,  
681 13339,13367,13381,13397,13399,13411,13417,13421,  
682 13441,13451,13457,13463,13469,13477,13487,13499,  
683 13513,13523,13537,13553,13567,13577,13591,13597,  
684 13613,13619,13627,13633,13649,13669,13679,13681,  
685 13687,13691,13693,13697,13709,13711,13721,13723,  
686 13729,13751,13757,13759,13763,13781,13789,13799,  
687 13807,13829,13831,13841,13859,13873,13877,13879,  
688 13883,13901,13903,13907,13913,13921,13931,13933,  
689 13963,13967,13997,13999,14009,14011,14029,14033,  
690 14051,14057,14071,14081,14083,14087,14107,14143,  
691 14149,14153,14159,14173,14177,14197,14207,14221,  
692 14243,14249,14251,14281,14293,14303,14321,14323,  
693 14327,14341,14347,14369,14387,14389,14401,14407,  
694 14411,14419,14423,14431,14437,14447,14449,14461,  
695 14479,14489,14503,14519,14533,14537,14543,14549,  
696 14551,14557,14561,14563,14591,14593,14621,14627,

```
697     14629,14633,14639,14653,14657,14669,14683,14699,  
698     14713,14717,14723,14731,14737,14741,14747,14753,  
699     14759,14767,14771,14779,14783,14797,14813,14821,  
700     14827,14831,14843,14851,14867,14869,14879,14887,  
701     14891,14897,14923,14929,14939,14947,14951,14957,  
702     14969,14983,15013,15017,15031,15053,15061,15073,  
703     15077,15083,15091,15101,15107,15121,15131,15137,  
704     15139,15149,15161,15173,15187,15193,15199,15217,  
705     15227,15233,15241,15259,15263,15269,15271,15277,  
706     15287,15289,15299,15307,15313,15319,15329,15331,  
707     15349,15359,15361,15373,15377,15383,15391,15401,  
708     15413,15427,15439,15443,15451,15461,15467,15473,  
709     15493,15497,15511,15527,15541,15551,15559,15569,  
710     15581,15583,15601,15607,15619,15629,15641,15643,  
711     15647,15649,15661,15667,15671,15679,15683,15727,  
712     15731,15733,15737,15739,15749,15761,15767,15773,  
713     15787,15791,15797,15803,15809,15817,15823,15859,  
714     15877,15881,15887,15889,15901,15907,15913,15919,  
715     15923,15937,15959,15971,15973,15991,16001,16007,  
716     16033,16057,16061,16063,16067,16069,16073,16087,  
717     16091,16097,16103,16111,16127,16139,16141,16183,  
718     16187,16189,16193,16217,16223,16229,16231,16249,  
719     16253,16267,16273,16301,16319,16333,16339,16349,  
720     16361,16363,16369,16381,16411,16417,16421,16427,  
721     16433,16447,16451,16453,16477,16481,16487,16493,  
722     16519,16529,16547,16553,16561,16567,16573,16603,  
723     16607,16619,16631,16633,16649,16651,16657,16661,  
724     16673,16691,16693,16699,16703,16729,16741,16747,  
725     16759,16763,16787,16811,16823,16829,16831,16843,  
726     16871,16879,16883,16889,16901,16903,16921,16927,  
727     16931,16937,16943,16963,16979,16981,16987,16993,  
728     17011,17021,17027,17029,17033,17041,17047,17053,  
729     17077,17093,17099,17107,17117,17123,17137,17159,  
730     17167,17183,17189,17191,17203,17207,17209,17231,  
731     17239,17257,17291,17293,17299,17317,17321,17327,  
732     17333,17341,17351,17359,17377,17383,17387,17389,  
733     17393,17401,17417,17419,17431,17443,17449,17467,  
734     17471,17477,17483,17489,17491,17497,17509,17519,  
735     17539,17551,17569,17573,17579,17581,17597,17599,  
736     17609,17623,17627,17657,17659,17669,17681,17683,  
737     17707,17713,17729,17737,17747,17749,17761,17783,  
738     17789,17791,17807,17827,17837,17839,17851,17863  
739 };  
740 #endif  
741 #endif
```

## B.10 Elliptic Curve Files

### B.10.1. CpriDataEcc.h

```

1  #ifndef      _CRYPTDATAECC_H_
2  #define      _CRYPTDATAECC_H_

```

Structure for the curve parameters. This is an analog to the TPMS\_ALGORITHM\_DETAIL\_ECC

```

3  typedef struct {
4      const TPM2B      *p;          // a prime number
5      const TPM2B      *a;          // linear coefficient
6      const TPM2B      *b;          // constant term
7      const TPM2B      *x;          // generator x coordinate
8      const TPM2B      *y;          // generator y coordinate
9      const TPM2B      *n;          // the order of the curve
10     const TPM2B      *h;          // cofactor
11 } ECC_CURVE_DATA;
12 typedef struct
13 {
14     TPM_ECC_CURVE      curveId;
15     UINT16              keySizeBits;
16     TPMT_KDF_SCHEME    kdf;
17     TPMT_ECC_SCHEME    sign;
18     const ECC_CURVE_DATA *curveData; // the address of the curve data
19 } ECC_CURVE;
20 extern const ECC_CURVE_DATA SM2_P256;
21 extern const ECC_CURVE_DATA NIST_P256;
22 extern const ECC_CURVE_DATA BN_P256;
23 extern const ECC_CURVE eccCurves[];
24 extern const UINT16 ECC_CURVE_COUNT;
25 #endif

```

### B.10.2. CpriDataEcc.c

#### B.10.2.1.1. Introduction

The curve parameters in this section replicate the information that is in the TCG Algorithm Registry. This curve data should be removed when the data in the registry is extracted into a data file (CryptDataEcc.c) and a header file (CryptDataEcc.h). The header file should be shared between CryptEcc.c and CypriUtil.c

NOTE: This file should be included by the Ecc module in the library.

#### B.10.2.1.2. NIST Prime 256-bit Curve

```

1  static const TPM2B_32_BYTE_VALUE NIST_P256_P = {32, {
2      0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x01,
3      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
4      0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
5      0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};
6  static const TPM2B_32_BYTE_VALUE NIST_P256_A = {32, {
7      0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x01,
8      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
9      0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
10     0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};
11 static const TPM2B_32_BYTE_VALUE NIST_P256_B = {32, {
12     0x5a,0xc6,0x35,0xd8,0xaa,0x3a,0x93,0xe7,
13     0xb3,0xeb,0xbd,0x55,0x76,0x98,0x86,0xbc,
14     0x65,0x1d,0x06,0xb0,0xcc,0x53,0xb0,0xf6,
15     0x3b,0xce,0x3c,0x3e,0x27,0xd2,0x60,0x4b}};

```

```

16 static const TPM2B_32_BYTE_VALUE NIST_P256_X = {32, {
17     0x6b,0x17,0xd1,0xf2,0xe1,0x2c,0x42,0x47,
18     0xf8,0xbc,0xe6,0xe5,0x63,0xa4,0x40,0xf2,
19     0x77,0x03,0x7d,0x81,0x2d,0xeb,0x33,0xa0,
20     0xf4,0xa1,0x39,0x45,0xd8,0x98,0xc2,0x96}};
21 static const TPM2B_32_BYTE_VALUE NIST_P256_Y = {32, {
22     0x4f,0xe3,0x42,0xe2,0xfe,0x1a,0x7f,0x9b,
23     0x8e,0xe7,0xeb,0x4a,0x7c,0x0f,0x9e,0x16,
24     0x2b,0xce,0x33,0x57,0x6b,0x31,0x5e,0xce,
25     0xcb,0xb6,0x40,0x68,0x37,0xbf,0x51,0xf5}};
26 static const TPM2B_32_BYTE_VALUE NIST_P256_N = {32, {
27     0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
28     0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
29     0xbc,0xe6,0xfa,0xad,0xa7,0x17,0x9e,0x84,
30     0xf3,0xb9,0xca,0xc2,0xfc,0x63,0x25,0x51}};
31 static const TPM2B_1_BYTE_VALUE NIST_P256_H = {1, {1}};
32 const ECC_CURVE_DATA NIST_P256 = {&NIST_P256_P.b, &NIST_P256_A.b, &NIST_P256_B.b,
33     &NIST_P256_X.b, &NIST_P256_Y.b, &NIST_P256_N.b,
34     &NIST_P256_H.b};

```

### B.10.2.1.3. BN Prime 256-bit Curve

```

35 static const TPM2B_32_BYTE_VALUE BN_P256_P = {32, {
36     0xff,0xff,0xff,0xff,0xff,0xfc,0xf0,0xcd,
37     0x46,0xe5,0xf2,0x5e,0xee,0x71,0xa4,0x9f,
38     0x0c,0xdc,0x65,0xfb,0x12,0x98,0x0a,0x82,
39     0xd3,0x29,0x2d,0xdb,0xae,0xd3,0x30,0x13}};
40 static const TPM2B_1_BYTE_VALUE BN_P256_A = {1, {0}};
41 static const TPM2B_1_BYTE_VALUE BN_P256_B = {1, {3}};
42 static const TPM2B_1_BYTE_VALUE BN_P256_X = {1, {1}};
43 static const TPM2B_1_BYTE_VALUE BN_P256_Y = {1, {2}};
44 static const TPM2B_32_BYTE_VALUE BN_P256_N = {32, {
45     0xff,0xff,0xff,0xff,0xff,0xfc,0xf0,0xcd,
46     0x46,0xe5,0xf2,0x5e,0xee,0x71,0xa4,0x9e,
47     0x0c,0xdc,0x65,0xfb,0x12,0x99,0x92,0x1a,
48     0xf6,0x2d,0x53,0x6c,0xd1,0x0b,0x50,0x0d}};
49 static const TPM2B_1_BYTE_VALUE BN_P256_H = {1, {1}};
50 const ECC_CURVE_DATA BN_P256 = {&BN_P256_P.b, &BN_P256_A.b, &BN_P256_B.b,
51     &BN_P256_X.b, &BN_P256_Y.b, &BN_P256_N.b,
52     &BN_P256_H.b};
53 #ifndef TPM_ECC_SM2_P256
54 #ifndef _SM2_SIGN_DEBUG

```

These are the actual values for SM2 curve

```

55 static const TPM2B_32_BYTE_VALUE SM2_P256_P = {32, {
56     0xff,0xff,0xff,0xfe,0xff,0xff,0xff,0xff,
57     0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
58     0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
59     0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};
60 static const TPM2B_32_BYTE_VALUE SM2_P256_A = {32, {
61     0xff,0xff,0xff,0xfe,0xff,0xff,0xff,0xff,
62     0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
63     0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
64     0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xfc}};
65 static const TPM2B_32_BYTE_VALUE SM2_P256_B = {32, {
66     0x28,0xe9,0xfa,0x9e,0x9d,0x9f,0x5e,0x34,
67     0x4d,0x5a,0x9e,0x4b,0xcf,0x65,0x09,0xa7,
68     0xf3,0x97,0x89,0xf5,0x15,0xab,0x8f,0x92,
69     0xDD,0xBC,0xBD,0x41,0x4D,0x94,0x0E,0x93}};
70 static const TPM2B_32_BYTE_VALUE SM2_P256_X = {32, {
71     0x32,0xC4,0xAE,0x2C,0x1F,0x19,0x81,0x19,
72     0x5F,0x99,0x04,0x46,0x6A,0x39,0xC9,0x94,
73     0x8F,0xE3,0x0B,0xBF,0xF2,0x66,0x0B,0xE1,

```

```

74         0x71,0x5A,0x45,0x89,0x33,0x4C,0x74,0xC7}};
75 static const TPM2B_32_BYTE_VALUE SM2_P256_Y = {32,{
76         0xBC,0x37,0x36,0xA2,0xF4,0xF6,0x77,0x9C,
77         0x59,0xBD,0xCE,0xE3,0x6B,0x69,0x21,0x53,
78         0xD0,0xA9,0x87,0x7C,0xC6,0x2A,0x47,0x40,
79         0x02,0xDF,0x32,0xE5,0x21,0x39,0xF0,0xA0}};
80 static const TPM2B_32_BYTE_VALUE SM2_P256_N = {32,{
81         0xFF,0xFF,0xFF,0xFE,0xFF,0xFF,0xFF,0xFF,
82         0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
83         0x72,0x03,0xDF,0x6B,0x21,0xC6,0x05,0x2B,
84         0x53,0xBB,0xF4,0x09,0x39,0xD5,0x41,0x23}};
85 #else // _SM2_SIGN_DEBUG

```

These are the values for debug of SM2 sign

```

86 static const TPM2B_32_BYTE_VALUE SM2_P256_P = {32,{
87         0x85,0x42,0xD6,0x9E,0x4C,0x04,0x4F,0x18,
88         0xE8,0xB9,0x24,0x35,0xBF,0x6F,0xF7,0xDE,
89         0x45,0x72,0x83,0x91,0x5C,0x45,0x51,0x7D,
90         0x72,0x2E,0xDB,0x8B,0x08,0xF1,0xDF,0xC3}};
91 static const TPM2B_32_BYTE_VALUE SM2_P256_A = {32,{
92         0x78,0x79,0x68,0xB4,0xFA,0x32,0xC3,0xFD,
93         0x24,0x17,0x84,0x2E,0x73,0xBB,0xFE,0xFF,
94         0x2F,0x3C,0x84,0x8B,0x68,0x31,0xD7,0xE0,
95         0xEC,0x65,0x22,0x8B,0x39,0x37,0xE4,0x98}};
96 static const TPM2B_32_BYTE_VALUE SM2_P256_B = {32,{
97         0x63,0xE4,0xC6,0xD3,0xB2,0x3B,0x0C,0x84,
98         0x9C,0xF8,0x42,0x41,0x48,0x4B,0xFE,0x48,
99         0xF6,0x1D,0x59,0xA5,0xB1,0x6B,0xA0,0x6E,
100        0x6E,0x12,0xD1,0xDA,0x27,0xC5,0x24,0x9A}};
101 static const TPM2B_32_BYTE_VALUE SM2_P256_X = {32,{
102        0x42,0x1D,0xEB,0xD6,0x1B,0x62,0xEA,0xB6,
103        0x74,0x64,0x34,0xEB,0xC3,0xCC,0x31,0x5E,
104        0x32,0x22,0x0B,0x3B,0xAD,0xD5,0x0B,0xDC,
105        0x4C,0x4E,0x6C,0x14,0x7F,0xED,0xD4,0x3D}};
106 static const TPM2B_32_BYTE_VALUE SM2_P256_Y = {32,{
107        0x06,0x80,0x51,0x2B,0xCB,0xB4,0x2C,0x07,
108        0xD4,0x73,0x49,0xD2,0x15,0x3B,0x70,0xC4,
109        0xE5,0xD7,0xFD,0xFC,0xBF,0xA3,0x6E,0xA1,
110        0xA8,0x58,0x41,0xB9,0xE4,0x6E,0x09,0xA2}};
111 static const TPM2B_32_BYTE_VALUE SM2_P256_N = {32,{
112        0x85,0x42,0xD6,0x9E,0x4C,0x04,0x4F,0x18,
113        0xE8,0xB9,0x24,0x35,0xBF,0x6F,0xF7,0xDD,
114        0x29,0x77,0x20,0x63,0x04,0x85,0x62,0x8D,
115        0x5A,0xE7,0x4E,0xE7,0xC3,0x2E,0x79,0xB7}};
116 #endif
117 static const TPM2B_1_BYTE_VALUE SM2_P256_H = {1, {1}};
118 const ECC_CURVE_DATA SM2_P256 = {&SM2_P256_P.b, &SM2_P256_A.b, &SM2_P256_B.b,
119         &SM2_P256_X.b, &SM2_P256_Y.b, &SM2_P256_N.b,
120         &SM2_P256_H.b};
121 #endif

```

Make sure that this table has algorithms in the same order as the `eccCurveValues[]` table in `CryptUtil.c`

```

122 const ECC_CURVE eccCurves[] =
123 {
124     {TPM_ECC_NIST_P256, // curveId
125     256, // key size in bits
126     {TPM_ALG_NULL, {TPM_ALG_NULL}}, // default KDF and hash
127     {TPM_ALG_NULL, {TPM_ALG_NULL}}, // default signing scheme and hash
128     &NIST_P256} // curve values
129 #ifdef TPM_ECC_SM2_P256
130     ,{TPM_ECC_SM2_P256,
131     256,
132     {TPM_ALG_NULL, {TPM_ALG_NULL}},

```

```

133     {TPM_ALG_NULL, {TPM_ALG_NULL}},
134     &SM2_P256}
135 #endif
136 #ifndef TPM_ALG_ECDSA
137     ,{TPM_ECC_BN_P256,
138       256,
139       {TPM_ALG_NULL, {TPM_ALG_NULL}},
140       {TPM_ALG_ECDSA, {TPM_ALG_NULL}},
141       &BN_P256}
142 #endif
143 };
144 const UINT16    ECC_CURVE_COUNT = sizeof(eccCurves) / sizeof(ECC_CURVE);

```

### B.10.3. CpriECC.c

```

1  /*(Copyright)
2  Microsoft Copyright 2009, 2010, 2011, 2012
3  Microsoft Confidential Contribution to a TCG Specification or Design Guide
4  under Article 15 of "The Bylaws of the Trusted Computing Group" as Amended
5  through March 20, 2003
6
7  */
8
9  /** Includes and Defines
10
11  #include    "CryptoEngine.h"
12
13  /** Functions
14
15  /***_cpri__EccStartup()
16  // This function is called at TPM Startup to initialize the crypto units.
17  //
18  // In this implementation, no initialization is performed at startup but a
19  // future version may initialize the self-test functions here.
20  BOOL
21  _cpri__EccStartup(
22  void
23  )
24  {
25  return TRUE;
26  }

```

#### B.10.3.1.1. \_cpri\_\_GetCurveIdByIndex()

This function returns the number of the *i*-th implemented curve. The normal use would be to call this function with *i* starting at 0. When the *i* is greater than or equal to the number of implemented curves, TPM\_ECC\_NONE is returned.

```

27  TPM_ECC_CURVE
28  _cpri__GetCurveIdByIndex(
29  UINT16    i
30  )
31  {
32  if(i >= ECC_CURVE_COUNT)
33  return TPM_ECC_NONE;
34  return eccCurves[i].curveId;
35  }
36  UINT32
37  _cpri__EccGetCurveCount(
38  void
39  )
40  {

```



```

41     return ECC_CURVE_COUNT;
42 }

```

### B.10.3.1.2. `_cpri__EccGetParametersByCurveId()`

This function returns a pointer to the curve data that is associated with the indicated *curveId*. If there is no curve with the indicated ID, the function returns NULL.

Return Value	Meaning
NULL	curve with the indicated TPM_ECC_CURVE value is not implemented
->	pointer to the curve data

```

43 const ECC_CURVE *
44 _cpri__EccGetParametersByCurveId(
45     TPM_ECC_CURVE    curveId    // IN: the curveID
46 )
47 {
48     int    i;
49     for(i = 0; i < ECC_CURVE_COUNT; i++)
50     {
51         if(eccCurves[i].curveId == curveId)
52             return &eccCurves[i];
53     }
54     return NULL;
55 }
56 static const ECC_CURVE_DATA *
57 GetCurveData(
58     TPM_ECC_CURVE    curveId    // IN: the curveID
59 )
60 {
61     const ECC_CURVE    *curve = _cpri__EccGetParametersByCurveId(curveId);
62     return curve->curveData;
63 }

```

### B.10.4. `Point2B()`

This function makes a TPMS\_ECC\_POINT from a BIGNUM EC\_POINT.

```

64 static BOOL
65 Point2B(
66     EC_GROUP    *group,    // IN: group for the point
67     TPMS_ECC_POINT *p,    // OUT: receives the converted point
68     EC_POINT    *ecP,    // IN: the point to convert
69     UINT16    size,    // IN: size of the coordinates
70     BN_CTX    *context    // IN: working context
71 )
72 {
73     BIGNUM    *bnX;
74     BIGNUM    *bnY;
75
76     BN_CTX_start(context);
77     bnX = BN_CTX_get(context);
78     bnY = BN_CTX_get(context);
79
80     if(    bnY == NULL
81
82         // Get the coordinate values
83         || EC_POINT_get_affine_coordinates_GFp(group, ecP, bnX, bnY, context) != 1
84
85         // Convert x
86         || (!BnTo2B(&p->x.b, bnX, size))

```

```

87
88     // Convert y
89     || (!BnTo2B(&p->y.b, bnY, size))
90     )
91     FAIL(FATAL_ERROR_INTERNAL);
92
93     BN_CTX_end(context);
94     return TRUE;
95 }

```

#### B.10.4.1.1. EccCurveInit()

This function initializes the *OpenSSL()* group definition structure

This function is only used within this file.

It is a fatal error if *groupContext* is not provided.

Return Value	Meaning
NULL	the TPM_ECC_CURVE is not valid
->	points to a structure in <i>groupContext</i> static EC_GROUP *

```

96 static EC_GROUP *
97 EccCurveInit(
98     TPM_ECC_CURVE      curveId,           // IN: the ID of the curve
99     BN_CTX              *groupContext     // IN: the context in which the
100                                     // group is to be created
101 )
102 {
103     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
104     EC_GROUP              *group = NULL;
105     EC_POINT              *P = NULL;
106     BN_CTX                *context;
107     BIGNUM                *bnP;
108     BIGNUM                *bnA;
109     BIGNUM                *bnB;
110     BIGNUM                *bnX;
111     BIGNUM                *bnY;
112     BIGNUM                *bnN;
113     BIGNUM                *bnH;
114     int                   ok = FALSE;
115
116     // Context must be provided and curve selector must be valid
117     pAssert(groupContext != NULL && curveData != NULL);
118
119     context = BN_CTX_new();
120     if(context == NULL)
121         FAIL(FATAL_ERROR_ALLOCATION);
122
123     BN_CTX_start(context);
124     bnP = BN_CTX_get(context);
125     bnA = BN_CTX_get(context);
126     bnB = BN_CTX_get(context);
127     bnX = BN_CTX_get(context);
128     bnY = BN_CTX_get(context);
129     bnN = BN_CTX_get(context);
130     bnH = BN_CTX_get(context);
131
132     if (bnH == NULL)
133         goto Cleanup;
134
135     // Convert the number formats
136

```

```

137
138     BnFrom2B(bnP, curveData->p);
139     BnFrom2B(bnA, curveData->a);
140     BnFrom2B(bnB, curveData->b);
141     BnFrom2B(bnX, curveData->x);
142     BnFrom2B(bnY, curveData->y);
143     BnFrom2B(bnN, curveData->n);
144     BnFrom2B(bnH, curveData->h);
145
146     // initialize EC group, associate a generator point and initialize the point
147     // from the parameter data
148     ok = ( (group = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, groupContext)) != NULL
149           && (P = EC_POINT_new(group)) != NULL
150           && EC_POINT_set_affine_coordinates_GFp(group, P, bnX, bnY, groupContext)
151           && EC_GROUP_set_generator(group, P, bnN, bnH)
152           );
153 Cleanup:
154     if (!ok && group != NULL)
155     {
156         EC_GROUP_free(group);
157         group = NULL;
158     }
159     if(P != NULL)
160         EC_POINT_free(P);
161     BN_CTX_end(context);
162     BN_CTX_free(context);
163     return group;
164 }

```

#### B.10.4.1.2. PointFrom2B()

This function sets the coordinates of an existing BN Point from a TPMS\_ECC\_POINT.

```

165 static EC_POINT *
166 PointFrom2B(
167     EC_GROUP      *group,           // IN: the group for the point
168     EC_POINT      *ecP,            // IN: an existing BN point in the group
169     TPMS_ECC_POINT *p,             // IN: the 2B coordinates of the point
170     BN_CTX        *context         // IN: the BIGNUM context
171 )
172 {
173     BIGNUM      *bnX;
174     BIGNUM      *bnY;
175
176     // If the point is not allocated then just return a NULL
177     if(ecP == NULL)
178         return NULL;
179
180     BN_CTX_start(context);
181     bnX = BN_CTX_get(context);
182     bnY = BN_CTX_get(context);
183     if( // Set the coordinates of the point
184         bnY == NULL
185         || BN_bin2bn(p->x.t.buffer, p->x.t.size, bnX) == NULL
186         || BN_bin2bn(p->y.t.buffer, p->y.t.size, bnY) == NULL
187         || !EC_POINT_set_affine_coordinates_GFp(group, ecP, bnX, bnY, context)
188     )
189         FAIL(FATAL_ERROR_INTERNAL);
190
191     BN_CTX_end(context);
192     return ecP;
193 }

```

**B.10.4.1.3. EccInitPoint2B()**

This function allocates a point in the provided group and initializes it with the values in a TPMS\_ECC\_POINT.

```

194 static EC_POINT *
195 EccInitPoint2B(
196     EC_GROUP      *group,          // IN: group for the point
197     TPMS_ECC_POINT *p,            // IN: the coordinates for the point
198     BN_CTX        *context        // IN: the BIGNUM context
199 )
200 {
201     EC_POINT      *ecP;
202
203     BN_CTX_start(context);
204     ecP = EC_POINT_new(group);
205
206     if(PointFrom2B(group, ecP, p, context) == NULL)
207         FAIL(FATAL_ERROR_INTERNAL);
208
209     BN_CTX_end(context);
210     return ecP;
211 }

```

**B.10.4.1.4. PointMul()**

This function does a point multiply and checks for the result being the point at infinity.  $Q = ([A]G + [B]P)$

Return Value	Meaning
CRYPT_NO_RESULT	point is at infinity
CRYPT_SUCCESS	point not at infinity

```

212 static CRYPT_RESULT
213 PointMul(
214     EC_GROUP      *group,          // IN: group curve
215     EC_POINT      *ecpQ,          // OUT: result
216     BIGNUM        *bnA,           // IN: scalar for [A]G
217     EC_POINT      *ecpP,          // IN: point for [B]P
218     BIGNUM        *bnB,           // IN: scalar for [B]P
219     BN_CTX        *context        // IN: working context
220 )
221 {
222     if(EC_POINT_mul(group, ecpQ, bnA, ecpP, bnB, context) != 1)
223         FAIL(FATAL_ERROR_INTERNAL);
224     if(EC_POINT_is_at_infinity(group, ecpQ))
225         return CRYPT_NO_RESULT;
226     return CRYPT_SUCCESS;
227 }

```

**B.10.4.1.5. GetRandomPrivate()**

This function gets a random value ( $d$ ) to use as a private ECC key and then qualifies the key so that it is between  $2^{(nLen/2)} \leq d < n$ .

It is a fatal error if  $dOut$  or  $pIn$  is not provided or if the size of  $pIn$  is larger than MAX\_ECC\_KEY\_BYTES (the largest buffer size of a TPM2B\_ECC\_PARAMETER)

```

228 static void
229 GetRandomPrivate(
230     TPM2B_ECC_PARAMETER *dOut,    // OUT: the qualified random value

```

```

231     const TPM2B          *pIn          // IN: the maximum value for the key
232 )
233 {
234     int                i;
235     BYTE              *pb;
236
237     pAssert(pIn != NULL && dOut != NULL && pIn->size <= MAX_ECC_KEY_BYTES);
238
239     // Set the size of the output
240     dOut->t.size = pIn->size;
241     // Get some random bits
242     while(TRUE)
243     {
244         _cpri__GenerateRandom(dOut->t.size, dOut->t.buffer);
245         if(memcmp(dOut->t.buffer, pIn->buffer, pIn->size) < 0)
246         {
247             // dIn is less than n so make sure that it is at least greater than
248             // 2^(nLen/2). That is, one of the bytes in the upper half of the
249             // value needs to be non-zero
250             for(pb = dOut->t.buffer, i = dOut->t.size/2; i > 0; i--)
251             {
252                 if(*pb++ != 0)
253                     return;
254             }
255         }
256     }
257 }

```

#### B.10.4.1.6. Mod2B()

Function does modular reduction of TPM2B values.

```

258 static CRYPT_RESULT
259 Mod2B(
260     TPM2B          *x,          // IN/OUT: value to reduce
261     const TPM2B    *n          // IN: mod
262 )
263 {
264     int            compare;
265     compare = _math__uComp(x->size, x->buffer, n->size, n->buffer);
266     if(compare < 0)
267         // if x < n, then mod is x
268         return CRYPT_SUCCESS;
269     if(compare == 0)
270     {
271         // if x == n then mod is 0
272         x->size = 0;
273         x->buffer[0] = 0;
274         return CRYPT_SUCCESS;
275     }
276     return _math__Div(x, n, NULL, x);
277 }

```

#### B.10.4.1.7. \_cpri\_\_EccPointMultiply

This function computes  $R := [dln]G + [uln]Qln$ . Where  $dln$  and  $uln$  are scalars,  $G$  and  $Qln$  are points on the specified curve and  $G$  is the default generator of the curve.

The  $xOut$  and  $yOut$  parameters are optional and may be set to NULL if not used.

It is not necessary to provide  $uln$  if  $Qln$  is specified but one of  $uln$  and  $dln$  must be provided. If  $dln$  and  $Qln$  are specified but  $uln$  is not provided, then  $R = [dln]Qln$ .

If the multiply produces the point at infinity, the CRYPT\_NO\_RESULT is returned.

The sizes of *xOut* and *yOut* will be set to be the size of the degree of the curve

It is a fatal error if *dIn* and *uIn* are both unspecified (NULL) or if *Qin* or *Rout* is unspecified.

Return Value	Meaning
CRYPT_SUCCESS	point multiplication succeeded
CRYPT_POINT	the point <i>Qin</i> is not on the curve
CRYPT_NO_RESULT	the product point is at infinity

```

278  CRYPT_RESULT
279  _cpri_EccPointMultiply(
280      TPMS_ECC_POINT      *Rout,           // OUT: the product point R
281      TPM_ECC_CURVE       curveId,        // IN: the curve to use
282      TPM2B_ECC_PARAMETER *dIn,           // IN: value to multiply against
283                                           // the curve generator
284      TPMS_ECC_POINT      *Qin,           // IN: point Q
285      TPM2B_ECC_PARAMETER *uIn,           // IN: scalar value for the multiplier
286                                           // of Q
287  )
288  {
289      BN_CTX                *context;
290      BIGNUM                *bnD;
291      BIGNUM                *bnU;
292      EC_GROUP              *group;
293      EC_POINT              *R = NULL;
294      EC_POINT              *Q = NULL;
295      CRYPT_RESULT          retVal = CRYPT_SUCCESS;
296
297
298      // Validate that the required parameters are provided.
299      pAssert((dIn != NULL || uIn != NULL) && (Qin != NULL || dIn != NULL));
300
301      // If a point is provided for the multiply, make sure that it is on the curve
302      if(Qin != NULL && !_cpri_EccIsPointOnCurve(curveId, Qin))
303          return CRYPT_POINT;
304
305      context = BN_CTX_new();
306      if(context == NULL)
307          FAIL(FATAL_ERROR_ALLOCATION);
308
309      BN_CTX_start(context);
310      bnU = BN_CTX_get(context);
311      bnD = BN_CTX_get(context);
312      group = EccCurveInit(curveId, context);
313
314      // There should be no path for getting a bad curve ID into this function.
315      pAssert(group != NULL);
316
317      // check allocations should have worked and allocate R
318      if( bnD == NULL
319          || (R = EC_POINT_new(group)) == NULL)
320          FAIL(FATAL_ERROR_ALLOCATION);
321
322      // If Qin is present, create the point
323      if(Qin != NULL)
324      {
325          // Assume the size variables do not overflow. This should not happen in
326          // the contexts in which this function will be called.
327          assert2Bsize(Qin->x.t);
328          assert2Bsize(Qin->y.t);
329          Q = EccInitPoint2B(group, Qin, context);

```

```

330     }
331     if(dIn != NULL)
332     {
333         // Assume the size variables do not overflow, which should not happen in
334         // the contexts that this function will be called.
335         assert2Bsize(dIn->t);
336         BnFrom2B(bnD, &dIn->b);
337     }
338     else
339         bnD = NULL;
340
341     // If uIn is specified, initialize its BIGNUM
342     if(uIn != NULL)
343     {
344         // Assume the size variables do not overflow, which should not happen in
345         // the contexts that this function will be called.
346         assert2Bsize(uIn->t);
347         BnFrom2B(bnU, &uIn->b);
348     }
349     // If uIn is not specified but Q is, then we are going to
350     // do R = [d]Q
351     else if(Qin != NULL)
352     {
353         bnU = bnD;
354         bnD = NULL;
355     }
356     // If neither Q nor u is specified, then null this pointer
357     else
358         bnU = NULL;
359
360     // Use the generator of the curve
361     if((retVal = PointMul(group, R, bnD, Q, bnU, context)) == CRYPT_SUCCESS)
362         Point2B(group, Rout, R, (UINT16) BN_num_bytes(&group->field), context);
363
364     if (Q)
365         EC_POINT_free(Q);
366     if(R)
367         EC_POINT_free(R);
368     if(group)
369         EC_GROUP_free(group);
370     BN_CTX_end(context);
371     BN_CTX_free(context);
372     return retVal;
373 }
374

```

#### B.10.4.1.8. ClearPoint2B()

Initialize the size values of a point

```

375 static void
376 ClearPoint2B(
377     TPMS_ECC_POINT *p           // IN: the point
378 )
379 {
380     if(p != NULL) {
381         p->x.t.size = 0;
382         p->y.t.size = 0;
383     }
384 }
385 #if defined TPM_ALG_ECDA || defined TPM_ALG_SM2 /*%

```

**B.10.4.1.9. `_cpri__EccCommitCompute()`**

This function performs the point multiply operations required by `TPM2_Commit()`.

If *B* or *M* is provided, they must be on the curve defined by *curveId*. This routine does not check that they are on the curve and results are unpredictable if they are not.

It is a fatal error if *r* or *d* is NULL. If *B* is not NULL, then it is a fatal error if *K* and *L* are both NULL. If *M* is not NULL, then it is a fatal error if *E* is NULL.

Return Value	Meaning
CRYPT_SUCCESS	computations completed normally
CRYPT_NO_RESULT	if <i>K</i> , <i>L</i> or <i>E</i> was computed to be the point at infinity
CRYPT_CANCEL	a cancel indication was asserted during this function

```

386  CRYPT_RESULT
387  _cpri__EccCommitCompute(
388      TPMS_ECC_POINT    *K,           // OUT: [d]B or [r]Q
389      TPMS_ECC_POINT    *L,           // OUT: [r]B
390      TPMS_ECC_POINT    *E,           // OUT: [r]M
391      TPM_ECC_CURVE     curveId,      // IN: the curve for the computations
392      TPMS_ECC_POINT    *M,           // IN: M (optional)
393      TPMS_ECC_POINT    *B,           // IN: B (optional)
394      TPM2B_ECC_PARAMETER *d,         // IN: d (required)
395      TPM2B_ECC_PARAMETER *r          // IN: the computed r value (required)
396  )
397  {
398      BN_CTX             *context;
399      BIGNUM             *bnX, *bnY, *bnR, *bnD;
400      EC_GROUP          *group;
401      EC_POINT          *pK = NULL, *pL = NULL, *pE = NULL, *pM = NULL, *pB = NULL;
402      UINT16            keySizeInBytes;
403      CRYPT_RESULT      retVal = CRYPT_SUCCESS;
404
405      // Validate that the required parameters are provided.
406      // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
407      // E := [r]Q if both M and B are NULL.
408      pAssert( r != NULL && (K != NULL || B == NULL) && (L != NULL || B == NULL)
409              || (E != NULL || (M == NULL && B != NULL)));
410
411      context = BN_CTX_new();
412      if(context == NULL)
413          FAIL(FATAL_ERROR_ALLOCATION);
414      BN_CTX_start(context);
415      bnR = BN_CTX_get(context);
416      bnD = BN_CTX_get(context);
417      bnX = BN_CTX_get(context);
418      bnY = BN_CTX_get(context);
419      if(bnY == NULL)
420          FAIL(FATAL_ERROR_ALLOCATION);
421
422      // Initialize the output points in case they are not computed
423      ClearPoint2B(K);
424      ClearPoint2B(L);
425      ClearPoint2B(E);
426
427      if((group = EccCurveInit(curveId, context)) == NULL)
428      {
429          retVal = CRYPT_PARAMETER;
430          goto Cleanup2;
431      }
432      keySizeInBytes = (UINT16) BN_num_bytes(&group->field);
433

```



```

434 // Sizes of the r and d parameters may not be zero
435 pAssert(((int) r->t.size > 0) && ((int) d->t.size > 0));
436
437 // Convert scalars to BIGNUM
438 BnFrom2B(bnR, &r->b);
439 BnFrom2B(bnD, &d->b);
440
441 // If B is provided, compute K=[d]B and L=[r]B
442 if(B != NULL)
443 {
444     // Allocate the points to receive the value
445     if( (pK = EC_POINT_new(group)) == NULL
446         || (pL = EC_POINT_new(group)) == NULL)
447         FAIL(FATAL_ERROR_ALLOCATION);
448     // need to compute K = [d]B
449     // Allocate and initialize BIGNUM version of B
450     pB = EccInitPoint2B(group, B, context);
451
452     // do the math for K = [d]B
453     if((retVal = PointMul(group, pK, NULL, pB, bnD, context)) != CRYPT_SUCCESS)
454         goto Cleanup;
455
456     // Convert BN K to TPM2B K
457     Point2B(group, K, pK, keySizeInBytes, context);
458
459     // compute L= [r]B after checking for cancel
460     if(_plat__IsCanceled())
461     {
462         retVal = CRYPT_CANCEL;
463         goto Cleanup;
464     }
465     // compute L = [r]B
466     if((retVal = PointMul(group, pL, NULL, pB, bnR, context)) != CRYPT_SUCCESS)
467         goto Cleanup;
468
469     // Convert BN L to TPM2B L
470     Point2B(group, L, pL, keySizeInBytes, context);
471 }
472 if(M != NULL || B == NULL)
473 {
474     // if this is the third point multiply, check for cancel first
475     if(B != NULL && _plat__IsCanceled())
476     {
477         retVal = CRYPT_CANCEL;
478         goto Cleanup;
479     }
480
481     // Allocate E
482     if((pE = EC_POINT_new(group)) == NULL)
483         FAIL(FATAL_ERROR_ALLOCATION);
484
485     // Create BIGNUM version of M unless M is NULL
486     if(M != NULL)
487     {
488         // M provided so initialize a BIGNUM M and compute E = [r]M
489         pM = EccInitPoint2B(group, M, context);
490         retVal = PointMul(group, pE, NULL, pM, bnR, context);
491     }
492     else
493         // compute E = [r]Q (this is only done if M and B are both NULL
494         retVal = PointMul(group, pE, bnR, NULL, NULL, context);
495
496     if(retVal == CRYPT_SUCCESS)
497         // Convert E to 2B format
498         Point2B(group, E, pE, keySizeInBytes, context);
499 }

```

```

500 Cleanup:
501     EC_GROUP_free(group);
502     if(pK != NULL) EC_POINT_free(pK);
503     if(pL != NULL) EC_POINT_free(pL);
504     if(pE != NULL) EC_POINT_free(pE);
505     if(pM != NULL) EC_POINT_free(pM);
506     if(pB != NULL) EC_POINT_free(pB);
507 Cleanup2:
508     BN_CTX_end(context);
509     BN_CTX_free(context);
510     return retVal;
511 }
512 #endif //%
```

#### B.10.4.1.10. `_cpri__EccIsPointOnCurve()`

This function is used to test if a point is on a defined curve. It does this by checking that  $y^2 \bmod p = x^3 + a*x + b \bmod p$

It is a fatal error if Q is not specified (is NULL).

Return Value	Meaning
TRUE	point is on curve
FALSE	point is not on curve or curve is not supported

```

513 BOOL
514 _cpri__EccIsPointOnCurve(
515     TPM_ECC_CURVE      curveId,           // IN: the curve selector
516     TPMS_ECC_POINT     *Q,               // IN: the point.
517 )
518 {
519     BN_CTX             *context;
520     BIGNUM              *bnX;
521     BIGNUM              *bnY;
522     BIGNUM              *bnA;
523     BIGNUM              *bnB;
524     BIGNUM              *bnP;
525     BIGNUM              *bn3;
526     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
527     BOOL                retVal;
528
529     pAssert(Q != NULL && curveData != NULL);
530
531     if((context = BN_CTX_new()) == NULL)
532         FAIL(FATAL_ERROR_ALLOCATION);
533     BN_CTX_start(context);
534     bnX = BN_CTX_get(context);
535     bnY = BN_CTX_get(context);
536     bnA = BN_CTX_get(context);
537     bnB = BN_CTX_get(context);
538     bn3 = BN_CTX_get(context);
539     bnP = BN_CTX_get(context);
540     if(bnP == NULL)
541         FAIL(FATAL_ERROR_ALLOCATION);
542
543     // Convert values
544     if ( !BN_bin2bn(Q->x.t.buffer, Q->x.t.size, bnX)
545         || !BN_bin2bn(Q->y.t.buffer, Q->y.t.size, bnY)
546         || !BN_bin2bn(curveData->p->buffer, curveData->p->size, bnP)
547         || !BN_bin2bn(curveData->a->buffer, curveData->a->size, bnA)
548         || !BN_set_word(bn3, 3)
549         || !BN_bin2bn(curveData->b->buffer, curveData->b->size, bnB)
```

```

550     )
551     FAIL(FATAL_ERROR_INTERNAL);
552
553
554     // The following sequence is probably not optimal but it seems to be correct.
555     // compute x^3 + a*x + b mod p
556     // first, compute a*x mod p
557     if( !BN_mod_mul(bnA, bnA, bnX, bnP, context)
558         // next, compute a*x + b mod p
559         || !BN_mod_add(bnA, bnA, bnB, bnP, context)
560         // next, compute X^3 mod p
561         || !BN_mod_exp(bnX, bnX, bn3, bnP, context)
562         // finally, compute x^3 + a*x + b mod p
563         || !BN_mod_add(bnX, bnX, bnA, bnP, context)
564         // then compute y^2
565         || !BN_mod_mul(bnY, bnY, bnY, bnP, context)
566     )
567     FAIL(FATAL_ERROR_INTERNAL);
568
569     retVal = BN_cmp(bnX, bnY) == 0;
570     BN_CTX_end(context);
571     BN_CTX_free(context);
572     return retVal;
573 }

```

#### B.10.4.1.11. `_cpri__GenerateKeyEcc()`

This function generates an ECC key pair based on the input parameters. This routine uses *KDFa()* to produce candidate numbers. The method is according to FIPS 186-3, section B. 4. 1 "GKey() Pair Generation Using Extra Random Bits." According to the method in FIPS 186-3, the resulting private value *d* should be  $1 \leq d < n$  where *n* is the order of the base point. In this implementation, the range of the private value is further restricted to be  $2^{(nLen/2)} \leq d < n$  where *nLen* is the order of *n*.

EXAMPLE: If the curve is NIST-P256, then *nLen* is 256 bits and *d* will need to be between  $2^{128} \leq d < n$

It is a fatal error if *Qout*, *dOut*, or *seed* is not provided (is NULL).

Return Value	Meaning
CRYPT_PARAMETER	the hash algorithm is not supported

```

574 CRYPT_RESULT
575 _cpri__GenerateKeyEcc(
576     TPM2B_ECC_POINT *Qout, // OUT: the public point
577     TPM2B_ECC_PARAMETER *dOut, // OUT: the private scalar
578     TPM_ECC_CURVE curveId, // IN: the curve identifier
579     TPM_ALG_ID hashAlg, // IN: hash algorithm to use in the key
580     // generation process
581     TPM2B *seed, // IN: the seed to use
582     const char *label, // IN: A label for the generation process.
583     TPM2B *extra, // IN: Party 1 data for the KDF
584     UINT32 *counter // IN/OUT: Counter value to allow KDF
585     iteration // to be propagated across multiple
586     functions
587 )
588 {
589     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
590     keySizeInBytes;
591     count = 0;
592     CRYPT_RESULT retVal;
593     lSize;
594     hLen = _cpri__GetDigestSize(hashAlg);

```

```

594     BIGNUM             *bnN;             // Order of the curve
595     BIGNUM             *bnD;             // the private scalar
596     BN_CTX             *context;         // the context for the BIGNUM values
597     BYTE               withExtra[MAX_ECC_KEY_BYTES + 8]; // trial key with
598                                     //extra bits
599     TPM2B_4_BYTE_VALUE marshaledCounter = {4, {0}};
600
601     // Validate parameters (these are fatal)
602     pAssert(seed != NULL && dOut != NULL && Qout != NULL && curveData != NULL);
603
604     // Non-fatal parameter checks.
605     if(hLen <= 0)
606         return CRYPT_PARAMETER;
607
608     // If there is a label, size it
609     if(label != NULL)
610         for(lSize = 0; label[lSize++]);
611
612     // allocate the local BN values
613     context = BN_CTX_new();
614     if(context == NULL)
615         FAIL(FATAL_ERROR_ALLOCATION);
616     BN_CTX_start(context);
617     bnN = BN_CTX_get(context);
618     bnD = BN_CTX_get(context);
619
620     // The size of the input scalars is limited by the size of the size of a
621     // TPM2B_ECC_PARAMETER. Make sure that it is not irrational.
622     pAssert((int) curveData->n->size <= MAX_ECC_KEY_BYTES);
623
624     if(    bnD == NULL
625        || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL
626        || (keySizeInBytes = (UINT16) BN_num_bytes(bnN)) > MAX_ECC_KEY_BYTES)
627         FAIL(FATAL_ERROR_INTERNAL);
628
629
630     // Initialize the count value
631     if(counter != NULL)
632         count = *counter;
633     if(count == 0)
634         count = 1;
635
636     // Start search for key (should be quick)
637     for(; count != 0; count++)
638     {
639
640         UINT32_TO_BYTE_ARRAY(count, marshaledCounter.t.buffer);
641         _cpri__KDFa(hashAlg, seed, label, extra, &marshaledCounter.b,
642                   BN_num_bits(bnN)+64, withExtra, NULL, FALSE);
643
644         // Convert the result and modular reduce
645         // Assume the size variables do not overflow, which should not happen in
646         // the contexts that this function will be called.
647         pAssert(keySizeInBytes <= MAX_ECC_KEY_BYTES);
648         if (    BN_bin2bn(withExtra, keySizeInBytes+8, bnD) == NULL
649            || BN_mod(bnD, bnD, bnN, context) != 1)
650             FAIL(FATAL_ERROR_INTERNAL);
651
652
653         // Make sure that the result is in the desired range
654         if(BN_num_bits(bnD) >= BN_num_bits(bnN)/2)
655         {
656             if(BnTo2B(&dOut->b, bnD, keySizeInBytes) != 1)
657                 FAIL(FATAL_ERROR_INTERNAL);
658
659             // Do the point multiply to create the public portion of the key. If

```

```

660         // the multiply generates the point at infinity (unlikely), do another
661         // iteration.
662         if((retVal = _cpri_EccPointMultiply(Qout, curveId, dOut, NULL, NULL))
663            != CRYPT_NO_RESULT)
664             break;
665     }
666 }
667
668 if(count == 0) // if counter wrapped, then the TPM should go into failure mode
669     FAIL(FATAL_ERROR_INTERNAL);
670
671
672 // Free up allocated BN values
673 BN_CTX_end(context);
674 BN_CTX_free(context);
675 if(counter != NULL)
676     *counter = count;
677 return retVal;
678 }

```

#### B.10.4.1.12. \_cpri\_\_GetEphemeralEcc()

This function creates an ephemeral ECC. It is ephemeral in that is expected that the private part of the key will be discarded

```

679 CRYPT_RESULT
680 _cpri_GetEphemeralEcc(
681     TPMS_ECC_POINT *Qout, // OUT: the public point
682     TPM2B_ECC_PARAMETER *dOut, // OUT: the private scalar
683     TPM_ECC_CURVE curveId // IN: the curve for the key
684 )
685 {
686     CRYPT_RESULT retVal;
687     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
688
689     pAssert(curveData != NULL);
690
691     // Keep getting random values until one is found that doesn't create a point
692     // at infinity. This will never, ever, ever, ever, happen but if it does
693     // we have to get a next random value.
694     while(TRUE)
695     {
696         GetRandomPrivate(dOut, curveData->p);
697
698         // _cpri_EccPointMultiply does not return CRYPT_ECC_POINT if no point is
699         // provided. CRYPT_PARAMETER should not be returned because the curve ID
700         // has to be supported. Thus the only possible error is CRYPT_NO_RESULT.
701         retVal = _cpri_EccPointMultiply(Qout, curveId, dOut, NULL, NULL);
702         if(retVal != CRYPT_NO_RESULT)
703             return retVal; // Will return CRYPT_SUCCESS
704     }
705 }
706 #ifdef TPM_ALG_ECDSA //%

```

#### B.10.4.1.13. SignEcdsa()

This function implements the ECDSA signing algorithm. The method is described in the comments below. It is a fatal error if *rOut*, *sOut*, *dIn*, or *digest* are not provided.

```

707 CRYPT_RESULT
708 SignEcdsa(
709     TPM2B_ECC_PARAMETER *rOut, // OUT: r component of the signature

```

```

710     TPM2B_ECC_PARAMETER    *sOut,        // OUT: s component of the signature
711     TPM_ECC_CURVE          curveId,      // IN: the curve used in the signature
712                               // process
713     TPM2B_ECC_PARAMETER    *dIn,        // IN: the private key
714     TPM2B                  *digest       // IN: the value to sign
715 )
716 {
717     BIGNUM                  *bnK;
718     BIGNUM                  *bnIk;
719     BIGNUM                  *bnN;
720     BIGNUM                  *bnR;
721     BIGNUM                  *bnD;
722     BIGNUM                  *bnZ;
723     TPM2B_ECC_PARAMETER    k;
724     TPMS_ECC_POINT         R;
725     BN_CTX                  *context;
726     CRYPT_RESULT           retVal = CRYPT_SUCCESS;
727     const ECC_CURVE_DATA   *curveData = GetCurveData(curveId);
728
729     pAssert(rOut != NULL && sOut != NULL && dIn != NULL && digest != NULL);
730
731     context = BN_CTX_new();
732     if(context == NULL)
733         FAIL(FATAL_ERROR_ALLOCATION);
734     BN_CTX_start(context);
735     bnN = BN_CTX_get(context);
736     bnZ = BN_CTX_get(context);
737     bnR = BN_CTX_get(context);
738     bnD = BN_CTX_get(context);
739     bnIk = BN_CTX_get(context);
740     bnK = BN_CTX_get(context);
741     // Assume the size variables do not overflow, which should not happen in
742     // the contexts that this function will be called.
743     pAssert(curveData->n->size <= MAX_ECC_PARAMETER_BYTES);
744     if( bnK == NULL
745         || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
746         FAIL(FATAL_ERROR_INTERNAL);
747
748     // The algorithm as described in "Suite B Implementer's Guide to FIPS 186-3(ECDSA)"
749     // 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a per-message
750     //    secret number and its inverse modulo n. Since n is prime, the
751     //    output will be invalid only if there is a failure in the RBG.
752     // 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
753     //    multiplication (see [Routines]), where G is the base point included in
754     //    the set of domain parameters.
755     // 3. Compute r = xR mod n. If r = 0, then return to Step 1. 1.
756     // 4. Use the selected hash function to compute H = Hash(M).
757     // 5. Convert the bit string H to an integer e as described in Appendix B.2.
758     // 6. Compute s = (k^-1 * (e + d * r)) mod n. If s = 0, return to Step 1.2.
759     // 7. Return (r, s).
760
761     // Generate a random value k in the range 1 <= k < n
762     // Want a K value that is the same size as the curve order
763     k.t.size = curveData->n->size;
764
765     while(TRUE) // This implements the loop at step 6. If s is zero, start over.
766     {
767         while(TRUE)
768         {
769             // Step 1 and 2 -- generate an ephemeral key and the modular inverse
770             // of the private key.
771             while(TRUE)
772             {
773                 GetRandomPrivate(&k, curveData->n);
774
775                 // Do the point multiply to generate a point and check to see if

```

```

776         // the point it at infinity
777         if(      _cpri_EccPointMultiply(&R, curveId, &k, NULL, NULL)
778             != CRYPT_NO_RESULT)
779             break; // can only be CRYPT_SUCCESS
780     }
781
782     // x coordinate is mod p. Make it mod n
783     // Assume the size variables do not overflow, which should not happen
784     // in the contexts that this function will be called.
785     assert2Bsize(R.x.t);
786     BN_bin2bn(R.x.t.buffer, R.x.t.size, bnR);
787     BN_mod(bnR, bnR, bnN, context);
788
789     // Make sure that it is not zero;
790     if(BN_is_zero(bnR))
791         continue;
792
793     // Make sure that a modular inverse exists
794     // Assume the size variables do not overflow, which should not happen
795     // in the contexts that this function will be called.
796     assert2Bsize(k.t);
797     BN_bin2bn(k.t.buffer, k.t.size, bnK);
798     if( BN_mod_inverse(bnIk, bnK, bnN, context) != NULL)
799         break;
800 }
801
802
803 // Set z = leftmost bits of the digest
804 // NOTE: This is implemented such that the key size needs to be
805 //       an even number of bytes in length.
806 if(digest->size > curveData->n->size)
807 {
808     // Assume the size variables do not overflow, which should not happen
809     // in the contexts that this function will be called.
810     pAssert(curveData->n->size <= MAX_ECC_KEY_BYTES);
811     // digest is larger than n so truncate
812     BN_bin2bn(digest->buffer, curveData->n->size, bnZ);
813 }
814 else
815 {
816     // Assume the size variables do not overflow, which should not happen
817     // in the contexts that this function will be called.
818     pAssert(digest->size <= MAX_DIGEST_SIZE);
819     // digest is same or smaller than n so use it all
820     BN_bin2bn(digest->buffer, digest->size, bnZ);
821 }
822
823 // Assume the size variables do not overflow, which should not happen in
824 // the contexts that this function will be called.
825 assert2Bsize(dIn->t);
826 if( bnZ == NULL
827
828     // need the private scalar of the signing key
829     || BN_bin2bn(dIn->t.buffer, dIn->t.size, bnD) == NULL)
830     FAIL(FATAL_ERROR_INTERNAL);
831
832
833 // NOTE: When the result of an operation is going to be reduced mod x
834 // any modular multiplication is done so that the intermediate values
835 // don't get too large.
836 //
837 // now have inverse of K (bnIk), z (bnZ), r (bnR), d (bnD) and n (bnN)
838 // Compute s = k^-1 (z + r*d) (mod n)
839 // first do d = r*d mod n
840 if( !BN_mod_mul(bnD, bnR, bnD, bnN, context)
841

```

```

842         // d = z + r * d
843         || !BN_add(bnD, bnZ, bnD)
844
845         // d = k^(-1)(z + r * d) (mod n)
846         || !BN_mod_mul(bnD, bnIk, bnD, bnN, context)
847
848         // convert to TPM2B format
849         || !BnTo2B(&sOut->b, bnD, curveData->n->size)
850
851         // and write the modular reduced version of r
852         // NOTE: this was deferred to reduce the number of
853         // error checks.
854         || !BnTo2B(&rOut->b, bnR, curveData->n->size)
855         FAIL(FATAL_ERROR_INTERNAL);
856
857     if(!BN_is_zero(bnD))
858         break; // signature not zero so done
859
860     // if the signature value was zero, start over
861 }
862
863 // Free up allocated BN values
864 BN_CTX_end(context);
865 BN_CTX_free(context);
866 return retVal;
867 }
868 #endif // %
869 #if defined TPM_ALG_ECDAE || defined TPM_ALG_ECSCHEINORR // %

```

#### B.10.4.1.14. EcDaa()

This function is used to perform a modified Schnorr signature for ECDAE.

This function performs  $s = k + T * d \pmod n$  where

- 'k' is a random, or pseudo-random value used in the commit phase
- T is the digest to be signed, and
- d is a private key.

If *tIn* is NULL then use *tOut* as T

Return Value	Meaning
CRYPT_SUCCESS	signature created

```

870 static CRYPT_RESULT
871 EcDaa(
872     TPM2B_ECC_PARAMETER *tOut, // OUT: T component of the signature
873     TPM2B_ECC_PARAMETER *sOut, // OUT: s component of the signature
874     TPM_ECC_CURVE curveId, // IN: the curve used in signing
875     TPM2B_ECC_PARAMETER *dIn, // IN: the private key
876     TPM2B *tIn, // IN: the value to sign
877     TPM2B_ECC_PARAMETER *kIn // IN: a random value from commit
878 )
879 {
880     BIGNUM *bnN, *bnK, *bnT, *bnD;
881     BN_CTX *context;
882     const TPM2B *n;
883     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
884     BOOL OK = TRUE;
885
886     // Parameter checks
887     pAssert( sOut != NULL && dIn != NULL && tOut != NULL

```



```

888         && kIn != NULL && curveData != NULL);
889
890     // this just saves key strokes
891     n = curveData->n;
892
893     if(tIn != NULL)
894         Copy2B(&tOut->b, tIn);
895
896     // The size of dIn and kIn input scalars is limited by the size of the size
897     // of a TPM2B_ECC_PARAMETER and tIn can be no larger than a digest.
898     // Make sure they are within range.
899     pAssert( (int) dIn->t.size <= MAX_ECC_KEY_BYTES
900             && (int) kIn->t.size <= MAX_ECC_KEY_BYTES
901             && (int) tOut->t.size <= MAX_DIGEST_SIZE
902             );
903
904     context = BN_CTX_new();
905     if(context == NULL)
906         FAIL(FATAL_ERROR_ALLOCATION);
907     BN_CTX_start(context);
908     bnN = BN_CTX_get(context);
909     bnK = BN_CTX_get(context);
910     bnT = BN_CTX_get(context);
911     bnD = BN_CTX_get(context);
912
913     // Check for allocation problems
914     if(bnD == NULL)
915         FAIL(FATAL_ERROR_ALLOCATION);
916
917     // Convert values
918     if( BN_bin2bn(n->buffer, n->size, bnN) == NULL
919        || BN_bin2bn(kIn->t.buffer, kIn->t.size, bnK) == NULL
920        || BN_bin2bn(dIn->t.buffer, dIn->t.size, bnD) == NULL
921        || BN_bin2bn(tOut->t.buffer, tOut->t.size, bnT) == NULL)
922
923         FAIL(FATAL_ERROR_INTERNAL);
924     // Compute T = T mod n
925     OK = OK && BN_mod(bnT, bnT, bnN, context);
926
927     // compute (s = k + T * d mod n)
928     //   d = T * d mod n
929     OK = OK && BN_mod_mul(bnD, bnT, bnD, bnN, context) == 1;
930     //   d = k + T * d mod n
931     OK = OK && BN_mod_add(bnD, bnK, bnD, bnN, context) == 1;
932     //   s = d
933     OK = OK && BnTo2B(&sOut->b, bnD, n->size);
934     //   r = T
935     OK = OK && BnTo2B(&tOut->b, bnT, n->size);
936     if(!OK)
937         FAIL(FATAL_ERROR_INTERNAL);
938
939     // Cleanup
940     BN_CTX_end(context);
941     BN_CTX_free(context);
942
943     return CRYPT_SUCCESS;
944 }
945 #endif //%
946 #ifdef TPM_ALG_EC Schnorr //%

```

#### B.10.4.1.15. SchnorrEcc()

This function is used to perform a modified Schnorr signature.

This function will generate a random value k and compute

- d)  $(xR, yR) = [k]G$   
 e)  $r = \text{hash}(P || xR) \pmod n$   
 f)  $s = k + r * ds$   
 g) return the tuple T, s

Return Value	Meaning
CRYPT_SUCCESS	signature created
CRYPT_SCHEME	<i>hashAlg</i> can't produce zero-length digest

```

947 static CRYPT_RESULT
948 SchnorrEcc (
949     TPM2B_ECC_PARAMETER *rOut,      // OUT: r component of the signature
950     TPM2B_ECC_PARAMETER *sOut,      // OUT: s component of the signature
951     TPM_ALG_ID           hashAlg,    // IN: hash algorithm used
952     TPM_ECC_CURVE        curveId,    // IN: the curve used in signing
953     TPM2B_ECC_PARAMETER *dIn,       // IN: the private key
954     TPM2B                 *digest,   // IN: the digest to sign
955     TPM2B_ECC_PARAMETER *kIn        // IN: for testing
956 )
957 {
958     TPM2B_ECC_PARAMETER k;
959     BIGNUM               *bnR, *bnN, *bnK, *bnT, *bnD;
960     BN_CTX               *context;
961     const TPM2B          *n;
962     EC_POINT             *pR = NULL;
963     EC_GROUP             *group = NULL;
964     CPRI_HASH_STATE     hashState;
965     UINT16               digestSize = _cpri_GetDigestSize(hashAlg);
966     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
967     TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_PARAMETER_BYTES));
968     TPM2B_T              T2b;
969     BOOL                 OK = TRUE;
970
971
972     // Parameter checks
973
974     // Must have a place for the 'r' and 's' parts of the signature, a private
975     // key ('d')
976     pAssert( rOut != NULL && sOut != NULL && dIn != NULL
977             && digest != NULL && curveData != NULL);
978
979     // to save key strokes
980     n = curveData->n;
981
982     // If the digest does not produce a hash, then null the signature and return
983     // a failure.
984     if(digestSize == 0)
985     {
986         rOut->t.size = 0;
987         sOut->t.size = 0;
988         return CRYPT_SCHEME;
989     }
990
991     // Allocate big number values
992     context = BN_CTX_new();
993     if(context == NULL)
994         FAIL(FATAL_ERROR_ALLOCATION);
995     BN_CTX_start(context);
996     bnR = BN_CTX_get(context);
997     bnN = BN_CTX_get(context);
998     bnK = BN_CTX_get(context);

```

```

999     bnT = BN_CTX_get(context);
1000    bnD = BN_CTX_get(context);
1001    if(   bnD == NULL
1002        // initialize the group parameters
1003        || (group = EccCurveInit(curveId, context)) == NULL
1004        // allocate a local point
1005        || (pR = EC_POINT_new(group)) == NULL
1006        )
1007        FAIL(FATAL_ERROR_ALLOCATION);
1008
1009    if(BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
1010        FAIL(FATAL_ERROR_INTERNAL);
1011
1012    while(OK)
1013    {
1014    // a) set k to a random value such that  $1 \leq k \leq n-1$ 
1015        if(kIn != NULL)
1016        {
1017            Copy2B(&k.b, &kIn->b); // copy input k if testing
1018            OK = FALSE;           // not OK to loop
1019        }
1020        else
1021            // If get a random value in the correct range
1022            GetRandomPrivate(&k, n);
1023
1024            // Convert 'k' and generate pR = [k]G
1025            BnFrom2B(bnK, &k.b);
1026
1027    // b) compute E := (xE, yE) := [k]G
1028            if(PointMul(group, pR, bnK, NULL, NULL, context) == CRYPT_NO_RESULT)
1029    // c) if E is the point at infinity, go to a)
1030                continue;
1031
1032    // d) compute e := xE (mod n)
1033            // Get the x coordinate of the point
1034            EC_POINT_get_affine_coordinates_GFp(group, pR, bnR, NULL, context);
1035
1036            // make (mod n)
1037            BN_mod(bnR, bnR, bnN, context);
1038
1039    // e) if e is zero, go to a)
1040            if(BN_is_zero(bnR))
1041                continue;
1042
1043            // Convert xR to a string (use T as a temp)
1044            BnTo2B(&T2b.b, bnR, (UINT16) (BN_num_bits(bnR)+7)/8);
1045
1046    // f) compute r := HschemeHash(P || e) (mod n)
1047            _cpri__StartHash(hashAlg, FALSE, &hashState);
1048            _cpri__UpdateHash(&hashState, digest->size, digest->buffer);
1049            _cpri__UpdateHash(&hashState, T2b.t.size, T2b.t.buffer);
1050            if(_cpri__CompleteHash(&hashState, digestSize, T2b.b.buffer) != digestSize)
1051                FAIL(FATAL_ERROR_INTERNAL);
1052            T2b.t.size = digestSize;
1053            BnFrom2B(bnT, &T2b.b);
1054            BN_div(NULL, bnT, bnT, bnN, context);
1055            BnTo2B(&rOut->b, bnT, (UINT16)BN_num_bytes(bnT));
1056
1057            // We have a value and we are going to exit the loop successfully
1058            OK = TRUE;
1059            break;
1060        }
1061    // Cleanup
1062    EC_POINT_free(pR);
1063    EC_GROUP_free(group);
1064    BN_CTX_end(context);

```

```

1065     BN_CTX_free(context);
1066
1067     // If we have a value, finish the signature
1068     if(OK)
1069         return EcDaa(rOut, sOut, curveId, dIn, NULL, &k);
1070     else
1071         return CRYPT_NO_RESULT;
1072 }
1073 #endif //%
1074 #ifndef TPM_ALG_SM2 //%
1075 #ifndef _SM2_SIGN_DEBUG //%
1076 static int
1077 cmp_bn2hex(
1078     BIGNUM      *bn,          //IN: big number value
1079     const char  *c           //IN: character string number
1080 )
1081 {
1082     int          result;
1083     BIGNUM      *bnC = BN_new();
1084     pAssert(bnC != NULL);
1085
1086     BN_hex2bn(&bnC, c);
1087     result = BN_ucmp(bn, bnC);
1088     BN_free(bnC);
1089     return result;
1090 }
1091 static int
1092 cmp_2B2hex(
1093     TPM2B      *a,          // IN: TPM2B number to compare
1094     const char  *c           // IN: character string
1095 )
1096 {
1097     int          result;
1098     int          sl = strlen(c);
1099     BIGNUM      *bnA;
1100
1101     result = (a->size * 2) - sl;
1102     if(result != 0)
1103         return result;
1104     pAssert((bnA = BN_bin2bn(a->buffer, a->size, NULL)) != NULL);
1105     result = cmp_bn2hex(bnA, c);
1106     BN_free(bnA);
1107     return result;
1108 }
1109 static void
1110 cpy_hexTo2B(
1111     TPM2B      *b,          // OUT: receives value
1112     const char  *c           // IN: source string
1113 )
1114 {
1115     BIGNUM      *bnB = BN_new();
1116     pAssert((strlen(c) & 1) == 0); // must have an even number of digits
1117     b->size = strlen(c) / 2;
1118     BN_hex2bn(&bnB, c);
1119     pAssert(bnB != NULL);
1120     BnTo2B(b, bnB, b->size);
1121     BN_free(bnB);
1122 }
1123 }
1124 #endif //% _SM2_SIGN_DEBUG

```

**B.10.4.1.16. SignSM2()**

This function signs a digest using the method defined in SM2 Part 2. The method in the standard will add a header to the message to be signed that is a hash of the values that define the key. This then hashed with the message to produce a digest (e) that is signed. This function signs e.

Return Value	Meaning
CRYPT_SUCCESS	sign worked

```

1125 static CRYPT_RESULT
1126 SignSM2(
1127     TPM2B_ECC_PARAMETER *rOut,      // OUT: r component of the signature
1128     TPM2B_ECC_PARAMETER *sOut,      // OUT: s component of the signature
1129     TPM_ECC_CURVE        curveId,    // IN: the curve used in signing
1130     TPM2B_ECC_PARAMETER *dIn,       // IN: the private key
1131     TPM2B                *digest     // IN: the digest to sign
1132 )
1133 {
1134     BIGNUM                *bnR;
1135     BIGNUM                *bnS;
1136     BIGNUM                *bnN;
1137     BIGNUM                *bnK;
1138     BIGNUM                *bnX1;
1139     BIGNUM                *bnD;
1140     BIGNUM                *bnT;      // temp
1141     BIGNUM                *bnE;
1142
1143     BN_CTX                *context;
1144     TPM2B_TYPE(DIGEST, MAX_DIGEST_SIZE);
1145     TPM2B_ECC_PARAMETER    k;
1146     TPMS_ECC_POINT         p2Br;
1147     const ECC_CURVE_DATA  *curveData = GetCurveData(curveId);
1148
1149     pAssert(curveData != NULL);
1150     context = BN_CTX_new();
1151     BN_CTX_start(context);
1152     bnK = BN_CTX_get(context);
1153     bnR = BN_CTX_get(context);
1154     bnS = BN_CTX_get(context);
1155     bnX1 = BN_CTX_get(context);
1156     bnN = BN_CTX_get(context);
1157     bnD = BN_CTX_get(context);
1158     bnT = BN_CTX_get(context);
1159     bnE = BN_CTX_get(context);
1160     if(bnE == NULL)
1161         FAIL(FATAL_ERROR_ALLOCATION);
1162
1163     BnFrom2B(bnE, digest);
1164     BnFrom2B(bnN, curveData->n);
1165     BnFrom2B(bnD, &dIn->b);
1166
1167     #ifdef _SM2_SIGN_DEBUG
1168     BN_hex2bn(&bnE, "B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9E7FE76");
1169     BN_hex2bn(&bnD, "128B2FA8BD433C6C068C8D803DFF79792A519A55171B1B650C23661D15897263");
1170     #endif
1171     // A3: Use random number generator to generate random number 1 <= k <= n-1;
1172     // NOTE: Ax: numbers are from the SM2 standard
1173     k.t.size = curveData->n->size;
1174     loop:
1175     {
1176         // Get a random number
1177         _cpri_GenerateRandom(k.t.size, k.t.buffer);
1178     }
1179     #ifdef _SM2_SIGN_DEBUG

```

```

1180 BN_hex2bn(&bnK, "6CB28D99385C175C94F94E934817663FC176D925DD72B727260DBAAE1FB2F96F");
1181 BnTo2B(&k.b, bnK, 32);
1182 k.t.size = 32;
1183 #endif
1184 //make sure that the number is 0 < k < n
1185 BnFrom2B(bnK, &k.b);
1186 if( BN_ucmp(bnK, bnN) >= 0
1187    || BN_is_zero(bnK)
1188    goto loop;
1189
1190 // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
1191 // to details specified in 4.2.7 in Part 1 of this document, transform the
1192 // data type of x1 into an integer;
1193 if( _cpri_EccPointMultiply(&p2Br, curveId, &k, NULL, NULL)
1194    == CRYPT_NO_RESULT)
1195    goto loop;
1196
1197 BnFrom2B(bnX1, &p2Br.x.b);
1198
1199 // A5: Figure out r = (e + x1) mod n,
1200 if(!BN_mod_add(bnR, bnE, bnX1, bnN, context)
1201    FAIL(FATAL_ERROR_INTERNAL);
1202 #ifdef _SM2_SIGN_DEBUG
1203 pAssert(cmp_bn2hex(bnR,
1204    "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1")
1205    == 0);
1206 #endif
1207
1208 // if r=0 or r+k=n, return to A3;
1209 if(!BN_add(bnT, bnK, bnR)
1210    FAIL(FATAL_ERROR_INTERNAL);
1211
1212 if(BN_is_zero(bnR) || BN_ucmp(bnT, bnN) == 0)
1213    goto loop;
1214
1215 // A6: Figure out s = ((1 + dA)^-1 * (k - r * dA)) mod n, if s=0, return to A3;
1216 // compute t = (1+d)-1
1217 BN_copy(bnT, bnD);
1218 if( !BN_add_word(bnT, 1)
1219    || !BN_mod_inverse(bnT, bnT, bnN, context) // (1 + dA)^-1 mod n
1220    )
1221    FAIL(FATAL_ERROR_INTERNAL);
1222 #ifdef _SM2_SIGN_DEBUG
1223 pAssert(cmp_bn2hex(bnT,
1224    "79BFCF3052C80DA7B939E0C6914A18CBB2D96D8555256E83122743A7D4F5F956")
1225    == 0);
1226 #endif
1227 // compute s = t * (k - r * dA) mod n
1228 if( !BN_mod_mul(bnS, bnD, bnR, bnN, context) // (r * dA) mod n
1229    || !BN_mod_sub(bnS, bnK, bnS, bnN, context) // (k - (r * dA) mod n
1230    || !BN_mod_mul(bnS, bnT, bnS, bnN, context)) // t * (k - (r * dA) mod n
1231    FAIL(FATAL_ERROR_INTERNAL);
1232 #ifdef _SM2_SIGN_DEBUG
1233 pAssert(cmp_bn2hex(bnS,
1234    "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7")
1235    == 0);
1236 #endif
1237
1238 if(BN_is_zero(bnS))
1239    goto loop;
1240 }
1241
1242 // A7: According to details specified in 4.2.1 in Part 1 of this document, transform
1243 // the data type of r, s into bit strings, signature of message M is (r, s).
1244
1245 BnTo2B(&rOut->b, bnR, curveData->n->size);

```

```

1246     BnTo2B(&sOut->b, bnS, curveData->n->size);
1247 #ifndef SM2_SIGN_DEBUG
1248 pAssert(cmp_2B2hex(&rOut->b,
1249     "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1")
1250     == 0);
1251 pAssert(cmp_2B2hex(&sOut->b,
1252     "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7")
1253     == 0);
1254 #endif
1255     BN_CTX_end(context);
1256     BN_CTX_free(context);
1257     return CRYPT_SUCCESS;
1258 }
1259 #endif // % TMP_ALG_SM2

```

#### B.10.4.1.17. \_cpri\_\_SignEcc()

This function is the dispatch function for the various ECC-based signing schemes.

Return Value	Meaning
CRYPT_SCHEME	<i>scheme</i> is not supported

```

1260 CRYPT_RESULT
1261 _cpri__SignEcc(
1262     TPM2B_ECC_PARAMETER *rOut,           // OUT: r component of the signature
1263     TPM2B_ECC_PARAMETER *sOut,           // OUT: s component of the signature
1264     TPM_ALG_ID           scheme,          // IN: the scheme selector
1265     TPM_ALG_ID           hashAlg,         // IN: the hash algorithm if need
1266     TPM_ECC_CURVE        curveId,        // IN: the curve used in the signature process
1267     TPM2B_ECC_PARAMETER *dIn,           // IN: the private key
1268     TPM2B                 *digest,       // IN: the digest to sign
1269     TPM2B_ECC_PARAMETER *kIn            // IN: k for input
1270 )
1271 {
1272     switch (scheme)
1273     {
1274         case TPM_ALG_ECDSA:
1275             // SignEcdsa always works
1276             return SignEcdsa(rOut, sOut, curveId, dIn, digest);
1277             break;
1278 #ifndef TPM_ALG_ECDA
1279         case TPM_ALG_ECDA:
1280             if(rOut != NULL)
1281                 rOut->b.size = 0;
1282             return EcDaa(rOut, sOut, curveId, dIn, digest, kIn);
1283             break;
1284 #endif
1285 #ifndef TPM_ALG_EC Schnorr
1286         case TPM_ALG_EC Schnorr:
1287             return SchnorrEcc(rOut, sOut, hashAlg, curveId, dIn, digest, kIn);
1288             break;
1289 #endif
1290 #ifndef TPM_ALG_SM2
1291         case TPM_ALG_SM2:
1292             return SignSM2(rOut, sOut, curveId, dIn, digest);
1293             break;
1294 #endif
1295         default:
1296             return CRYPT_SCHEME;
1297     }
1298 }
1299 #ifndef TPM_ALG_ECDSA // %

```

**B.10.4.1.18. ValidateSignatureEcdsa()**

This function validates an ECDSA signature.

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid

```

1300 static CRYPT_RESULT
1301 ValidateSignatureEcdsa(
1302     TPM2B_ECC_PARAMETER *rIn, // IN: r component of the signature
1303     TPM2B_ECC_PARAMETER *sIn, // IN: s component of the signature
1304     TPM_ECC_CURVE curveId, // IN: the curve used in the signature
1305                             // process
1306     TPMS_ECC_POINT *Qin, // IN: the public point of the key
1307     TPM2B *digest // IN: the digest that was signed
1308 )
1309 {
1310     TPM2B_ECC_PARAMETER U1;
1311     TPM2B_ECC_PARAMETER U2;
1312     TPMS_ECC_POINT R;
1313     const TPM2B *n;
1314     BN_CTX *context;
1315     EC_POINT *pQ = NULL;
1316     EC_GROUP *group = NULL;
1317     BIGNUM *bnU1;
1318     BIGNUM *bnU2;
1319     BIGNUM *bnR;
1320     BIGNUM *bnS;
1321     BIGNUM *bnW;
1322     BIGNUM *bnV;
1323     BIGNUM *bnN;
1324     BIGNUM *bnE;
1325     BIGNUM *bnGx;
1326     BIGNUM *bnGy;
1327     BIGNUM *bnQx;
1328     BIGNUM *bnQy;
1329     CRYPT_RESULT retVal = CRYPT_FAIL;
1330     int t;
1331
1332     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
1333
1334     // The curve selector should have been filtered by the unmarshaling process
1335     pAssert (curveData != NULL);
1336     n = curveData->n;
1337
1338     // 1. If r and s are not both integers in the interval [1, n - 1], output
1339     // INVALID.
1340     if( _math_uComp(rIn->t.size, rIn->t.buffer, n->size, n->buffer) >= 0
1341         || _math_uComp(sIn->t.size, sIn->t.buffer, n->size, n->buffer) >= 0
1342         )
1343         return CRYPT_FAIL;
1344
1345     context = BN_CTX_new();
1346     if(context == NULL)
1347         FAIL(FATAL_ERROR_ALLOCATION);
1348     BN_CTX_start(context);
1349     bnR = BN_CTX_get(context);
1350     bnS = BN_CTX_get(context);
1351     bnN = BN_CTX_get(context);
1352     bnE = BN_CTX_get(context);
1353     bnV = BN_CTX_get(context);
1354     bnW = BN_CTX_get(context);

```



```

1355     bnGx = BN_CTX_get(context);
1356     bnGy = BN_CTX_get(context);
1357     bnQx = BN_CTX_get(context);
1358     bnQy = BN_CTX_get(context);
1359     bnU1 = BN_CTX_get(context);
1360     bnU2 = BN_CTX_get(context);
1361
1362     // Assume the size variables do not overflow, which should not happen in
1363     // the contexts that this function will be called.
1364     assert2Bsize(Qin->x.t);
1365     assert2Bsize(rIn->t);
1366     assert2Bsize(sIn->t);
1367
1368     // BN_CTX_get() is sticky so only need to check the last value to know that
1369     // all worked.
1370     if( bnU2 == NULL
1371
1372         // initialize the group parameters
1373         || (group = EccCurveInit(curveId, context)) == NULL
1374
1375         // allocate a local point
1376         || (pQ = EC_POINT_new(group)) == NULL
1377
1378         // use the public key values (QxIn and QyIn) to initialize Q
1379         || BN_bin2bn(Qin->x.t.buffer, Qin->x.t.size, bnQx) == NULL
1380         || BN_bin2bn(Qin->y.t.buffer, Qin->y.t.size, bnQy) == NULL
1381         || !EC_POINT_set_affine_coordinates_GFp(group, pQ, bnQx, bnQy, context)
1382
1383         // convert the signature values
1384         || BN_bin2bn(rIn->t.buffer, rIn->t.size, bnR) == NULL
1385         || BN_bin2bn(sIn->t.buffer, sIn->t.size, bnS) == NULL
1386
1387         // convert the curve order
1388         || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
1389         FAIL(FATAL_ERROR_INTERNAL);
1390
1391
1392 // 2. Use the selected hash function to compute H0 = Hash(M0).
1393 // This is an input parameter
1394
1395 // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
1396 t = (digest->size > rIn->t.size) ? rIn->t.size : digest->size;
1397 if(BN_bin2bn(digest->buffer, t, bnE) == NULL)
1398     FAIL(FATAL_ERROR_INTERNAL);
1399
1400 // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
1401 if (BN_mod_inverse(bnW, bnS, bnN, context) == NULL)
1402     FAIL(FATAL_ERROR_INTERNAL);
1403
1404 // 5. Compute u1 = (e' * w) mod n, and compute u2 = (r' * w) mod n.
1405 if( !BN_mod_mul(bnU1, bnE, bnW, bnN, context)
1406     || !BN_mod_mul(bnU2, bnR, bnW, bnN, context))
1407     FAIL(FATAL_ERROR_INTERNAL);
1408
1409 BnTo2B(&U1.b, bnU1, (UINT16) BN_num_bytes(bnU1));
1410 BnTo2B(&U2.b, bnU2, (UINT16) BN_num_bytes(bnU2));
1411
1412 // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
1413 // scalar multiplication and EC addition (see [Routines]). If R is equal to
1414 // the point at infinity O, output INVALID.
1415 if(_cpri__EccPointMultiply(&R, curveId, &U1, Qin, &U2) == CRYPT_SUCCESS)
1416 {
1417     // 7. Compute v = Rx mod n.
1418     if( BN_bin2bn(R.x.t.buffer, R.x.t.size, bnV) == NULL
1419         || !BN_mod(bnV, bnV, bnN, context))
1420         FAIL(FATAL_ERROR_INTERNAL);

```

```

1421
1422 // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
1423     if(BN_cmp(bnV, bnR) == 0)
1424         retVal = CRYPT_SUCCESS;
1425     }
1426
1427     if(pQ != NULL) EC_POINT_free(pQ);
1428     if(group != NULL) EC_GROUP_free(group);
1429     BN_CTX_end(context);
1430     BN_CTX_free(context);
1431
1432     return retVal;
1433 }
1434 #endif // % TPM_ALG_ECDSA
1435 #ifdef TPM_ALG_EC Schnorr // %

```

#### B.10.4.1.19. ValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid
CRYPT_SCHEME	<i>hashAlg</i> is not supported

```

1436 static CRYPT_RESULT
1437 ValidateSignatureEcSchnorr(
1438     TPM2B_ECC_PARAMETER *rIn, // IN: r component of the signature
1439     TPM2B_ECC_PARAMETER *sIn, // IN: s component of the signature
1440     TPM_ALG_ID hashAlg, // IN: hash algorithm of the signature
1441     TPM_ECC_CURVE curveId, // IN: the curve used in the signature
1442     // process
1443     TPMS_ECC_POINT *Qin, // IN: the public point of the key
1444     TPM2B *digest // IN: the digest that was signed
1445 )
1446 {
1447     TPMS_ECC_POINT pE;
1448     const TPM2B *n;
1449     CPRI_HASH_STATE hashState;
1450     TPM2B_DIGEST rPrime;
1451     TPM2B_ECC_PARAMETER minusR;
1452     UINT16 digestSize = _cpri_GetDigestSize(hashAlg);
1453     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
1454
1455     // The curve parameter should have been filtered by unmarshaling code
1456     pAssert(curveData != NULL);
1457
1458     if(digestSize == 0)
1459         return CRYPT_SCHEME;
1460
1461     // Input parameter validation
1462     pAssert(rIn != NULL && sIn != NULL && Qin != NULL && digest != NULL);
1463
1464     n = curveData->n;
1465
1466     // if sIn or rIn are not between 1 and N-1, signature check fails
1467     if(
1468         _math_uComp(sIn->b.size, sIn->b.buffer, n->size, n->buffer) >= 0
1469         || _math_uComp(rIn->b.size, rIn->b.buffer, n->size, n->buffer) >= 0
1470     )
1471         return CRYPT_FAIL;
1472
1473     //E = [s]InG - [r]InQ

```

```

1473     _math__sub(n->size, n->buffer,
1474               rIn->t.size, rIn->t.buffer,
1475               &minusR.t.size, minusR.t.buffer);
1476     if(_cpri__EccPointMultiply(&pE, curveId, sIn, Qin, &minusR) != CRYPT_SUCCESS)
1477         return CRYPT_FAIL;
1478
1479     // Ex = Ex mod N
1480     if(Mod2B(&pE.x.b, n) != CRYPT_SUCCESS)
1481         FAIL(FATAL_ERROR_INTERNAL);
1482
1483     _math__Normalize2B(&pE.x.b);
1484
1485     // rPrime = h(digest || pE.x) mod n;
1486     _cpri__StartHash(hashAlg, FALSE, &hashState);
1487     _cpri__UpdateHash(&hashState, digest->size, digest->buffer);
1488     _cpri__UpdateHash(&hashState, pE.x.t.size, pE.x.t.buffer);
1489     if(_cpri__CompleteHash(&hashState, digestSize, rPrime.t.buffer) != digestSize)
1490         FAIL(FATAL_ERROR_INTERNAL);
1491
1492     rPrime.t.size = digestSize;
1493
1494     // rPrime = rPrime (mod n)
1495     if(Mod2B(&rPrime.b, n) != CRYPT_SUCCESS)
1496         FAIL(FATAL_ERROR_INTERNAL);
1497
1498     // If rIn and rPrime are not the same size, denormalize rPrime.
1499     if(rIn->t.size > rPrime.t.size)
1500         _math__Denormalize2B(&rPrime.b, rIn->t.size);
1501
1502     // see if the values match
1503     if ( rIn->t.size == rPrime.t.size
1504         && (memcmp(rIn->t.buffer, rPrime.t.buffer, rIn->t.size) == 0))
1505         return CRYPT_SUCCESS;
1506     else
1507         return CRYPT_FAIL;
1508 }
1509 #endif  /*% TPM_ALG_ECSCNORR
1510 #ifdef TPM_ALG_SM2  /*%

```

#### B.10.4.1.20. ValidateSignatureSM2Dsa()

This function is used to validate an SM2 signature.

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid

```

1511 static CRYPT_RESULT
1512 ValidateSignatureSM2Dsa(
1513     TPM2B_ECC_PARAMETER *rIn,      // IN: r component of the signature
1514     TPM2B_ECC_PARAMETER *sIn,      // IN: s component of the signature
1515     TPM_ECC_CURVE        curveId,  // IN: the curve used in the signature
1516                               // process
1517     TPMS_ECC_POINT       *Qin,      // IN: the public point of the key
1518     TPM2B                 *digest   // IN: the digest that was signed
1519 )
1520 {
1521     BIGNUM *bnR;
1522     BIGNUM *bnRp;
1523     BIGNUM *bnT;
1524     BIGNUM *bnS;
1525     BIGNUM *bnE;

```

```

1526     EC_POINT          *pQ;
1527     BN_CTX            *context;
1528     EC_GROUP          *group = NULL;
1529     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
1530     BOOL              fail = FALSE;
1531
1532
1533     if((context = BN_CTX_new()) == NULL || curveData == NULL)
1534         FAIL(FATAL_ERROR_INTERNAL);
1535     bnR = BN_CTX_get(context);
1536     bnRp= BN_CTX_get(context);
1537     bnE = BN_CTX_get(context);
1538     bnT = BN_CTX_get(context);
1539     bnS = BN_CTX_get(context);
1540     if( bnS == NULL
1541         || (group = EccCurveInit(curveId, context)) == NULL)
1542         FAIL(FATAL_ERROR_INTERNAL);
1543
1544     #ifdef _SM2_SIGN_DEBUG
1545     cpy_hexTo2B(&Qin->x.b,
1546               "0AE4C7798AA0F119471BEE11825BE46202BB79E2A5844495E97C04FF4DF2548A");
1547     cpy_hexTo2B(&Qin->y.b,
1548               "7C0240F88F1CD4E16352A73C17B7F16F07353E53A176D684A9FE0C6BB798E857");
1549     cpy_hexTo2B(digest,
1550               "B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9E9FFE76");
1551     #endif
1552     pQ = EccInitPoint2B(group, Qin, context);
1553
1554     #ifdef _SM2_SIGN_DEBUG
1555     pAssert(EC_POINT_get_affine_coordinates_GFp(group, pQ, bnT, bnS, context));
1556     pAssert(cmp_bn2hex(bnT,
1557                       "0AE4C7798AA0F119471BEE11825BE46202BB79E2A5844495E97C04FF4DF2548A")
1558            == 0);
1559     pAssert(cmp_bn2hex(bnS,
1560                       "7C0240F88F1CD4E16352A73C17B7F16F07353E53A176D684A9FE0C6BB798E857")
1561            == 0);
1562     #endif
1563
1564     BnFrom2B(bnR, &rIn->b);
1565     BnFrom2B(bnS, &sIn->b);
1566     BnFrom2B(bnE, digest);
1567
1568     #ifdef _SM2_SIGN_DEBUG
1569     // Make sure that the input signature is the test signature
1570     pAssert(cmp_2B2hex(&rIn->b,
1571                       "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1") == 0);
1572     pAssert(cmp_2B2hex(&sIn->b,
1573                       "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7") == 0);
1574     #endif
1575
1576     // a) verify that r and s are in the inclusive interval 1 to (n - 1)
1577     fail = BN_is_zero(bnR) || (BN_ucmp(bnR, &group->order) >= 0);
1578
1579     fail = BN_is_zero(bnS) || (BN_ucmp(bnS, &group->order) >= 0) || fail;
1580     if(fail)
1581         // There is no reason to continue. Since r and s are inputs from the caller,
1582         // they can know that the values are not in the proper range. So, exiting here
1583         // does not disclose any information.
1584         goto Cleanup;
1585
1586     // b) compute t := (r + s) mod n
1587     if(!BN_mod_add(bnT, bnR, bnS, &group->order, context))
1588         FAIL(FATAL_ERROR_INTERNAL);
1589     #ifdef _SM2_SIGN_DEBUG
1590     pAssert(cmp_bn2hex(bnT,
1591                       "2B75F07ED7ECE7CCC1C8986B991F441AD324D6D619FE06DD63ED32E0C997C801")

```

```

1592         == 0);
1593 #endif
1594
1595 // c) verify that t > 0
1596 if(BN_is_zero(bnT)) {
1597     fail = TRUE;
1598     // set to a value that should allow rest of the computations to run without
1599     // trouble
1600     BN_copy(bnT, bnS);
1601 }
1602 // d) compute (x, y) := [s]G + [t]Q
1603 if(!EC_POINT_mul(group, pQ, bnS, pQ, bnT, context))
1604     FAIL(FATAL_ERROR_INTERNAL);
1605 // Get the x coordinate of the point
1606 if(!EC_POINT_get_affine_coordinates_GFp(group, pQ, bnT, NULL, context))
1607     FAIL(FATAL_ERROR_INTERNAL);
1608
1609 #ifdef _SM2_SIGN_DEBUG
1610     pAssert(cmp_bn2hex(bnT,
1611         "110FCDA57615705D5E7B9324AC4B856D23E6D9188B2AE47759514657CE25D112")
1612         == 0);
1613 #endif
1614
1615 // e) compute r' := (e + x) mod n (the x coordinate is in bnT)
1616 if(!BN_mod_add(bnRp, bnE, bnT, &group->order, context))
1617     FAIL(FATAL_ERROR_INTERNAL);
1618
1619 // f) verify that r' = r
1620 fail = BN_ucmp(bnR, bnRp) != 0 || fail;
1621
1622 Cleanup:
1623 if(pQ) EC_POINT_free(pQ);
1624 if(group) EC_GROUP_free(group);
1625 BN_CTX_end(context);
1626 BN_CTX_free(context);
1627
1628 if(fail)
1629     return CRYPT_FAIL;
1630 else
1631     return CRYPT_SUCCESS;
1632 }
1633 #endif /*% TMP_ALG_SM2

```

#### B.10.4.1.21. \_cpri\_\_ValidateSignatureEcc()

This function validates

Return Value	Meaning
CRYPT_SUCCESS	signature is valid
CRYPT_FAIL	not a valid signature
CRYPT_SCHEME	unsupported scheme or hash algorithm

```

1634 BOOL
1635 _cpri__ValidateSignatureEcc(
1636     TPM2B_ECC_PARAMETER *rIn,      // IN: r component of the signature
1637     TPM2B_ECC_PARAMETER *sIn,      // IN: s component of the signature
1638     TPM_ALG_ID scheme,            // IN: the scheme selector
1639     TPM_ALG_ID hashAlg,           // IN: the hash algorithm used (not
1640                                     // used in all schemes)
1641     TPM_ECC_CURVE curveId,         // IN: the curve used in the signature
1642                                     // process
1643     TPMS_ECC_POINT *Qin,           // IN: the public point of the key

```

```

1644     TPM2B          *digest      // IN: the digest that was signed
1645 )
1646 {
1647     switch (scheme)
1648     {
1649         case TPM_ALG_ECDSA:
1650             return ValidateSignatureEcdsa(rIn, sIn, curveId, Qin, digest);
1651             break;
1652
1653 #ifdef TPM_ALG_ECSCNORR
1654         case TPM_ALG_ECSCNORR:
1655             return ValidateSignatureEcSchnorr(rIn, sIn, hashAlg, curveId, Qin,
1656                                             digest);
1657             break;
1658 #endif
1659
1660 #ifdef TPM_ALG_SM2
1661         case TPM_ALG_SM2:
1662             return ValidateSignatureSM2Dsa(rIn, sIn, curveId, Qin, digest);
1663 #endif
1664         default:
1665             break;
1666     }
1667     return CRYPT_SCHEME;
1668 }
1669 #if CC_ZGen_2Phase == YES //%
1670 #ifdef TPM_ALG_ECMQV

```

#### B.10.4.1.22. avf1()

This function does the associated value computation required by MQV key exchange. Process:

- h) Convert  $xQ$  to an integer  $xq_i$  using the convention specified in Appendix C. 3.
- i) Calculate  $xqm = xq_i \bmod 2^{\lceil f/2 \rceil}$  (where  $f = \lceil \log_2(n) \rceil$ ).
- j) Calculate the associate value function  $avf(Q) = xqm + 2^{\lceil f/2 \rceil}$

```

1671 static BOOL
1672 avf1(
1673     BIGNUM          *bnX,          // IN/OUT: the reduced value
1674     BIGNUM          *bnN          // IN: the order of the curve
1675 )
1676 {
1677     // compute f = 2^(ceil(ceil(log2(n)) / 2))
1678     int             f = (BN_num_bits(bnN) + 1) / 2;
1679     // x' = 2^f + (x mod 2^f)
1680     BN_mask_bits(bnX, f);        // This is mod 2*2^f but it doesn't matter because
1681                                 // the next operation will SET the extra bit anyway
1682     BN_set_bit(bnX, f);
1683     return TRUE;
1684 }

```

#### B.10.4.1.23. C\_2\_2\_MQV()

This function performs the key exchange defined in SP800-56A 6. 1. 1. 4 Full MQV, C(2, 2, ECC MQV).

CAUTION: Implementation of this function may require use of essential claims in patents not owned by TCG members.

Points  $QsB()$  and  $QeB()$  are required to be on the curve of  $inQsA$ . The function will fail, possibly catastrophically, if this is not the case.

Return Value	Meaning
CRYPT_SUCCESS	results is valid
CRYPT_NO_RESULT	the value for dsA does not give a valid point on the curve

```

1685 static CRYPT_RESULT
1686 C_2_2_MQV(
1687     TPMS_ECC_POINT      *outZ,           // OUT: the computed point
1688     TPM_ECC_CURVE       curveId,        // IN: the curve for the computations
1689     TPM2B_ECC_PARAMETER *dsA,          // IN: static private TPM key
1690     TPM2B_ECC_PARAMETER *deA,          // IN: ephemeral private TPM key
1691     TPMS_ECC_POINT      *QsB,          // IN: static public party B key
1692     TPMS_ECC_POINT      *QeB,          // IN: ephemeral public party B key
1693 )
1694 {
1695     BN_CTX                *context;
1696     EC_POINT              *pQeA = NULL;
1697     EC_POINT              *pQeB = NULL;
1698     EC_POINT              *pQsB = NULL;
1699     EC_GROUP              *group = NULL;
1700     BIGNUM                *bnTa;
1701     BIGNUM                *bnDeA;
1702     BIGNUM                *bnDsA;
1703     BIGNUM                *bnXeA;       // x coordinate of ephemeral party A key
1704     BIGNUM                *bnH;
1705     BIGNUM                *bnN;
1706     BIGNUM                *bnXeB;
1707     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
1708     CRYPT_RESULT          retVal;
1709
1710     pAssert( curveData != NULL && outZ != NULL && dsA != NULL
1711             && deA != NULL && QsB != NULL && QeB != NULL);
1712
1713     context = BN_CTX_new();
1714     if(context == NULL || curveData == NULL)
1715         FAIL(FATAL_ERROR_ALLOCATION);
1716     BN_CTX_start(context);
1717     bnTa = BN_CTX_get(context);
1718     bnDeA = BN_CTX_get(context);
1719     bnDsA = BN_CTX_get(context);
1720     bnXeA = BN_CTX_get(context);
1721     bnH = BN_CTX_get(context);
1722     bnN = BN_CTX_get(context);
1723     bnXeB = BN_CTX_get(context);
1724     if(bnXeB == NULL)
1725         FAIL(FATAL_ERROR_ALLOCATION);
1726
1727     // Process:
1728     // 1.  $\text{implicitsigA} = (de,A + \text{avf}(Qe,A)ds,A) \bmod n$ .
1729     // 2.  $P = h(\text{implicitsigA})(Qe,B + \text{avf}(Qe,B)Qs,B)$ .
1730     // 3. If  $P = O$ , output an error indicator.
1731     // 4.  $Z=xP$ , where  $xP$  is the x-coordinate of P.
1732
1733     // Initialize group parameters and local values of input
1734     if((group = EccCurveInit(curveId, context)) == NULL)
1735         FAIL(FATAL_ERROR_INTERNAL);
1736
1737     if((pQeA = EC_POINT_new(group)) == NULL)
1738         FAIL(FATAL_ERROR_ALLOCATION);
1739
1740     BnFrom2B(bnDeA, &deA->b);
1741     BnFrom2B(bnDsA, &dsA->b);
1742     BnFrom2B(bnH, curveData->h);
1743     BnFrom2B(bnN, curveData->n);

```

```

1744     BnFrom2B(bnXeB, &QeB->x.b);
1745     pQeB = EccInitPoint2B(group, QeB, context);
1746     pQsB = EccInitPoint2B(group, QsB, context);
1747
1748     // Compute the public ephemeral key pQeA = [de,A]G
1749     if( (retVal = PointMul(group, pQeA, bnDeA, NULL, NULL, context))
1750         != CRYPT_SUCCESS)
1751         goto Cleanup;
1752
1753     if(EC_POINT_get_affine_coordinates_GFp(group, pQeA, bnXeA, NULL, context) != 1)
1754         FAIL(FATAL_ERROR_INTERNAL);
1755
1756     // 1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
1757     // tA := (ds,A + de,A * avf(Xe,A)) mod n (3)
1758     // Compute 'tA' = ('deA' + 'dsA' * avf('XeA')) mod n
1759     // Ta = avf(XeA);
1760     BN_copy(bnTa, bnXeA);
1761     avf1(bnTa, bnN);
1762     if(// do Ta = ds,A * Ta mod n = dsA * avf(XeA) mod n
1763         !BN_mod_mul(bnTa, bnDsA, bnTa, bnN, context)
1764
1765         // now Ta = deA + Ta mod n = deA + dsA * avf(XeA) mod n
1766         || !BN_mod_add(bnTa, bnDeA, bnTa, bnN, context)
1767     )
1768         FAIL(FATAL_ERROR_INTERNAL);
1769
1770     // 2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
1771     // Put this in because almost every case of h is == 1 so skip the call when
1772     // not necessary.
1773     if(!BN_is_one(bnH))
1774     {
1775         // Cofactor is not 1 so compute Ta := Ta * h mod n
1776         if(!BN_mul(bnTa, bnTa, bnH, context))
1777             FAIL(FATAL_ERROR_INTERNAL);
1778     }
1779
1780
1781     // Now that 'tA' is (h * 'tA' mod n)
1782     // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
1783
1784     // first, compute XeB = avf(XeB)
1785     avf1(bnXeB, bnN);
1786
1787     // QsB := [XeB]QsB
1788     if( !EC_POINT_mul(group, pQsB, NULL, pQsB, bnXeB, context)
1789
1790         // QeB := QsB + QeB
1791         || !EC_POINT_add(group, pQeB, pQeB, pQsB, context)
1792     )
1793         FAIL(FATAL_ERROR_INTERNAL);
1794
1795     // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
1796     if(PointMul(group, pQeB, NULL, pQeB, bnTa, context) == CRYPT_SUCCESS)
1797         // Convert BIGNUM E to TPM2B E
1798         Point2B(group, outZ, pQeB, (UINT16)BN_num_bytes(bnN), context);
1799
1800 Cleanup:
1801     if(pQeA != NULL) EC_POINT_free(pQeA);
1802     if(pQeB != NULL) EC_POINT_free(pQeB);
1803     if(pQsB != NULL) EC_POINT_free(pQsB);
1804     if(group != NULL) EC_GROUP_free(group);
1805     BN_CTX_end(context);
1806     BN_CTX_free(context);
1807
1808     return retVal;
1809

```



```

1810 }
1811 #endif // TPM_ALG_ECMQV
1812 #ifdef TPM_ALG_SM2 //%
```

#### B.10.4.1.24. avfSm2()

This function does the associated value computation required by SM2 key exchange. This is different from the avf() in the international standards because it returns a value that is half the size of the value returned by the standard avf. For example, if n is 15, Ws (w in the standard) is 2 but the W here is 1. This means that an input value of 14 (110b) would return a value of 110b with the standard but 10b with the scheme in SM2.

```

1813 static BOOL
1814 avfSm2(
1815     BIGNUM          *bnX,          // IN/OUT: the reduced value
1816     BIGNUM          *bnN,          // IN: the order of the curve
1817 )
1818 {
1819 // a) set w := ceil(ceil(log2(n)) / 2) - 1
1820     int              w = ((BN_num_bits(bnN) + 1) / 2) - 1;
1821
1822 // b) set x' := 2^w + ( x & (2^w - 1))
1823 // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
1824     BN_mask_bits(bnX, w); // as wiht avf1, this is too big by a factor of 2 but
1825                          // it doesn't matter because we SET the extra bit anyway
1826     BN_set_bit(bnX, w);
1827     return TRUE;
1828 }
```

*SM2KeyExchange()* This function performs the key exchange defined in SM2. The first step is to compute  $tA = (dsA + deA \cdot avf(Xe,A)) \bmod n$ . Then, compute the Z value from  $outZ = (h \cdot tA \bmod n) (QsA + [avf(QeB). x])(QeB))$ . The function will compute the ephemeral public key from the ephemeral private key. All points are required to be on the curve of *inQsA*. The function will fail catastrophically if this is not the case

Return Value	Meaning
CRYPT_SUCCESS	results is valid
CRYPT_NO_RESULT	the value for <i>dsA</i> does not give a valid point on the curve

```

1829 static CRYPT_RESULT
1830 SM2KeyExchange(
1831     TPMS_ECC_POINT      *outZ,          // OUT: the computed point
1832     TPM_ECC_CURVE       curveId,        // IN: the curve for the computations
1833     TPM2B_ECC_PARAMETER *dsA,          // IN: static private TPM key
1834     TPM2B_ECC_PARAMETER *deA,          // IN: ephemeral private TPM key
1835     TPMS_ECC_POINT      *QsB,          // IN: static public party B key
1836     TPMS_ECC_POINT      *QeB,          // IN: ephemeral public party B key
1837 )
1838 {
1839     BN_CTX              *context;
1840     EC_POINT            *pQeA = NULL;
1841     EC_POINT            *pQeB = NULL;
1842     EC_POINT            *pQsB = NULL;
1843     EC_GROUP            *group = NULL;
1844     BIGNUM              *bnTa;
1845     BIGNUM              *bnDeA;
1846     BIGNUM              *bnDsA;
1847     BIGNUM              *bnXeA;          // x coordinate of ephemeral party A key
1848     BIGNUM              *bnH;
1849     BIGNUM              *bnN;
```

```

1850     BIGNUM                *bnXeB;
1851     const ECC_CURVE_DATA  *curveData = GetCurveData(curveId);
1852     CRYPT_RESULT          retVal;
1853
1854     pAssert(      curveData != NULL && outZ != NULL && dsA != NULL
1855               &&      deA != NULL && QsB != NULL && QeB != NULL);
1856
1857     context = BN_CTX_new();
1858     if(context == NULL || curveData == NULL)
1859         FAIL(FATAL_ERROR_ALLOCATION);
1860     BN_CTX_start(context);
1861     bnTa = BN_CTX_get(context);
1862     bnDeA = BN_CTX_get(context);
1863     bnDsA = BN_CTX_get(context);
1864     bnXeA = BN_CTX_get(context);
1865     bnH = BN_CTX_get(context);
1866     bnN = BN_CTX_get(context);
1867     bnXeB = BN_CTX_get(context);
1868     if(bnXeB == NULL)
1869         FAIL(FATAL_ERROR_ALLOCATION);
1870
1871     // Initialize group parameters and local values of input
1872     if((group = EccCurveInit(curveId, context)) == NULL)
1873         FAIL(FATAL_ERROR_INTERNAL);
1874
1875     if((pQeA = EC_POINT_new(group)) == NULL)
1876         FAIL(FATAL_ERROR_ALLOCATION);
1877
1878     BnFrom2B(bnDeA, &deA->b);
1879     BnFrom2B(bnDsA, &dsA->b);
1880     BnFrom2B(bnH, curveData->h);
1881     BnFrom2B(bnN, curveData->n);
1882     BnFrom2B(bnXeB, &QeB->x.b);
1883     pQeB = EccInitPoint2B(group, QeB, context);
1884     pQsB = EccInitPoint2B(group, QsB, context);
1885
1886     // Compute the public ephemeral key pQeA = [de,A]G
1887     if(      (retVal = PointMul(group, pQeA, bnDeA, NULL, NULL, context))
1888         != CRYPT_SUCCESS)
1889         goto Cleanup;
1890
1891     if(EC_POINT_get_affine_coordinates_Gfp(group, pQeA, bnXeA, NULL, context) != 1)
1892         FAIL(FATAL_ERROR_INTERNAL);
1893
1894     // tA := (ds,A + de,A * avf(Xe,A)) mod n    (3)
1895     // Compute 'tA' = ('dsA' + 'deA' * avf('XeA')) mod n
1896     // Ta = avf(XeA);
1897     BN_copy(bnTa, bnXeA);
1898     avfSm2(bnTa, bnN);
1899     if(// do Ta = de,A * Ta mod n = deA * avf(XeA) mod n
1900        !BN_mod_mul(bnTa, bnDeA, bnTa, bnN, context)
1901
1902        // now Ta = dsA + Ta mod n = dsA + deA * avf(XeA) mod n
1903        || !BN_mod_add(bnTa, bnDsA, bnTa, bnN, context)
1904        )
1905         FAIL(FATAL_ERROR_INTERNAL);
1906
1907     // outZ ? [h * tA mod n] (Qs,B + [avf(Xe,B)](Qe,B))    (4)
1908     // Put this in because almost every case of h is == 1 so skip the call when
1909     // not necessary.
1910     if(!BN_is_one(bnH))
1911     {
1912         // Cofactor is not 1 so compute Ta := Ta * h mod n
1913         if(!BN_mul(bnTa, bnTa, bnH, context))
1914             FAIL(FATAL_ERROR_INTERNAL);
1915     }

```

```

1916
1917
1918 // Now that 'tA' is (h * 'tA' mod n)
1919 // 'outZ' = ['tA'](QsB + [avf(QeB.x)](QeB)).
1920
1921 // first, compute XeB = avf(XeB)
1922 avfSm2(bnXeB, bnN);
1923
1924 // QeB := [XeB]QeB
1925 if(!EC_POINT_mul(group, pQeB, NULL, pQeB, bnXeB, context)
1926
1927 // QeB := QsB + QeB
1928 || !EC_POINT_add(group, pQeB, pQeB, pQsB, context)
1929 )
1930 FAIL(FATAL_ERROR_INTERNAL);
1931
1932 // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
1933 if(PointMul(group, pQeB, NULL, pQeB, bnTa, context) == CRYPT_SUCCESS)
1934 // Convert BIGNUM E to TPM2B E
1935 Point2B(group, outZ, pQeB, (UINT16)BN_num_bytes(bnN), context);
1936
1937 Cleanup:
1938 if(pQeA != NULL) EC_POINT_free(pQeA);
1939 if(pQeB != NULL) EC_POINT_free(pQeB);
1940 if(pQsB != NULL) EC_POINT_free(pQsB);
1941 if(group != NULL) EC_GROUP_free(group);
1942 BN_CTX_end(context);
1943 BN_CTX_free(context);
1944
1945 return retVal;
1946
1947 }
1948 #endif // % TPM_ALG_SM2

```

#### B.10.4.1.25. C\_2\_2\_ECDH()

This function performs the two phase key exchange defined in SP800-56A, 6. 1. 1. 2 Full Unified Model, C(2, 2, ECC CDH).

```

1949 static CRYPT_RESULT
1950 C_2_2_ECDH(
1951     TPMS_ECC_POINT      *outZ1,           // OUT: Zs
1952     TPMS_ECC_POINT      *outZ2,           // OUT: Ze
1953     TPM_ECC_CURVE        curveId,         // IN: the curve for the computations
1954     TPM2B_ECC_PARAMETER  *dsA,           // IN: static private TPM key
1955     TPM2B_ECC_PARAMETER  *deA,           // IN: ephemeral private TPM key
1956     TPMS_ECC_POINT      *QsB,           // IN: static public party B key
1957     TPMS_ECC_POINT      *QeB,           // IN: ephemeral public party B key
1958 )
1959 {
1960     BN_CTX                *context;
1961     EC_POINT              *pQ = NULL;
1962     EC_GROUP              *group = NULL;
1963     BIGNUM                *bnD;
1964     UINT16                size;
1965     const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
1966
1967     context = BN_CTX_new();
1968     if(context == NULL || curveData == NULL)
1969         FAIL(FATAL_ERROR_ALLOCATION);
1970     BN_CTX_start(context);
1971     if((bnD = BN_CTX_get(context)) == NULL)
1972         FAIL(FATAL_ERROR_INTERNAL);
1973

```

```

1974 // Initialize group parameters and local values of input
1975 if((group = EccCurveInit(curveId, context)) == NULL)
1976     FAIL(FATAL_ERROR_INTERNAL);
1977 size = (UINT16)BN_num_bytes(&group->order);
1978
1979 // Get the static private key of A
1980 BnFrom2B(bnD, &dsA->b);
1981
1982 // Initialize the static public point from B
1983 pQ = EccInitPoint2B(group, QsB, context);
1984
1985 // Do the point multiply for the Zs value
1986 if(PointMul(group, pQ, NULL, pQ, bnD, context) != CRYPT_NO_RESULT)
1987     // Convert the Zs value
1988     Point2B(group, outZ1, pQ, size, context);
1989
1990 // Get the ephemeral private key of A
1991 BnFrom2B(bnD, &deA->b);
1992
1993 // Initialize the ephemeral public point from B
1994 PointFrom2B(group, pQ, QeB, context);
1995
1996 // Do the point multiply for the Ze value
1997 if(PointMul(group, pQ, NULL, pQ, bnD, context) != CRYPT_NO_RESULT)
1998     // Convert the Ze value.
1999     Point2B(group, outZ2, pQ, size, context);
2000
2001 if(pQ != NULL) EC_POINT_free(pQ);
2002 if(group != NULL) EC_GROUP_free(group);
2003 BN_CTX_end(context);
2004 BN_CTX_free(context);
2005 return CRYPT_SUCCESS;
2006 }

```

#### B.10.4.1.26. \_cpri\_\_C\_2\_2\_KeyExchange()

This function is the dispatch routine for the EC key exchange functions that use two ephemeral and two static keys.

Return Value	Meaning
CRYPT_SCHEME	scheme is not defined

```

2007 CRYPT_RESULT
2008 _cpri__C_2_2_KeyExchange(
2009     TPMS_ECC_POINT *outZ1, // OUT: a computed point
2010     TPMS_ECC_POINT *outZ2, // OUT: and optional second point
2011     TPM_ECC_CURVE curveId, // IN: the curve for the computations
2012     TPM_ALG_ID scheme, // IN: the key exchange scheme
2013     TPM2B_ECC_PARAMETER *dsA, // IN: static private TPM key
2014     TPM2B_ECC_PARAMETER *deA, // IN: ephemeral private TPM key
2015     TPMS_ECC_POINT *QsB, // IN: static public party B key
2016     TPMS_ECC_POINT *QeB // IN: ephemeral public party B key
2017 )
2018 {
2019     pAssert( outZ1 != NULL
2020             && dsA != NULL && deA != NULL
2021             && QsB != NULL && QeB != NULL);
2022
2023     // Initialize the output points so that they are empty until one of the
2024     // functions decides otherwise
2025     outZ1->x.b.size = 0;
2026     outZ1->y.b.size = 0;
2027     if(outZ2 != NULL)

```

```

2028     {
2029         outZ2->x.b.size = 0;
2030         outZ2->y.b.size = 0;
2031     }
2032
2033     switch (scheme)
2034     {
2035         case TPM_ALG_ECDH:
2036             return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
2037             break;
2038 #ifdef TPM_ALG_ECMQV
2039         case TPM_ALG_ECMQV:
2040             return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
2041             break;
2042 #endif
2043 #ifdef TPM_ALG_SM2
2044         case TPM_ALG_SM2:
2045             return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
2046             break;
2047 #endif
2048         default:
2049             return CRYPT_SCHEME;
2050     }
2051 }
2052 #else    //%
```

Stub used when the 2-phase key exchange is not defined so that the linker has something to associate with the value in the .def file.

```

2053 CRYPT_RESULT
2054 __cpri__C_2_2_KeyExchange()
2055 {
2056     return CRYPT_FAIL;
2057 }
2058 #endif //% CC_ZGen_2Phase
```

**Annex C**  
(informative)  
**Simulation Environment**

**C.1 Introduction**

These files are used to simulate some of the implementation-dependent hardware of a TPM. These files are provided to allow creation of a simulation environment for the TPM. These files are not expected to be part of a hardware TPM implementation.

## C.2 Cancel.c

### C.2.1. Introduction

This module simulates the cancel pins on the TPM.

### C.2.2. Includes, Typedefs, Structures, and Defines

```
1 #include "PlatformData.h"
```

### C.2.3. Functions

#### C.2.3.1. \_plat\_\_IsCanceled()

Check if the cancel flag is set

Return Value	Meaning
TRUE	if cancel flag is set
FALSE	if cancel flag is not set

```
2  BOOL
3  _plat__IsCanceled(void)
4  {
5      // return cancel flag
6      return s_isCanceled;
7  }
```

#### C.2.3.2. \_plat\_\_SetCancel()

Set cancel flag.

```
8  void
9  _plat__SetCancel(void)
10 {
11     s_isCanceled = TRUE;
12     return;
13 }
```

#### C.2.3.3. \_plat\_\_ClearCancel()

Clear cancel flag

```
14 void
15 _plat__ClearCancel(void)
16 {
17     s_isCanceled = FALSE;
18     return;
19 }
```



## C.3 Clock.c

### C.3.1. Introduction

This file contains the routines that are used by the simulator to mimic a hardware clock on a TPM. In this implementation, all the time values are measured in millisecond. However, the precision of the clock functions may be implementation dependent.

### C.3.2. Includes and Data Definitions

```

1  #include <time.h>
2  //#include <bool.h>
3  #include <assert.h>
4  #include "PlatformData.h"
5  #include "Platform.h"

```

### C.3.3. Functions

#### C.3.3.1. `_plat__ClockReset()`

Set the current clock time as initial time. This function is called at a power on event to reset the clock

```

6  void
7  _plat__ClockReset(void)
8  {
9      // Implementation specific: Microsoft C set CLOCKS_PER_SEC to be 1/1000,
10     // so here the measurement of clock() is in millisecond.
11     s_initClock = clock();
12     s_adjustRate = CLOCK_NOMINAL;
13
14     return;
15 }

```

#### C.3.3.2. `_plat__ClockTimeFromStart()`

Function returns the compensated time from the start of the command when `_plat__ClockTimeFromStart()` was called.

```

16 unsigned long long
17 _plat__ClockTimeFromStart(
18     void
19 )
20 {
21     unsigned long long currentClock = clock();
22     return ((currentClock - s_initClock) * CLOCK_NOMINAL) / s_adjustRate;
23 }

```

#### C.3.3.3. `_plat__ClockTimeElapsed()`

Get the time elapsed from current to the last time the `_plat__ClockTimeElapsed()` is called. For the first `_plat__ClockTimeElapsed()` call after a power on event, this call report the elapsed time from power on to the current call

```

24 unsigned long long
25 _plat__ClockTimeElapsed(void)
26 {

```

```

27     unsigned long long elapsed;
28     unsigned long long currentClock = clock();
29     elapsed = ((currentClock - s_initClock) * CLOCK_NOMINAL) / s_adjustRate;
30     s_initClock += (elapsed * s_adjustRate) / CLOCK_NOMINAL;
31
32     #ifdef DEBUGGING_TIME
33         // Put this in so that TPM time will pass much faster than real time when
34         // doing debug.
35         // A value of 1000 for DEBUG_TIME_MULTIPLIER will make each ms into a second
36         // A good value might be 100
37         elapsed *= DEBUG_TIME_MULTIPLIER
38     #endif
39         return elapsed;
40 }

```

#### C.3.3.4. `_plat__ClockAdjustRate()`

Adjust the clock rate

```

41 void
42 _plat__ClockAdjustRate(
43     int adjust // IN: the adjust number. It could be
44               // positive or negative
45 )
46 {
47     // We expect the caller should only use a fixed set of constant values to
48     // adjust the rate
49     switch(adjust)
50     {
51         case CLOCK_ADJUST_COARSE:
52             s_adjustRate += CLOCK_ADJUST_COARSE;
53             break;
54         case -CLOCK_ADJUST_COARSE:
55             s_adjustRate -= CLOCK_ADJUST_COARSE;
56             break;
57         case CLOCK_ADJUST_MEDIUM:
58             s_adjustRate += CLOCK_ADJUST_MEDIUM;
59             break;
60         case -CLOCK_ADJUST_MEDIUM:
61             s_adjustRate -= CLOCK_ADJUST_MEDIUM;
62             break;
63         case CLOCK_ADJUST_FINE:
64             s_adjustRate += CLOCK_ADJUST_FINE;
65             break;
66         case -CLOCK_ADJUST_FINE:
67             s_adjustRate -= CLOCK_ADJUST_FINE;
68             break;
69         default:
70             assert(FALSE);
71             break;
72     }
73
74
75     if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
76         s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
77     if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
78         s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;
79
80     return;
81 }

```

## C.4 Entropy.c

### C.4.1. Includes

```
1  #define _CRT_RAND_S
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <memory.h>
5  #include "bool.h"
6  #include "Platform.h"
```

### C.4.2. \_plat\_\_GetEntropy()

```
7  unsigned int
8  _plat__GetEntropy (
9      unsigned char* EntropyBuffer,
10     unsigned int    EntropySize // Assumption: EntropyBuffer is big enough to
11                                     // receive it, we don't do any checks for that
12 )
13 {
14     unsigned int rndNum;
15     unsigned int i;
16     errno_t err;
17
18     // Use h/w random number generator to build entropy for crypto PRNG.
19     for ( i = 0; i < EntropySize/<K>sizeof(unsigned int); i++)
20     {
21         err = rand_s(&rndNum);
22         assert(err == 0);
23
24         memcpy((char *)EntropyBuffer+i*sizeof(unsigned int),
25              (char *)&rndNum,
26              sizeof(unsigned int));
27     }
28
29     return EntropySize;
30 }
```

## C.5 LocalityPlat.c

### C.5.1. Includes

```
1 #include <assert.h>
2 #include "PlatformData.h"
```

### C.5.2. Functions

#### C.5.2.1. `_plat__LocalityGet()`

Get the most recent command locality in locality value form

```
3 unsigned char
4 _plat__LocalityGet(void)
5 {
6     return s_locality;
7 }
```

#### C.5.2.2. `_plat__LocalitySet()`

Set the most recent command locality in locality value form

```
8 void
9 _plat__LocalitySet(
10     unsigned char    locality
11 )
12 {
13     assert(locality <= 4 || locality > 31);
14     s_locality = locality;
15     return;
16 }
```

## C.6 NVMem.c

### C.6.1. Introduction

This file contains the NV read and write access methods. This implementation uses RAM/file and does not manage the RAM/file as NV blocks. The implementation may become more sophisticated over time.

### C.6.2. Includes

```

1  #include <assert.h>
2  #include <memory.h>
3  #include <string.h>
4  #include "PlatformData.h"

```

### C.6.3. Functions

#### C.6.3.1. \_plat\_\_NVEnable()

Enable NV memory

Return Value	Meaning
0	if success
non-0	if fail

```

5  int
6  _plat__NVEnable(
7      void    *platParameter           // IN: platform specific parameters
8  )
9  {
10     platParameter = 0;           // to try to satisfy the compiler and remove warning
11
12     #ifndef FILE_BACKED_NV
13
14         if(s_NVFile != NULL) return 0;
15
16         // Try to open an exist NVChip file for read/write
17         if(0 != fopen_s(&s_NVFile, "NVChip", "r+b"))
18             s_NVFile = NULL;
19
20         if(NULL != s_NVFile)
21         {
22             // See if the NVChip file is empty
23             fseek(s_NVFile, 0, SEEK_END);
24             if(0 == ftell(s_NVFile))
25                 s_NVFile = NULL;
26         }
27
28         if(s_NVFile == NULL)
29         {
30             // Initialize all the byte in the new file to 0
31             memset(s_NV, 0, NV_MEMORY_SIZE);
32
33             // If NVChip file does not exist, try to create it for read/write
34             fopen_s(&s_NVFile, "NVChip", "w+b");
35             // Start initialize at the end of new file
36             fseek(s_NVFile, 0, SEEK_END);
37             // Write 0s to NVChip file
38             fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);

```

```

39     }
40     else
41     {
42         // If NVChip file exist, assume the size is correct
43         fseek(s_NVFile, 0, SEEK_END);
44         assert(ftell(s_NVFile) == NV_MEMORY_SIZE);
45         // read NV file data to memory
46         fseek(s_NVFile, 0, SEEK_SET);
47         fread(s_NV, NV_MEMORY_SIZE, 1, s_NVFile);
48     }
49 #endif
50
51     return 0;
52 }

```

### C.6.3.2. `_plat__NVDisable()`

Disable NV memory

```

53 void
54 _plat__NVDisable(void)
55 {
56 #ifdef FILE_BACKED_NV
57
58     assert(s_NVFile != NULL);
59     // Close NV file
60     fclose(s_NVFile);
61     // Set file handle to NULL
62     s_NVFile = NULL;
63
64 #endif
65
66     return;
67 }

```

### C.6.3.3. `_plat__IsNvAvailable()`

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```

68 int
69 _plat__IsNvAvailable(void)
70 {
71
72     if(s_NvIsAvailable == FALSE)
73         return 1;
74
75 #ifdef FILE_BACKED_NV
76     if(s_NVFile == NULL)
77         return 1;
78 #endif
79
80     return 0;
81
82 }

```

**C.6.3.4. \_plat\_\_NvMemoryRead()**

Function: Read a chunk of NV memory

```

83 void
84 _plat__NvMemoryRead(
85     unsigned int    startOffset,           // IN: read start
86     unsigned int    size,                 // IN: size of bytes to read
87     void            *data                 // OUT: data buffer
88 )
89 {
90     assert(startOffset + size <= NV_MEMORY_SIZE);
91
92     // Copy data from RAM
93     memcpy(data, &s_NV[startOffset], size);
94     return;
95 }

```

**C.6.3.5. \_plat\_\_NvIsDifferent()**

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE	the NV location is different from the test value
FALSE	the NV location is the same as the test value

```

96 BOOL
97 _plat__NvIsDifferent(
98     unsigned int    startOffset,           // IN: read start
99     unsigned int    size,                 // IN: size of bytes to read
100    void            *data                 // IN: data buffer
101 )
102 {
103     return (memcmp(&s_NV[startOffset], data, size) != 0);
104 }

```

**C.6.3.6. \_plat\_\_NvMemoryWrite()**

This function is used to update NV memory. The **write** is to a memory copy of NV. At the end of the current command, any changes are written to the actual NV memory.

```

105 void
106 _plat__NvMemoryWrite(
107     unsigned int    startOffset,           // IN: write start
108     unsigned int    size,                 // IN: size of bytes to write
109     void            *data                 // OUT: data buffer
110 )
111 {
112     assert(startOffset + size <= NV_MEMORY_SIZE);
113
114     // Copy the data to the NV image
115     memcpy(&s_NV[startOffset], data, size);
116 }

```

**C.6.3.7. \_plat\_\_NvMemoryMove()**

Function: Move a chunk of NV memory from source to destination This function should ensure that if there overlap, the original data is copied before it is written

```

117 void
118 _plat__NvMemoryMove(
119     unsigned int    sourceOffset,           // IN: source offset
120     unsigned int    destOffset,           // IN: destination offset
121     unsigned int    size                   // IN: size of data being moved
122 )
123 {
124     assert(sourceOffset + size <= NV_MEMORY_SIZE);
125     assert(destOffset + size <= NV_MEMORY_SIZE);
126
127     // Move data in RAM
128     memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);
129
130     return;
131 }
132

```

**C.6.3.8. \_plat\_\_NvCommit()**

Update NV chip

Return Value	Meaning
0	NV write success
non-0	NV write fail

```

133 int
134 _plat__NvCommit(void)
135 {
136     #ifdef FILE_BACKED_NV
137         // If NV file is not available, return failure
138         if(s_NVfile == NULL || s_NvIsAvailable == FALSE)
139             return 1;
140
141         // Write RAM data to NV
142         fseek(s_NVfile, 0, SEEK SET);
143         fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVfile);
144         return 0;
145     #else
146         return 0;
147     #endif
148 }
149

```

**C.6.3.9. \_plat\_\_SetNvAvail()**

Set the current NV state to available. This function is for testing purpose only. It is not part of the platform NV logic

```

150 void
151 _plat__SetNvAvail(void)
152 {
153     s_NvIsAvailable = TRUE;
154     return;
155 }

```



**C.6.3.10. `_plat__ClearNvAvail()`**

Set the current NV state to unavailable. This function is for testing purpose only. It is not part of the platform NV logic

```
156 void
157 _plat__ClearNvAvail(void)
158 {
159     s_NvIsAvailable = FALSE;
160     return;
161 }
```

## C.7 PowerPlat.c

### C.7.1. Includes and Function Prototypes

```
1 #include <assert.h>
2 #include "PlatformData.h"
3 #include "Platform.h"
```

Platform power on and off functions

### C.7.2. Functions

#### C.7.2.1. \_plat\_\_Signal\_PowerOn()

Signal platform power on

```
4 void
5 _plat__Signal_PowerOn(void)
6 {
7     // Start clock
8     _plat__ClockReset();
9     // Prepare NV memory for power on
10    _plat__NVEnable(0);
11
12    return;
13 }
```

#### C.7.2.2. \_plat\_\_Signal\_PowerOff()

Signal platform power off

```
14 void
15 _plat__Signal_PowerOff(void)
16 {
17     // Prepare NV memory for power off
18     _plat__NVDisable();
19
20     return;
21 }
```

## C.8 Platform.h

```
1 #ifndef PLATFORM_H
2 #define PLATFORM_H
```

### C.8.1. Includes

```
3 // #include "bool.h"
```

### C.8.2. Power Functions

#### C.8.2.1. \_plat\_\_Signal\_PowerOn

Signal power on This signal is simulate by a RPC call

```
4 void
5 _plat__Signal_PowerOn(void);
```

#### C.8.2.2. \_plat\_\_Signal\_PowerOff()

Signal power off This signal is simulate by a RPC call

```
6 void
7 _plat__Signal_PowerOff(void);
```

### C.8.3. Physical Presence Functions

#### C.8.3.1. \_plat\_\_PhysicalPresenceAsserted()

Check if physical presence is signaled

Return Value	Meaning
TRUE	if physical presence is signaled
FALSE	if physical presence is not signaled

```
8 BOOL
9 _plat__PhysicalPresenceAsserted(void);
```

#### C.8.3.2. \_plat\_\_Signal\_PhysicalPresenceOn

Signal physical presence on This signal is simulate by a RPC call

```
10 void
11 _plat__Signal_PhysicalPresenceOn(void);
```

#### C.8.3.3. \_plat\_\_Signal\_PhysicalPresenceOff()

Signal physical presence off This signal is simulate by a RPC call

```
12 void
13 _plat__Signal_PhysicalPresenceOff(void);
```

**C.8.4. Command Canceling Functions****C.8.4.1. `_plat__IsCanceled()`**

Check if the cancel flag is set

Return Value	Meaning
TRUE	if cancel flag is set
FALSE	if cancel flag is not set

```
14  BOOL
15  _plat__IsCanceled(void);
```

**C.8.4.2. `_plat__SetCancel()`**

Set cancel flag.

```
16  void
17  _plat__SetCancel(void);
```

**C.8.4.3. `_plat__ClearCancel()`**

Clear cancel flag

```
18  void
19  _plat__ClearCancel( void);
```

**C.8.5. NV memory functions****C.8.5.1. `_plat__NVEnable()`**

Enable platform NV memory NV memory is automatically enabled at power on event. This function is mostly for *TPM\_Manufacture()* to access NV memory without a power on event

Return Value	Meaning
0	if success
non-0	if fail

```
20  int
21  _plat__NVEnable(
22      void    *platParameter           // IN: platform specific parameters
23  );
```

**C.8.5.2. `_plat__NVDisable()`**

Disable platform NV memory NV memory is automatically disabled at power off event. This function is mostly for *TPM\_Manufacture()* to disable NV memory without a power off event

```
24  void
25  _plat__NVDisable(void);
```

**C.8.5.3. `_plat__IsNvAvailable()`**

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
26  int
27  _plat__IsNvAvailable(void);
```

**C.8.5.4. `_plat__NvCommit()`**

Update NV chip

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
28  int
29  _plat__NvCommit(void);
```

**C.8.5.5. `_plat__NvMemoryRead()`**

Read a chunk of NV memory

```
30  void
31  _plat__NvMemoryRead(
32      unsigned int    startOffset,           // IN: read start
33      unsigned int    size,                 // IN: size of bytes to read
34      void            *data                 // OUT: data buffer
35  );
```

**C.8.5.6. `_plat__NvIsDifferent()`**

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE	the NV location is different from the test value
FALSE	the NV location is the same as the test value

```
36  BOOL
37  _plat__NvIsDifferent(
38      unsigned int    startOffset,           // IN: read start
39      unsigned int    size,                 // IN: size of bytes to compare
40      void            *data                 // IN: data buffer
41  );
```

**C.8.5.7. `_plat__NvMemoryWrite()`**

Write a chunk of NV memory

```

42 void
43 _plat__NvMemoryWrite(
44     unsigned int    startOffset,           // IN: read start
45     unsigned int    size,                 // IN: size of bytes to read
46     void            *data                 // OUT: data buffer
47 );

```

#### C.8.5.8. \_plat\_\_NvMemoryMove()

Move a chunk of NV memory from source to destination This function should ensure that if there overlap, the original data is copied before it is written

```

48 void
49 _plat__NvMemoryMove(
50     unsigned int    sourceOffset,         // IN: source offset
51     unsigned int    destOffset,          // IN: destination offset
52     unsigned int    size                 // IN: size of data being moved
53 );

```

#### C.8.5.9. \_plat\_\_SetNvAvail()

Set the current NV state to available. This function is for testing purposes only. It is not part of the platform NV logic

```

54 void
55 _plat__SetNvAvail(void);

```

#### C.8.5.10. \_plat\_\_ClearNvAvail()

Set the current NV state to unavailable. This function is for testing purposes only. It is not part of the platform NV logic

```

56 void
57 _plat__ClearNvAvail(void);

```

### C.8.6. Locality Functions

#### C.8.6.1. \_plat\_\_LocalityGet()

Get the most recent command locality in locality value form

```

58 unsigned char
59 _plat__LocalityGet(void);

```

#### C.8.6.2. \_plat\_\_LocalitySet()

Set the most recent command locality in locality value form

```

60 void
61 _plat__LocalitySet(
62     unsigned char    locality
63 );

```

**C.8.7. Clock Constants and Functions**

Assume that the nominal divisor is 30000

```
64 #define    CLOCK_NOMINAL        30000
```

A 1% change in rate is 300 counts

```
65 #define    CLOCK_ADJUST_COARSE  300
```

A . 1 change in rate is 30 counts

```
66 #define    CLOCK_ADJUST_MEDIUM  30
```

A minimum change in rate is 1 count

```
67 #define    CLOCK_ADJUST_FINE    1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16. 7%)

```
68 #define    CLOCK_ADJUST_LIMIT   5000
```

**C.8.7.1. `_plat__ClockReset()`**

This function sets the current clock time as initial time. This function is called at a power on event to reset the clock

```
69 void
70 _plat__ClockReset(void);
```

**C.8.7.2. `_plat__ClockTimeFromStart()`**

Function returns the compensated time from the start of the command when `_plat__ClockTimeFromStart()` was called.

```
71 unsigned long long
72 _plat__ClockTimeFromStart(
73     void
74 );
```

**C.8.7.3. `_plat__ClockTimeElapsed()`**

Get the time elapsed from current to the last time the `_plat__ClockTimeElapsed()` is called. For the first `_plat__ClockTimeElapsed()` call after a power on event, this call report the elapsed time from power on to the current call

```
75 unsigned long long
76 _plat__ClockTimeElapsed(void);
```

**C.8.7.4. `_plat__ClockAdjustRate()`**

Adjust the clock rate

```
77 void
78 _plat__ClockAdjustRate(
79     int          adjust           // IN: the adjust number. It could be
```

```
80                                     // positive or negative
81     );
```

## C.8.8. Entropy Constants and Functions

### C.8.8.1. `_plat_GetEntropy`

Returns the number of bytes of entropy generated

```
82     unsigned int
83     _plat_GetEntropy (
84         unsigned char* EntropyBuffer,    // IN/OUT: Buffer to receive the entropy.
85         unsigned int EntropySize        // IN: amount of entropy to generate. We
86                                         // assume that EntropyBuffer is big enough
87                                         // to receive it.
88     );
```

## C.8.9. Failure Mode

### C.8.9.1. `_plat__TpmFail`

Put TPM to failure mode

```
89     int
90     _plat__TpmFail(const char *function, int line, int code);
91 #endif
```



## C.9 PlatformData.h

### C.9.1. Description

This file contains the instance data for the Platform module. It is collected in this file so that the state of the module is easier to manage.

```

1  #ifndef _PLATFORM_DATA_H_
2  #define _PLATFORM_DATA_H_
3  #include "Implementation.h"
4  #include "bool.h"

```

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

```

5  extern BOOL      s_isCanceled;

```

From Clock.c This variable records the time when `_plat_ClockReset()` is called. This mechanism allow us to subtract the time when TPM is power off from the total time reported by `clock()` function

```

6  extern unsigned long long  s_initClock;
7  extern unsigned int        s_adjustRate;

```

From LocalityPlat.c Locality of current command

```

8  extern unsigned char s_locality;

```

From NVMem.c Choose if the NV memory should be backed by RAM or by file. If this macro is defined, then a file is used as NV. If it is not defined, then RAM is used to back NV memory. Comment out to use RAM.

```

9  #define FILE_BACKED_NV
10 #if defined FILE_BACKED_NV
11 #include <stdio.h>

```

A file to emulate NV storage

```

12 extern FILE*      s_NVfile;
13 #endif
14 extern unsigned char  s_NV[NV_MEMORY_SIZE];
15 extern BOOL        s_NvIsAvailable;

```

From PPPlat.c Physical presence. It is initialized to FALSE

```

16 extern BOOL      s_physicalPresence;
17 #endif // _PLATFORM_DATA_H_

```

## C.10 PlatformData.c

### C.10.1. Description

This file will instance the TPM variables that are not stack allocated. The descriptions for these variables is in Global.h for this project.

### C.10.2. Includes

This include is required to set the NV memory size consistently across all parts of the implementation.

```
1 #include "Implementation.h"
2 #include "Platform.h"
3 #include "PlatformData.h"
```

From Cancel.c

```
4 BOOL s_isCanceled = FALSE;
```

From Clock.c

```
5 unsigned long long s_initClock = 0;
6 unsigned int s_adjustRate = CLOCK_NOMINAL;
```

From LocalityPlat.c

```
7 unsigned char s_locality = 0;
```

From NVMem.c

```
8 #ifdef FILE_BACKED_NV
9 FILE *s_NVfile = NULL;
10 #endif
11 unsigned char s_NV[NV_MEMORY_SIZE];
12 BOOL s_NvIsAvailable = TRUE;
```

From PPPlat.c

```
13 BOOL s_physicalPresence;
```

## C.11 PPPlat.c

### C.11.1. Description

This module simulates the physical present interface pins on the TPM.

### C.11.2. Includes

```
1 #include "PlatformData.h"
```

### C.11.3. Functions

#### C.11.3.1. `_plat__PhysicalPresenceAsserted()`

Check if physical presence is signaled

Return Value	Meaning
TRUE	if physical presence is signaled
FALSE	if physical presence is not signaled

```
2  BOOL
3  _plat__PhysicalPresenceAsserted(void)
4  {
5      // Do not know how to check physical presence without real hardware.
6      // so always return TRUE;
7      return s_physicalPresence;
8  }
```

#### C.11.3.2. `_plat__Signal_PhysicalPresenceOn()`

Signal physical presence on

```
9  void
10 _plat__Signal_PhysicalPresenceOn(void)
11 {
12     s_physicalPresence = TRUE;
13     return;
14 }
```

#### C.11.3.3. `_plat__Signal_PhysicalPresenceOff()`

Signal physical presence off

```
15 void
16 _plat__Signal_PhysicalPresenceOff( void)
17 {
18     s_physicalPresence = FALSE;
19     return;
20 }
```

## C.12 TpmFail.c

### C.12.1. Description

This file contains the function that is called when the TPM experiences a fatal error. This function is stubbed out. It should be replaced with a function that will save the calling parameters so that they may be returned on a subsequent TPM2\_GetTestResult(). The function should then clean the stack (as much as possible), set the flag to indicate that the TPM is in failure mode, and return TPM\_RC\_FAIL.

```
1  #include "assert.h"
```

### C.12.2. \_plat\_\_TpmFail()

```
2  int
3  _plat__TpmFail(const char *function, int line, int code)
4  {
5  // These lines are added to keep the compiler from complaining about no reference to
6  // the formal parameter
7      char    a = *function;
8      int     l = line;
9      int     c = code;
10     a += 1;
11     l += 1;
12     c += 1;
13 // LAST JUNK LINE
14
15     assert(0);
16     return 0;
17 }
```

**Annex D**  
(informative)  
**Remote Procedure Interface**

## D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as `_TPM_HashStart`.

## D.2 TpmTcpProtocol.h

```

1  #ifndef      TCP_TPM_PROTOCOL_H
2  #define      TCP_TPM_PROTOCOL_H

```

### D.2.1. Introduction

TPM commands are communicated as BYTE streams on a TCP connection. The TPM command protocol is enveloped with the interface protocol described in this file. The command is indicated by a UINT32 with one of the values below. Most commands take no parameters return no TPM errors. In these cases the TPM interface protocol acknowledges that command processing is complete by returning a UINT32=0. The command TPM\_SIGNAL\_HASH\_DATA takes a UINT32-prepended variable length BYTE array and the interface protocol acknowledges command completion with a UINT32=0. Most TPM commands are enveloped using the TPM\_SEND\_COMMAND interface command. The parameters are as indicated below. The interface layer also appends a UIN32=0 to the TPM response for regularity.

### D.2.2. Typedefs

TPM Commands. All commands acknowledge processing by returning a UINT32 == 0 except where noted

```

3  #define TPM_SIGNAL_POWER_ON          1
4  #define TPM_SIGNAL_POWER_OFF        2
5  #define TPM_SIGNAL_PHYS_PRE_ON      3
6  #define TPM_SIGNAL_PHYS_PRE_OFF    4
7  #define TPM_SIGNAL_HASH_START      5
8  #define TPM_SIGNAL_HASH_DATA       6
9      // {UINT32 BufferSize, BYTE[BufferSize] Buffer}
10 #define TPM_SIGNAL_HASH_END         7
11 #define TPM_SEND_COMMAND            8
12      // {BYTE Locality, UINT32 InBufferSize, BYTE[InBufferSize] InBuffer} ->
13      //      {UINT32 OutBufferSize, BYTE[OutBufferSize] OutBuffer}
14 #define TPM_SIGNAL_CANCEL_ON        9
15 #define TPM_SIGNAL_CANCEL_OFF      10
16 #define TPM_SIGNAL_NV_ON           11
17 #define TPM_SIGNAL_NV_OFF          12
18 #define TPM_REMOTE_HANDSHAKE       15
19 #define TPM_SET_ALTERNATIVE_RESULT  16
20 #define TPM_SHUTDOWN               20
21 enum TpmEndPointInfo
22 {
23     tpmPlatformAvailable = 0x01,
24     tpmUsesTbs = 0x02,
25     tpmInRawMode = 0x04,
26     tpmSupportsPP = 0x08
27 };
28
29 // Existing RPC interface type definitions retained so that the implementation
30 // can be re-used
31 typedef struct
32 {
33     unsigned long BufferSize;
34     unsigned char *Buffer;
35 } _IN_BUFFER;
36
37 typedef unsigned char *_OUTPUT_BUFFER;
38
39 typedef struct
40 {

```

```
41     unsigned long BufferSize;
42     OUTPUT_BUFFER Buffer;
43 } _OUT_BUFFER;
44
45 /** TPM Command Function Prototypes
46 void
47 _rpc__Signal_PowerOn();
48 void
49 _rpc__Signal_PowerOff();
50 void
51 _rpc__Signal_PhysicalPresenceOn();
52 void
53 _rpc__Signal_PhysicalPresenceOff();
54 void
55 _rpc__Signal_Hash_Start();
56 void
57 _rpc__Signal_Hash_Data(
58     _IN_BUFFER input
59 );
60 void
61 _rpc__Signal_HashEnd();
62 void
63 _rpc__Send_Command(
64     unsigned char    locality,
65     _IN_BUFFER      request,
66     _OUT_BUFFER      *response
67 );
68 void
69 _rpc__Signal_CancelOn();
70 void
71 _rpc__Signal_CancelOff();
72 void
73 _rpc__Signal_NvOn();
74 void
75 _rpc__Signal_NvOff();
```

start the TPM server on the indicated socket. The TPM is single-threaded and will accept connections first-come-first-served. Once a connection is dropped another client can connect.

```
76 BOOL TpmServer(SOCKET ServerSocket);
77 #endif
```



## D.3 TcpServer.c

### D.3.1. Description

This file contains the socket interface to a TPM simulator.

### D.3.2. Includes and Function Prototypes

```

1  #include <stdio.h>
2  #include <windows.h>
3  #include <winsock.h>
4  #include "string.h"
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <assert.h>
8  #include "TpmTcpProtocol.h"
9  BOOL ReadBytes(SOCKET s, char* buffer, int NumBytes);
10 BOOL WriteBytes(SOCKET s, char* buffer, int NumBytes);
11 BOOL WriteUINT32(SOCKET s, UINT32 val);
12 static UINT32 ServerVersion = 1;

```

The input and output data buffers for the simulator.

```

13 #define MAX_BUFFER 1048576
14 char InputBuffer[MAX_BUFFER];
15 char OutputBuffer[MAX_BUFFER];

```

### D.3.3. Functions

#### D.3.3.1. CreateSocket()

Function creates a socket listening on *PortNumber*.

```

16 static int CreateSocket(int PortNumber, SOCKET *listenSocket)
17 {
18     WSADATA wsaData;
19     struct sockaddr_in MyAddress;
20
21     int res;
22
23     // Initialize Winsock
24     res = WSASStartup(MAKEWORD(2,2), &wsaData);
25     if (res != 0)
26     {
27         printf("WSASStartup failed with error: %d\n", res);
28         return -1;
29     }
30
31     // create listening socket
32     *listenSocket = socket(PF_INET, SOCK_STREAM, 0);
33     if(INVALID_SOCKET == *listenSocket)
34     {
35         printf("Cannot create server listen socket. Error is 0x%x\n",
36             WSAGetLastError());
37         return -1;
38     }
39
40     // bind the listening socket to the specified port
41     ZeroMemory(&MyAddress, sizeof(MyAddress));
42     MyAddress.sin_port=htons((short) PortNumber);

```

```

43     MyAddress.sin_family=AF_INET;
44
45     res= bind(*listenSocket, (struct sockaddr*) &MyAddress, sizeof(MyAddress));
46     if(res==SOCKET_ERROR)
47     {
48         printf("Bind error. Error is 0x%x\n", WSAGetLastError());
49         return -1;
50     };
51
52     // listen/wait for server connections
53     res= listen(*listenSocket,3);
54     if(res==SOCKET_ERROR)
55     {
56         printf("Listen error. Error is 0x%x\n", WSAGetLastError());
57         return -1;
58     };
59
60     return 0;
61 }

```

### D.3.3.2. PlatformServer()

This function processes incoming platform requests.

```

62 BOOL PlatformServer(SOCKET s)
63 {
64     UINT32 Command;
65     BOOL ok;
66
67     for(;;)
68     {
69         ok = ReadBytes(s, (char*) &Command, 4);
70         // client disconnected (or other error). We stop processing this client
71         // and return to our caller who can stop the server or listen for another
72         // connection.
73         if(!ok) return TRUE;
74         Command = ntohl(Command);
75         switch(Command)
76         {
77             case TPM_SIGNAL_POWER_ON:
78                 _rpc_Signal_PowerOn();
79                 break;
80
81             case TPM_SIGNAL_POWER_OFF:
82                 _rpc_Signal_PowerOff();
83                 break;
84
85             case TPM_SIGNAL_PHYS_PRE_ON:
86                 _rpc_Signal_PhysicalPresenceOn();
87                 break;
88
89             case TPM_SIGNAL_PHYS_PRE_OFF:
90                 _rpc_Signal_PhysicalPresenceOff();
91                 break;
92
93             case TPM_SIGNAL_CANCEL_ON:
94                 _rpc_Signal_CancelOn();
95                 break;
96
97             case TPM_SIGNAL_CANCEL_OFF:
98                 _rpc_Signal_CancelOff();
99                 break;
100
101             case TPM_SIGNAL_NV_ON:

```

```

102         _rpc_Signal_NvOn();
103         break;
104
105     case TPM_SIGNAL_NV_OFF:
106         _rpc_Signal_NvOff();
107         break;
108
109     case TPM_SHUTDOWN:
110         // Client signaled end-of-session
111         return TRUE;
112
113     default:
114         printf("Unrecognized platform interface command %d\n", Command);
115         return TRUE;
116     }
117     WriteUINT32(s,0);
118 }
119 return FALSE;
120 }

```

### D.3.3.3. PlatformSvcRoutine()

This function is called to set up the socket interfaces listen for commands.

```

121 DWORD WINAPI PlatformSvcRoutine(LPVOID port)
122 {
123     int PortNumber = (int)(INT_PTR) port;
124     SOCKET listenSocket, serverSocket;
125     struct sockaddr_in HerAddress;
126     int res, length;
127     BOOL continueServing;
128
129     res = CreateSocket(PortNumber, &listenSocket);
130     if(res != 0)
131     {
132         printf("Create platform service socket fail\n");
133         return res;
134     }
135
136     // Loop accepting connections one-by-one until we are killed or asked to stop
137     // Note the platform service is single-threaded so we don't listen for a new
138     // connection until the prior connection drops.
139     do
140     {
141         printf("Platform server listening on port %d\n", PortNumber);
142
143         // blocking accept
144         length = sizeof(HerAddress);
145         serverSocket = accept(listenSocket,
146                             (struct sockaddr*) &HerAddress,
147                             &length);
148         if(serverSocket == SOCKET_ERROR)
149         {
150             printf("Accept error. Error is 0x%x\n", WSAGetLastError());
151             return -1;
152         };
153         printf("Client accepted\n");
154
155         // normal behavior on client disconnection is to wait for a new client
156         // to connect
157         continueServing = PlatformServer(serverSocket);
158         closesocket(serverSocket);
159     }
160     while(continueServing);

```

```

161
162     return 0;
163 }

```

#### D.3.3.4. PlatformSignalService()

Start service for processing platform signals. This function starts a new thread waiting for platform signals. Platform signals are processed by a single thread in sequence.

```

164 int PlatformSignalService(int PortNumber)
165 {
166     HANDLE hPlatformSvc;
167     int ThreadId;
168     int port = PortNumber;
169
170     // Create service thread for platform signals
171     hPlatformSvc = CreateThread(NULL, 0,
172                               (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
173                               (LPVOID) (INT_PTR) port, 0, (LPDWORD)&ThreadId);
174     if(hPlatformSvc == NULL)
175     {
176         printf("Thread Creation failed\n");
177         return -1;
178     }
179
180     return 0;
181 }

```

#### D.3.3.5. RegularCommandService()

```

182 int RegularCommandService(int PortNumber)
183 {
184     SOCKET listenSocket;
185     SOCKET serverSocket;
186     struct sockaddr_in HerAddress;
187
188     int res, length;
189     BOOL continueServing;
190
191     res = CreateSocket(PortNumber, &listenSocket);
192     if(res != 0)
193     {
194         printf("Create platform service socket fail\n");
195         return res;
196     }
197
198     // Loop accepting connections one-by-one until we are killed or asked to stop
199     // Note the TPM command service is single-threaded so we don't listen for
200     // a new connection until the prior connection drops.
201     do
202     {
203         printf("TPM command server listening on port %d\n", PortNumber);
204
205         // blocking accept
206         length = sizeof(HerAddress);
207         serverSocket = accept(listenSocket,
208                              (struct sockaddr*) &HerAddress,
209                              &length);
210         if(serverSocket == SOCKET_ERROR)
211         {
212             printf("Accept error. Error is 0x%x\n", WSAGetLastError());
213             return -1;
214         }

```

```

215         printf("Client accepted\n");
216
217         // normal behavior on client disconnection is to wait for a new client
218         // to connect
219         continueServing = TpmServer(serverSocket);
220         closesocket(serverSocket);
221     }
222     while(continueServing);
223
224     return 0;
225 }

```

#### D.3.3.6. StartTcpServer()

Main entry-point. The server listens on port specified. Note that there is no way to specify the network interface in this implementation.

```

226 int StartTcpServer(int PortNumber)
227 {
228     int res;
229
230     // Start Platform Signal Processing Service
231     res = PlatformSignalService(PortNumber+1);
232     if (res != 0)
233     {
234         printf("PlatformSignalService failed\n");
235         return res;
236     }
237
238     // Start Regular/DRTM TPM command service
239     res = RegularCommandService(PortNumber);
240     if (res != 0)
241     {
242         printf("RegularCommandService failed\n");
243         return res;
244     }
245
246     return 0;
247 }

```

#### D.3.3.7. ReadBytes()

Read *NumBytes()* into buffer on indicated socket.

```

248 BOOL ReadBytes(SOCKET s, char* buffer, int NumBytes)
249 {
250     int res;
251     int numGot = 0;
252     while(numGot<NumBytes)
253     {
254         res = recv(s, buffer+numGot, NumBytes-numGot, 0);
255         if(res == -1)
256         {
257             printf("Receive error. Error is 0x%x\n", WSAGetLastError());
258             return FALSE;
259         }
260         if(res==0)
261         {
262             return FALSE;
263         }
264         numGot+=res;
265     }
266     return TRUE;

```

267 }

### D.3.3.8. WriteBytes()

Send *NumBytes*() on indicated socket

```

268 BOOL WriteBytes(SOCKET s, char* buffer, int NumBytes)
269 {
270     int res;
271     int numSent = 0;
272     while(numSent < NumBytes)
273     {
274         res = send(s, buffer+numSent, NumBytes-numSent, 0);
275         if(res == -1)
276         {
277             if(WSAGetLastError() == 0x2745)
278             {
279                 printf("Client disconnected\n");
280             }
281             else
282             {
283                 printf("Send error. Error is 0x%x\n", WSAGetLastError());
284             }
285             return FALSE;
286         }
287         numSent+=res;
288     }
289     return TRUE;
290 }

```

### D.3.3.9. WriteUINT32()

Send one byte containing htonl(1)

```

291 BOOL WriteUINT32(SOCKET s, UINT32 val)
292 {
293     UINT32 netVal = htonl(val);
294     return WriteBytes(s, (char*) &netVal, 4);
295 }

```

### D.3.3.10. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order

```

296 BOOL ReadVarBytes(SOCKET s, char* buffer, UINT32* BytesReceived, int MaxLen)
297 {
298     int length;
299     BOOL res;
300
301     res = ReadBytes(s, (char*) &length, 4);
302     if(!res) return res;
303     length = ntohl(length);
304     *BytesReceived = length;
305     if(length > MaxLen)
306     {
307         printf("Buffer too big. Client says %d\n", length);
308         return FALSE;
309     }
310     if(length == 0) return TRUE;
311     res = ReadBytes(s, buffer, length);
312     if(!res) return res;
313     return TRUE;

```

```
314 }
```

### D.3.3.11. WriteVarBytes()

Send a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order

```
315 BOOL WriteVarBytes(SOCKET s, char* buffer, int BytesToSend)
316 {
317     UINT32 netLength = htonl(BytesToSend);
318     BOOL res;
319
320     res = WriteBytes(s, (char*) &netLength, 4);
321     if(!res) return res;
322     res = WriteBytes(s, buffer, BytesToSend);
323     if(!res) return res;
324     return TRUE;
325 }
```

### D.3.3.12. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```
326 BOOL TpmServer(SOCKET s)
327 {
328     UINT32 length;
329     UINT32 Command;
330     BYTE locality;
331     BOOL ok;
332     BOOL WasDebugCommand=FALSE;
333     int result;
334     int clientVersion;
335     _IN_BUFFER InBuffer;
336     _OUT_BUFFER OutBuffer;
337
338     for(;;)
339     {
340         ok = ReadBytes(s, (char*) &Command, 4);
341         // client disconnected (or other error). We stop processing this client
342         // and return to our caller who can stop the server or listen for another
343         // connection.
344         if(!ok) return TRUE;
345         Command = ntohl(Command);
346         switch(Command)
347         {
348             case TPM_SIGNAL_HASH_START:
349                 _rpc_Signal_Hash_Start();
350                 WriteUINT32(s,0);
351                 break;
352
353             case TPM_SIGNAL_HASH_END:
354                 _rpc_Signal_Hash_End();
355                 WriteUINT32(s,0);
356                 break;
357
358             case TPM_SIGNAL_HASH_DATA:
359                 ok = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
360                 if(!ok) return TRUE;
361                 InBuffer.Buffer = (BYTE*) InputBuffer;
362                 InBuffer.BufferSize = length;
363                 _rpc_Signal_Hash_Data(InBuffer);
364                 WriteUINT32(s,0);
365                 break;
366         }
```

```

367     case TPM_SEND_COMMAND:
368         ok = ReadBytes(s, (char*) &locality, 1);
369         if(!ok) return TRUE;
370
371         ok = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
372         if(!ok) return TRUE;
373         InBuffer.Buffer = (BYTE*) InputBuffer;
374         InBuffer.BufferSize = length;
375         OutBuffer.BufferSize = MAX_BUFFER;
376         OutBuffer.Buffer = (_OUTPUT_BUFFER) OutputBuffer;
377         _rpc_Send_Command(locality, InBuffer, &OutBuffer);
378         ok = WriteVarBytes(s, (char*) OutBuffer.Buffer, OutBuffer.BufferSize);
379         if(!ok) return TRUE;
380         ok = WriteUINT32(s, 0);
381         if(!ok) return TRUE;
382         break;
383
384     case TPM_REMOTE_HANDSHAKE:
385         ok = ReadBytes(s, (char*)&clientVersion, 4);
386         if(!ok) return TRUE;
387         if( clientVersion == 0 )
388         {
389             printf("Unsupported client version (0).\n");
390             return TRUE;
391         }
392         ok &= WriteUINT32(s, ServerVersion);
393         ok &= WriteUINT32(s, tpmInRawMode | tpmPlatformAvailable |
tpmSupportsPP);
394         ok &= WriteUINT32(s, 0);
395         if(!ok) return TRUE;
396         break;
397
398     case TPM_SET_ALTERNATIVE_RESULT:
399         ok = ReadBytes(s, (char*)&result, 4);
400         if(!ok) return TRUE;
401         ok = WriteUINT32(s, 0);
402         if(!ok) return TRUE;
403         // Alternative result is not applicable to the simulator.
404         break;
405
406     case TPM_SHUTDOWN:
407         // Client signaled end-of-session
408         return TRUE;
409
410     default:
411         printf("Unrecognized TPM interface command. Client says %d\n",
412             Command);
413         return TRUE;
414     }
415 }
416 return FALSE;
417 }

```



## D.4 TPMCmdp.c

### D.4.1. Description

This file contains the functions that process the commands received on the control port or the command port of the simulator. The control port is used to allow simulation of hardware events (such as, `_TPM_Hash_Start()`) to test the simulated TPM's reaction to those events. This improves code coverage of the testing.

### D.4.2. Includes and Data Definitions

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <assert.h>
4  #include <<K>>bool.h>
5  #include <TPMLib.h>
6  #include <platform.h>
7  #include <windows.h>
8  #include "TpmTcpProtocol.h"
9  static BOOL      s_isPowerOn = FALSE;
```

### D.4.3. Functions

#### D.4.3.1. Signal\_PowerOn()

Signal a power on event.

```
10 void
11 _rpc__Signal_PowerOn()
12 {
13     if(s_isPowerOn) return;
14
15     // Pass power on signal to platform
16     _plat__Signal_PowerOn();
17
18     // Pass power on signal to TPM
19     _TPM_Init();
20
21     // Set state as power on
22     s_isPowerOn = TRUE;
23 }
```

#### D.4.3.2. Signal\_PowerOff()

Signal a power off event.

```
24 void
25 _rpc__Signal_PowerOff()
26 {
27     if(!s_isPowerOn) return;
28
29     // Pass power off signal to platform
30     _plat__Signal_PowerOff();
31
32     s_isPowerOn = FALSE;
33
34     return;
35 }
```

**D.4.3.3. \_rpc\_\_Signal\_PhysicalPresenceOn()**

Function to simulate activation of the physical presence **pin**.

```

36 void
37 _rpc__Signal_PhysicalPresenceOn()
38 {
39     // If TPM is power off, reject this signal
40     if(!s_isPowerOn) return;
41
42     // Pass physical presence on to platform
43     _plat__Signal_PhysicalPresenceOn();
44
45     return;
46 }

```

**D.4.3.4. \_rpc\_\_Signal\_PhysicalPresenceOff()**

Function to simulate deactivation of the physical presence **pin**.

```

47 void
48 _rpc__Signal_PhysicalPresenceOff()
49 {
50     // If TPM is power off, reject this signal
51     if(!s_isPowerOn) return;
52
53     // Pass physical presence off to platform
54     _plat__Signal_PhysicalPresenceOff();
55
56     return;
57 }

```

**D.4.3.5. \_rpc\_\_Signal\_Hash\_Start()**

Function to simulate a \_TPM\_Hash\_Start() event.

```

58 void
59 _rpc__Signal_Hash_Start()
60 {
61     // If TPM is power off, reject this signal
62     if(!s_isPowerOn) return;
63
64     // Pass _TPM_Hash_Start signal to TPM
65     Signal_Hash_Start();
66     return;
67 }

```

**D.4.3.6. \_rpc\_\_Signal\_Hash\_Data()**

Function to simulate a \_TPM\_Hash\_Data() event.

```

68 void
69 _rpc__Signal_Hash_Data(
70     _IN_BUFFER input
71 )
72 {
73     // If TPM is power off, reject this signal
74     if(!s_isPowerOn) return;
75
76     // Pass _TPM_Hash_Data signal to TPM
77     Signal_Hash_Data(input.BufferSize, input.Buffer);

```

```

78     return;
79 }

```

#### D.4.3.7. `_rpc__Signal_HashEnd()`

Function to simulate a `_TPM_Hash_End()` event.

```

80 void
81 _rpc__Signal_HashEnd()
82 {
83     // If TPM is power off, reject this signal
84     if(!s_isPowerOn) return;
85
86     // Pass _TPM_HashEnd signal to TPM
87     Signal_Hash_End();
88     return;
89 }

```

#### D.4.3.8. `_rpc__Send_Command()`

This is the TPM command interface.

```

90 void
91 _rpc__Send_Command(
92     unsigned char    locality,
93     _IN_BUFFER      request,
94     _OUT_BUFFER     *response
95 )
96 {
97     // If TPM is power off, reject any commands.
98     if(!s_isPowerOn)
99     {
100         response->BufferSize = 0;
101         return;
102     }
103
104     // Set command locality. Command locality is a signal rather than a part
105     // of TPM internal state. So we always set the locality information even
106     // the command may fail
107     _plat__LocalitySet(locality);
108
109     // Call command execution
110     // response buffer space is provided by the called function.
111     ExecuteCommand(request.BufferSize, request.Buffer,
112                   &response->BufferSize, &response->Buffer);
113     if(response->BufferSize == 10 && response->Buffer[9] != 0)
114         response->Buffer[6] = 0;
115
116     return;
117 }
118 }

```

#### D.4.3.9. `_rpc__Signal_CancelOn()`

Function to turn on the indication to cancel a command in process.

```

119 void
120 _rpc__Signal_CancelOn()
121 {
122     // If TPM is power off, reject this signal
123     if(!s_isPowerOn) return;

```

```

124
125     // Set the platform canceling flag.
126     _plat__SetCancel();
127
128     return;
129 }

```

#### D.4.3.10. \_rpc\_\_Signal\_CancelOff()

Function to turn off the indication to cancel a command in process.

```

130 void
131 _rpc__Signal_CancelOff()
132 {
133     // If TPM is power off, reject this signal
134     if(!s_isPowerOn) return;
135
136     // Set the platform canceling flag.
137     _plat__ClearCancel();
138
139     return;
140 }

```

#### D.4.3.11. \_rpc\_\_Signal\_NvOn()

In a system where the NV memory used by the TPM is not within the TPM, the NV may not always be available. This function turns on the indicator that indicates that NV is available.

```

141 void
142 _rpc__Signal_NvOn()
143 {
144     // If TPM is power off, reject this signal
145     if(!s_isPowerOn) return;
146
147     _plat__SetNvAvail();
148     return;
149 }

```

#### D.4.3.12. \_rpc\_\_Signal\_NvOff()

This function set the indication that NV memory is no longer available.

```

150 void
151 _rpc__Signal_NvOff()
152 {
153     // If TPM is power off, reject this signal
154     if(!s_isPowerOn) return;
155
156     _plat__ClearNvAvail();
157     return;
158 }

```

#### D.4.3.13. \_rpc\_\_Shutdown()

This function is used to stop the TPM simulator.

```

159 void
160 _rpc__Shutdown()
161 {
162     RPC_STATUS status;

```

```
163
164     // Stop TPM
165     TPM_TearDown();
166
167     status = RpcMgmtStopServerListening(NULL);
168     if (status != RPC_S_OK)
169     {
170         printf_s("RpcMgmtStopServerListening returned: 0x%x\n", status);
171         exit(status);
172     }
173
174     status = RpcServerUnregisterIf(NULL, NULL, FALSE);
175     if (status != RPC_S_OK)
176     {
177         printf_s("RpcServerUnregisterIf returned 0x%x\n", status);
178         exit(status);
179     }
180 }
```

## D.5 TPMCmds.c

### D.5.1. Description

This file contains the entry point for the simulator.

### D.5.2. Includes, Defines, Data Definitions, and Function Prototypes

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <ctype.h>
4  #include <windows.h>
5  #include <strsafe.h>
6  #include "string.h"
7  #include "TpmTcpProtocol.h"
8  #define PURPOSE \
9  "TPM Reference Simulator.\nCopyright Microsoft 2010, 2011.\n"
10 #define DEFAULT_TPM_PORT 2321
11 void* MainPointer;
12 int TPM_Manufacture();
13 int _plat_NVEnable(void* platParameters);
14 void _plat_NVDisable();
15 int StartTcpServer(int PortNumber);

```

### D.5.3. Functions

#### D.5.3.1. Usage()

This function prints the proper calling sequence for the simulator.

```

16 void Usage(char * pszProgramName)
17 {
18     fprintf_s(stderr, "%s", PURPOSE);
19     fprintf_s(stderr, "Usage:\n");
20     fprintf_s(stderr, "%s          - Starts the TPM server listening on port %d\n",
21             pszProgramName, DEFAULT_TPM_PORT);
22     fprintf_s(stderr,
23             "%s PortNum - Starts the TPM server listening on port PortNum\n",
24             pszProgramName);
25     fprintf_s(stderr, "%s ?          - This message\n", pszProgramName);
26     exit(1);
27 }

```

#### D.5.3.2. main()

Entry point for the simulator.

main: register the interface, start listening for clients

```

28 void __cdecl main(int argc, char * argv[])
29 {
30     RPC_STATUS status;
31     int portNum = DEFAULT_TPM_PORT;
32     if(argc>2)
33     {
34         Usage(argv[0]);
35     }
36
37     if(argc==2)

```

```
38     {
39         if(strcmp(argv[1], "?") ==0)
40         {
41             Usage(argv[0]);
42         }
43         portNum = atoi(argv[1]);
44         if(portNum <=0 || portNum>65535)
45         {
46             Usage(argv[0]);
47         }
48     }
49     _plat__NVEnable(NULL);
50     if(TPM_Manufacture() != 0)
51     {
52         status = RPC_S_INTERNAL_ERROR;
53         exit(status);
54     }
55     // Disable NV memory
56     _plat__NVDisable();
57
58     StartTcpServer(portNum);
59     return;
60 }
```