Part 4: Supporting Routines

Trusted Platform Module Library

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

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

Where:

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

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

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

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

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

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

The function prototype for a union has the form:

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

where:

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

4.2.2.3 Null Types

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

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

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

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

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:
4.2.3.1 Data Types

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:

```c
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:

```c
UINT16 TYPE.Array.Marshal(TYPE *source, BYTE **buffer, INT32 *size, INT32 count);
```

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

4.3 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 0th octet of the structure in which it is placed regardless of the size of the structure.

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

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

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

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

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

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

5.1 Introduction

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

5.2 bool.h

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

5.3 Capabilities.h

This file contains defines for the number of capability values that will fit into the largest data buffer. These defines are used in various function in the "support" and the "subsystem" code groups. A module that supports a type that is returned by a capability will have a function that returns the capabilities of the type.

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

```c
#ifndef _CAPABILITIES_H
#define _CAPABILITIES_H
#define MAX_CAP_DATA (MAX_CAP_BUFFER-sizeof(TPM_CAP)-sizeof(UINT32))
#define MAX_CAP_ALGS (MAX_CAP_DATA/sizeof(TPMS_ALG_PROPERTY))
#define MAX_CAP_HANDLES (MAX_CAP_DATA/sizeof(TPM_HANDLE))
#define MAX_CAP_CC (MAX_CAP_DATA/sizeof(TPM_CC))
#define MAX_TPM_PROPERTIES (MAX_CAP_DATA/sizeof(TPMS_TAGGED_PROPERTY))
#define MAX_PCR_PROPERTIES (MAX_CAP_DATA/sizeof(TPMS_TAGGED_PCR_SELECT))
#define MAX_ECC_CURVES (MAX_CAP_DATA/sizeof(TPM_ECC_CURVE))
#endif
```

5.4 TPMB.h

This file contains extra TPM2B structures

```c
#ifndef _TPMB_H
#define _TPMB_H
#include "TPM_Types.h"
#define TPMB_TYPE(name, bytes) \
typedef union { \
    struct { \
        UINT16 size; \
    }
```
Macro to instance and initialize a TPM2B value

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

A 2B structure for a seed

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

A 2B hash block

```c
TPM2B_TYPE(HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
```

```c
TPM2B_TYPE(RSA_PRIME, MAX_RSA_KEY_BYTES/2);
```

```c
TPM2B_TYPE(_BYTE_VALUE, 1);
```

```c
TPM2B_TYPE(_2_BYTE_VALUE, 2);
```

```c
TPM2B_TYPE(_4_BYTE_VALUE, 4);
```

```c
TPM2B_TYPE(_20_BYTE_VALUE, 20);
```

```c
TPM2B_TYPE(_32_BYTE_VALUE, 32);
```

```c
TPM2B_TYPE(_48_BYTE_VALUE, 48);
```

```c
TPM2B_TYPE(_64_BYTE_VALUE, 64);
```

```c
TPM2B_TYPE(_MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
```

```c
#endif
```

5.5 TpmError.h

```c
#define _TPM_ERROR_H
```

```c
#define FATAL_ERROR_ALLOCATION (1)
```

```c
#define FATAL_ERROR_DIVIDE_ZERO (2)
```

```c
#define FATAL_ERROR_INTERNAL (3)
```

```c
#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

```c
int plat__TpmFail(const char *function, int line, int code);
#define pAssert(a) (!!(a) || plat__TpmFail(__FUNCTION__, __LINE__, FATAL_ERROR_PARAMETER))
#define FAIL(a) (_plat__TpmFail(__FUNCTION__, __LINE__, a))
```

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

```c
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.

```c
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.

```c
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.

```c
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.

```c
20 typedef struct
21 {
```
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.

```c
typedef struct
{
    // The attributes field is required to be first followed by the publicArea.
    // This allows the overlay of the object structure and a sequence structure
    OBJECT_ATTRIBUTES attributes;   // object attributes
    TPMT_PUBLIC publicArea;         // public area of an object
    TPMT_SENSITIVE sensitive;      // sensitive area of an object

    #ifdef TPM_ALG_RSA
    TPM2B_PUBLIC_KEY_RSA privateExponent; // Additional field for the private
                                            // exponent of an RSA key.
    #endif

    TPM2B_NAME qualifiedName;       // object qualified name
    TPMI_DH_OBJECT evictHandle;     // if the object is an evict object,
                                    // the original handle is kept here.
                                    // The ‘working’ handle will be the
                                    // handle of an object slot.

    TPM2B_NAME name;                // Name of the object name. Kept here
                                    // to avoid repeatedly computing it.
} OBJECT;
```
5.6.5.4 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.

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

  // The data below is unique to a sequence object
  TPM2B_AUTH auth; // auth for use of sequence
  union
  {
    HASH_STATE hashState[HASH_COUNT];
    HMAC_STATE hmacState;
  }
} HASH_OBJECT;
```

5.6.5.5 ANY_OBJECT

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

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

5.6.6 AUTH_DUP Types

These values are used in the authorization processing.

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

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.

```c
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.
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 TPM2B_DIGEST cpHash; // the required cpHash value for the
175 // command being authorized
176 } u1;
177 // 'boundEntity' and 'cpHash' may
178 // share the same space to save memory
179 union
180 {
181 TPM2B_DIGEST auditDigest; // audit session digest
182 TPM2B_DIGEST policyDigest; // policyHash
183 } u2;
184 // audit log and policyHash may
185 // share space to save memory
186 } 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.

typedef struct
{
#define TPM_ALG_SHA1
192 BYTE sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
#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 // This counter increments whenever the PCR are updated.
207 // NOTE: A platform-specific specification may designate
208 // certain PCR changes as not causing this counter
209 // to increment.
210 UINT32 pcrCounter;
212
5.6.8.2 PCR_POLICY

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

```c
typedef struct {
    TPM_ALG_HASH       hashAlg[NUM_POLICY_PCR_GROUP];
    TPM2B_DIGEST        a;
    TPM2B_DIGEST        policy[NUM_POLICY_PCR_GROUP];
} PCR_POLICY;
```

5.6.8.3 PCR_AUTHVALUE

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

```c
typedef struct {
    TPM2B_DIGEST        auth[NUM_AUTHVALUE_PCR_GROUP];
} 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.

```
#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().

```c
typedef enum {
    SU_RESET,
    SU_RESTART,
    SU_RESUME
} 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 is method provides an imperfect analog to an actual NV implementation.

```c
typedef enum {
```
// Entries below mirror the PERSISTENT_DATA structure. These values are written
// to NV as individual items.

// hierarchy
NV_DISABLE_CLEAR, NV_OWNER_ALG,
NV_ENDORSEMENT_ALG, NV_OWNER_POLICY,
NV_ENDORSEMENT_POLICY, NV_OWNER_AUTH,
NV_ENDORSEMENT_AUTH, NV_LOCKOUT_AUTH,
NV_EP_SEED, NV_SP_SEED,
NV_PP_SEED,
NV_PH_PROOF, NV_SH_PROOF,
NV_EH_PROOF,

// Time
NV_TOTAL_RESET_COUNT, NV_RESET_COUNT,

// PCR
NV_PCR_POLICIES, NV_PCR_ALLOCATED,

// Physical Presence
NV_PP_LIST,

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

// Orderly State flag
NV_ORDERLY,

// Command Audit
NV_AUDIT_COMMANDS, NV_AUDIT_HASH_ALG,
NV_AUDIT_COUNTER,

// Algorithm Set
NV_ALGORITHM_SET,

NV_FIRMWARE_V1, NV_FIRMWARE_V2,

// The entries above are in PERSISTENT_DATA. The entries below represent
// structures that are read and written as a unit.

// ORDERLY_DATA data structure written on each orderly shutdown
NV_CLOCK,

// STATE_CLEAR_DATA structure written on each Shutdown(STATE)
NV_STATE_CLEAR,

// STATE_RESET_DATA structure written on each Shutdown(STATE)
NV_STATE_RESET,

NV_RESERVE_LAST // end of NV reserved data list
} 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.

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

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

```
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.

```
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().

```
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.

```
extern BOOL             g_phEnable;
```
Part 4: Supporting Routines

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).

```c
extern BOOL g_pceReConfig;
```

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().

```c
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().

```c
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.

```c
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.

```c
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.

```c
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()).

```c
typedef struct
{
    //************************************************************************************
    // Hierarchy
    //************************************************************************************
    // The values in this section are related to the hierarchies.
    BOOL disableClear; // TRUE if TPM2_Clear() using lockoutAuth is disabled

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

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

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

    // Hierarchy proofs
    TPM2B_AUTH phProof;
    TPM2B_AUTH shProof;
    TPM2B_AUTH ehProof;
    // Note there is a nullProof in the state_reset memory.

    //************************************************************************************
    // Reset Events
    //************************************************************************************
    // A count that increments at each TPM reset and never get reset during the life time of TPM. The value of this counter is initialized to 1 during TPM manufacture process.
    UINT64 totalResetCount;

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

    //************************************************************************************
    // PCR
    //************************************************************************************
    // This structure hold the policies for those PCR that have an update policy.
    // This implementation only supports a single group of PCR controlled by policy. If more are required, then this structure would be changed to an array
    PCR_POLICY pcrPolicies;

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

The PP_LIST type contains a bit map of the commands that require physical presence when the authorization is evaluated. Physical presence will be checked if the corresponding bit in the array is SET and if the authorization handle is TPM_RH_PLATFORM.

These bits may be changed with TPM2_PP_Commands().

```c
BYTE                ppList[((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7)/8];
```

// Dictionary attack values

These values are used for dictionary attack tracking and control.

```c
UINT32              failedTries;   // the current count of unexpired authorization failures
UINT32              maxTries;     // number of unexpired authorization failures before the TPM is in lockout
UINT32              recoveryTime; // time between authorization failures before failedTries is decremented
UINT32              lockoutRecovery; // time that must expire between authorization failures associated with lockoutAuth
BOOL                lockOutAuthEnabled; // TRUE if use of lockoutAuth is allowed
```

Orderly State

The orderly state for current cycle

```c
TPM_SU              orderlyState;
```

Command audit values.

```c
BYTE                auditComands[((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8];
TPMI_ALG_HASH       auditHashAlg;
UINT64              auditCounter;
```

Algorithm selection

```c
```

The 'algorithmSet' value indicates the collection of algorithms that are currently in use on the TPM. The interpretation of value is vendor dependent.

```c
UINT32              algorithmSet;
```

Firmware version

```c```

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

```c
```

In the reference implementation, if this value is printed as a hex
5.6.13.3 ORDERLY_DATA

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

```c
typedef struct orderly_data {
    //*****************************************************************************
    //           TIME
    //*****************************************************************************
    // Clock has two parts. One is the state save part and one is the NV part. The
    // state save version is updated on each command. When the clock rolls over, the
    // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
    // orderly way, then the sClock value is used to initialize the clock. If the
    // TPM shutdown was not orderly, then the persistent value is used and the safe
    // attribute is clear.
    UINT64          clock;         // The orderly version of clock
    TPMI_YES_NO     clockSafe;    // Indicates if the clock value is
                                      // safe.
} ORDERLY_DATA;

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).

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

    //*****************************************************************************
    //           PCR
    //*****************************************************************************
    // The set of PCR to be saved on Shutdown(STATE)
    PCR_SAVE        pcrSave;      // default reset is 0...0

    // This structure hold the authorization values for those PCR that have an
    // update authorization.
} STATE_CLEAR_DATA;
```
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.

5.6.13.5  State Reset Data

typedef struct state_reset_data
500 {
501     //************************************************************
502     //               Hierarchy Control
503     //************************************************************
504     TPM2B_AUTH         nullProof;   // The proof value associated with
505     // the TPM_RH_NULL hierarchy. The
506     // default reset value is from the RNG.
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     UINT32              clearCount;  // The default reset value is 0.
515     UINT64              objectContextID;  // This is the context ID for a saved
516     // object context. The default reset
517     // value is 0.
518     CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS]; // This is the value from which the
519     // 'contextID' is derived. The
520     // default reset value is {0}.
521     CONTEXT_COUNTER     contextCounter; // This array contains the
522     // values used to track the version
523     // numbers of saved contexts (see
524     // Session.c in for details). The
525     // default reset value is 0.
526
527     //***************************************************************************
528     //               Command Audit
529     //***************************************************************************
530     TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
// by TPM2_GetCommandAuditDigest() or a
// TPM Reset.

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

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

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

UINT32 pcrCounter; // This counter counts PCR updates. This counter is
// preserved across TPM Resume even though the PCR are not preserved.
// sessions remain active across TPM Restart and the count value in the
// is compared to this counter so this counter must have values that are unique
// as long as the sessions are active.
// NOTE: A platform-specific specification may designate that certain PCR changes
do not increment this counter to increment.

#ifdef TPM_ALG_ECC
//****************************************************************************
//         ECDAA
//****************************************************************************

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

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

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

#endif //TPM_ALG_ECC

} STATE_RESET_DATA;

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.

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

5.6.15 Private data

#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

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

Array of authorization session attributes

```c
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

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

Array of nonces provided by the caller for the corresponding sessions

```c
TPM2B_NONCE s_nonceCaller[MAX_SESSION_NUM];
```

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

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

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter

```c
extern UINT32 s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```c
extern UINT32 s_decryptSessionIndex;
```

Index of a session used for audit

```c
extern UINT32 s_auditSessionIndex;
```

The cpHash for an audit session

```c
extern TPM2B_DIGEST s_cpHashForAudit;
```

The cpHash for command audit

```c
#ifdef TPM_CC_GetCommandAuditDigest
extern TPM2B_DIGEST s_cpHashForCommandAudit;
#endif
```

Number of authorization sessions present in the command

```c
extern UINT32 s_sessionNum;
```

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

```c
extern BOOL s_DAPendingOnNV;
```

```c
#endif // SESSION_PROCESS_C
#ifdef 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.

```c
typedef struct
{
    BOOL        occupied;
    ANY_OBJECT      object;
} OBJECT_SLOT;
```

This is the memory that holds the loaded objects.

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

From PCR.c

```c
typedef struct
{
    #ifdef TPM_ALG_SHA1
        // SHA1 PCR
        BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
    #endif
    #ifdef TPM_ALG_SHA256
        // SHA256 PCR
        BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
    #endif
    #ifdef TPM_ALG_SHA384
        // SHA384 PCR
        BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
    #endif
    #ifdef TPM_ALG_SHA512
        // SHA512 PCR
        BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
    #endif
    #ifdef TPM_ALG_SM3_256
        // SHA256 PCR
        BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
    #endif
} PCR;
```

```c
typedef struct
{
    unsigned int stateSave : 1;    // if the PCR value should be saved in state save
    unsigned int resetLocality : 5; // The locality that the PCR can be reset
    unsigned int extendLocality : 5; // The locality that the PCR can be extend
} PCR_Attributes;
```

```c
extern PCR          s_pcrs[IMPLEMENTATION_PCR];
#endif // PCR_C
if defined SESSION_C || defined GLOBAL_C

From Session.c

Container for HMAC or policy session tracking information

```c
typedef struct
{
    BOOL                occupied;
    SESSION             session;    // session structure
} SESSION_SLOT;
```

```c
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.

```c
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)

```c
extern int s_freeSessionSlots;
```

From Manufacture.c

```c
extern BOOL s_manufactured;
```

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.

```c
extern BOOL s_initialized;
```

5.7 swap.h

```c
#ifndef _SWAP_H
#define _SWAP_H
#include <Implementation.h>
#endif
```

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

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

Disaggregate a UINT into a byte array

```c
#define UINT8_TO_BYTE_ARRAY(i, b)   ((b)[0] = (BYTE)(i), i)
#define UINT16_TO_BYTE_ARRAY(i, b)  ((b)[0] = (BYTE)((i) >>  8), (b)[1] = (BYTE) (i), 
```

Family "02"  Published  Page 25
Level 00 Revision 00.96  Copyright © TCG 2006-2013  March 15, 2013
#define UINT32_TO_BYTE_ARRAY(i, b)    ((b)[0] = (BYTE)((i) >> 24), \n   (b)[1] = (BYTE)((i) >> 16), \n   (b)[2] = (BYTE)((i) >>  8), \n   (b)[3] = (BYTE) (i), \n   (i))

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

#else

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

#define BYTE_ARRAY_TO_UINT8(b)        *((UINT8  *)(b))
#define BYTE_ARRAY_TO_UINT16(b)       *((UINT16 *)(b))
#define BYTE_ARRAY_TO_UINT32(b)       *((UINT32 *)(b))
#define BYTE_ARRAY_TO_UINT64(b)       *((UINT64 *)(b))

Disaggregate a UINT into a byte array

#define UINT8_TO_BYTE_ARRAY(i, b)   (*((UINT8  *)(b)) = (i))
#define UINT16_TO_BYTE_ARRAY(i, b)  (*((UINT16 *)(b)) = (i))
#define UINT32_TO_BYTE_ARRAY(i, b)  (*((UINT32 *)(b)) = (i))
#define UINT64_TO_BYTE_ARRAY(i, b)  (*((UINT64 *)(b)) = (i))
#endif // NO_AUTO_ALIGN == YES
#endif // _SWAP_H

5.8  InternalRoutines.h

#define INTERNAL_ROUTINES_H

Error Reporting

#include "TpmError.h"

NULL definition

#define NULL        (0)

UNUSED_PARAMETER

#define UNUSED_PARAMETER(param)     (void)(param);

Internal data definition

#include "Global.h"
#include "VendorString.h"

DRTM functions
#include "TPM_Hash_Start_fp.h"
#include "TPM_Hash_Data_fp.h"
#include "TPM_Hash_End_fp.h"

Internal subsystem functions

#include "Object_fp.h"
#include "Entity_fp.h"
#include "Hierarchy_fp.h"
#include "NV_fp.h"
#include "PCR_fp.h"
#include "DA_fp.h"

Internal support functions

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

Internal crypto functions

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

5.9 VendorString.h

#ifndef _VENDOR_STRING_H
#define _VENDOR_STRING_H

#define MANUFACTURER "%s"

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

#define MANUFACTURER "MSFT"

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

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

Define up to 4, 4-byte values. The values must each be 4 bytes long and the last value used may contain trailing zeros. These values define the response for TPM_PT_VENDOR_STRING_(1-4) in TPM2_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here. The vendor strings 2-4 may also be defined as appropriately.
#define VENDOR_STRING_1 "Micr"
#define VENDOR_STRING_2 "osof"
#define VENDOR_STRING_3 "t Co"
#define VENDOR_STRING_4 "rp."

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

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

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

#define FIRMWARE_V1 (0x20130118)

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

#define FIRMWARE_V2 (0x00093437)

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

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

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

```
#define "InternalRoutines.h"
#define "HandleProcess_fp.h"
#define "SessionProcess_fp.h"
#define "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.

```c
void ExecuteCommand(
    unsigned int requestSize, // IN: command buffer size
    unsigned char *request,  // IN: command buffer
    unsigned int *responseSize, // OUT: response buffer size
    unsigned char **response // OUT: response buffer
)
{
    // Command local variables
    TPM_ST tag;           // these first three variables are the
    UINT32 commandSize;   // pointer to the first byte of an
    TPM_CC commandCode = 0;
    BYTE *parmBufferStart; // optional parameter buffer
    UINT32 parmBufferSize = 0; // number of bytes in parameter area
    UINT32 handleNum = 0;   // number of handles unmarshaled into
    TPM_HANDLE handles[MAX_HANDLE_NUM]; // array to hold handles in the
                                        // command. Only handles in the handle
                                        // area are stored here, not handles
                                        // passed as parameters.

    // Response local variables
    TPM_RC result;         // return code for the command
    TPM_ST resTag;         // tag for the response
    UINT32 resHandleSize = 0; // size of the handle area in the
     resParmSize = 0; // the size of the response parameters
     resAuthSize = 0; // size of authorization area in the
     size;            // remaining data to be unmarshaled
     // or remaining space in the marshaling
     buffer            // buffer
    BYTE *buffer;          // pointer into the buffer being used
    UINT32 i;              // local temp

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

    // Set flags for NV access state. This should happen before any other
    // operation that may require a NV write.
```
60 
g_updateNV = FALSE;
61 
g_clearOrderly = FALSE;
62 
// Query platform to get the NV state. The result state is saved internally
63 // and will be reported by NvIsAvailable(). The reference code requires that
64 // accessibility of NV does not change during the execution of a command.
65 // Specifically, if NV is available when the command execution starts and then
66 // is not available later when it is necessary to write to NV, then the TPM
67 // will go into failure mode.
68 
NvCheckState();
69 
// Due to the limitations of the simulation, TPM clock must be explicitly
70 // synchronized with the system clock whenever a command is received.
71 // This function call is not necessary in a hardware TPM. However, taking
72 // a snapshot of the hardware timer at the beginning of the command allows
73 // the time value to be consistent for the duration of the command execution.
74 TimeUpdateToCurrent();
75 
// Any command through this function will unceremoniously end the
76 // TPM Hash_Data/_TPM Hash_End sequence.
77 if(g_DRTMHandle != TPM_RH_UNASSIGNED)
78 
ObjectTerminateEvent();
79 
// Get command buffer size and command buffer.
80 size = requestSize;
81 buffer = request;
82 
83 // First parse the tag. The unmarshaling routine will validate
84 // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
85 result = TPMI_ST_COMMAND_TAG_Unmarshal(&tag, &buffer, &size);
86 if(result != TPM_RC_SUCCESS)
87 
goto Cleanup;
88 
// Unmarshal the commandSize indicator.
89 result = UINT32_Unmarshal(&commandSize, &buffer, &size);
90 if(result != TPM_RC_SUCCESS)
91 
goto Cleanup;
92 
// On a TPM that receives bytes on a port, the number of bytes that were
93 // received on that port is requestSize it must be identical to commandSize.
94 // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
95 // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
96 // as the input processing (the function that receives the command bytes and
97 // places them in the input buffer) would likely have the input truncated when
98 // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
99 if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
100 
{ 
  result = TPM_RC_COMMAND_SIZE;
101 
goto Cleanup;
102 }
103 
// Unmarshal the command code.
104 result = TPM_CC_Unmarshal(&commandCode, &buffer, &size);
105 if(result != TPM_RC_SUCCESS)
106 
goto Cleanup;
107 
// Check to see if the command is implemented.
108 if(!CommandIsImplemented(commandCode))
109 
{ 
  result = TPM_RC_COMMAND_CODE;
110 
goto Cleanup;
111 }
112 
#if FIELD_UPGRADE_IMPLEMENTED == YES
113 // If the TPM is in FUM, then the only allowed command is
```c
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        UINT32 authorizationSize; // number of bytes in the session area
175
176        // Find out session buffer size.
177        result = UINT32_Unmarshal(&authorizationSize, &buffer, &size);
178        if(result != TPM_RC_SUCCESS)
179            goto Cleanup;
180
181        // Perform sanity check on the unmarshaled value. If it is smaller than
182        // the smallest possible session or larger than the remaining size of
183        // the command, then it is an error. NOTE: This check could pass but the
184        // session size could still be wrong. That will be determined after the
185        // sessions are unmarshaled.
186        if(    authorizationSize < 9
187            || authorizationSize > (UINT32) size)
188            {
189                result = TPM_RC_SIZE;
190                goto Cleanup;
191            }
```
sessionBufferStart = buffer;

// The parameters follow the session area.
parmBufferStart = sessionBufferStart + authorizationSize;

// Any data left over after removing the authorization sessions is param data. If the command does not have parameters, then an
// error will be returned if the remaining size is not zero. This is
// checked later.
parmBufferSize = size - authorizationSize;

// The actions of ParseSessionBuffer() are described in the introduction.
result = ParseSessionBuffer(commandCode, handleNum, handles, sessionBufferStart, authorizationSize, parmBufferStart, parmBufferSize);

if(result != TPM_RC_SUCCESS)
    goto Cleanup;
} else {

    parmBufferStart = buffer;
    parmBufferSize = size;

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

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

if(result != TPM_RC_SUCCESS)
    goto Cleanup;

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

Cleanup:

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

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

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

    // Adding parameter size field.
    if (tag == TPM_ST_SESSIONS)
        *responseSize += sizeof(UINT32);
    else
    {
        // The command failed.
        resTag = TPM_ST_NO_SESSIONS;
    }

    // Try to commit all the writes to NV if any NV write happened during this
    // command execution. This check should be made for both succeeded and failed
    // commands, because a failed one may trigger a NV write in DA logic as well.
    // This is the only place in the command execution path that may call the NV
    // commit. If the NV commit fails, the TPM should be put in failure mode.
    if (g_updateNV)
    {
        if (!NvCommit())
            FAIL(FATAL_ERROR_INTERNAL);
    }

    // Marshal the response header.
    buffer = MemoryGetResponseBuffer(commandCode);
    TPM_ST_Marshal(&resTag, &buffer, NULL);
    UINT32_Marshal((UINT32 *)responseSize, &buffer, NULL);
    pAssert(*responseSize <= MAX_RESPONSE_SIZE);
    TPM_RC_Marshal(&result, &buffer, NULL);

    *response = MemoryGetResponseBuffer(commandCode);

    // Clear unused bit in response buffer.
    MemorySet(*response + *responseSize, 0, MAX_RESPONSE_SIZE - *responseSize);

    return;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>handle is exempted from DA logic</td>
</tr>
<tr>
<td>FALSE</td>
<td>handle is not exempted from DA logic</td>
</tr>
</tbody>
</table>

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
15.             return (handle != TPM_RH_LOCKOUT);
16.             break;
17.         case TPM_HT_TRANSIENT:
18.             {
19.                 // When this function is called, a persistent object will have been loaded
20.                 // into an object slot and assigned a transient handle.
21.                 OBJECT *object;
22.                 object = ObjectGet(handle);
23.                 if(object->publicArea.objectAttributes.noDA == SET)
24.                     return TRUE;
25.                 break;
26.             }
27.         case TPM_HT_NV_INDEX:
28.             {
29.                 NV_INDEX    nvIndex;
30.                 NvGetIndexInfo(handle, &nvIndex);
31.                 break;
6.4.3.2 IncrementLockout()

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_AUTH_FAIL</td>
<td>authorization failure that caused DA lockout to increment</td>
</tr>
<tr>
<td>TPM_RC_BAD_AUTH</td>
<td>authorization failure did not cause DA lockout to increment</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
IncrementLockout(
    UINT32    sessionIndex
)
{
    TPM_HANDLE handle = s_associatedHandles[sessionIndex];
    TPM_HANDLE sessionHandle = s_sessionHandles[sessionIndex];
    TPM_RC    result;
    SESSION   *session = NULL;

    // Don't increment lockout unless the handle associated with the session
    // is DA protected or the session is bound to a DA protected entity.
    if(sessionHandle == TPM_RS_PW)
    {
        if(IsDAExempted(handle))
            return TPM_RC_BAD_AUTH;
    }
    else
    {
        session = SessionGet(sessionHandle);
        // If the session is bound to lockout, then use that as the relevant
        // handle. This means that an auth failure with a bound session
        // bound to lockoutAuth will take precedence over any other
        // lockout check
        if(session->attributes.isLockoutBound == SET)
            handle = TPM_RH_LOCKOUT;
        if(   session->attributes.isDaBound == CLEAR
            && IsDAExempted(handle)
        )
            // If the handle was changed to TPM_RH_LOCKOUT, this will not return
            // TPM_RC_BAD_AUTH
            return TPM_RC_BAD_AUTH;
    }
}
```
```c
if (handle == TPM_RH_LOCKOUT)
{
    pAssert(gp.lockOutAuthEnabled);
    gp.lockOutAuthEnabled = FALSE;
    // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
    // the lockout auth will be reset at startup.
    if (gp.lockoutRecovery != 0)
    {
        result = NvIsAvailable();
        if (result != TPM_RC_SUCCESS)
        {
            // No NV access for now. Put the TPM in pending mode.
            s_DAPendingOnNV = TRUE;
        }
        else
        {
            // Update NV.
            NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
            g_updateNV = TRUE;
        }
    }
    else
    {
        if (gp.recoveryTime != 0)
        {
            gp.failedTries++;
            result = NvIsAvailable();
            if (result != TPM_RC_SUCCESS)
            {
                // No NV access for now. Put the TPM in pending mode.
                s_DAPendingOnNV = TRUE;
            }
            else
            {
                // Record changes to NV.
                NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
                g_updateNV = TRUE;
            }
        }
    }
}
// Register a DA failure and reset the timers.
DAResisterFailure(handle);
return TPM_RC_AUTH_FAIL;
```

### 6.4.3.3 IsSessionBindEntity()

This function indicates if the entity associated with the handle is the entity, to which this session is bound. The binding would occur by making the bind parameter in TPM2_StartAuthSession() not equal to TPM_RH_NULL. The binding only occurs if the session is an HMAC session. The bind value is a combination of the Name and the authValue of the entity.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>handle points to the session start entity</td>
</tr>
<tr>
<td>FALSE</td>
<td>handle does not point to the session start entity</td>
</tr>
</tbody>
</table>

```c
static BOOL IsSessionBindEntity(
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>the command is an NV write operation</td>
</tr>
<tr>
<td>FALSE</td>
<td>the command is not an NV write operation</td>
</tr>
</tbody>
</table>

static BOOL IsWriteOperation(
    TPM_CC command_code
)
{
    switch(command_code)
    {
        case TPM_CC_NV_Write:
        case TPM_CC_NV_Increment:
        case TPM_CC_NV_SetBits:
        case TPM_CC_NV_Extend:
            return TRUE;
        default:
            return FALSE;
    }
}

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

```c
static BOOL
IsPolicySessionRequired(  
    TPM_CC commandCode,       // IN: command code
    UINT32 sessionIndex        // IN: session index
)
{
    AUTH_ROLE role = CommandAuthRole(commandCode, sessionIndex);
    TPM_HT type = HandleGetType(s_associatedHandles[sessionIndex]);

    if(role == AUTH_DUP)
        return TRUE;

    if(role == AUTH_ADMIN)
        {
            if(type == TPM_HT_TRANSIENT)
                {
                    OBJECT *object = ObjectGet(s_associatedHandles[sessionIndex]);
                    if(object->publicArea.objectAttributes.adminWithPolicy == CLEAR)
                        return FALSE;
                }
            return TRUE;
        }

    if(type == TPM_HT_PCR)
        {
            if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
                {
                    TPM2B_DIGEST policy;
                    TPMI_ALG_HASH policyAlg;
                    policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex], &policy);
                    if(policyAlg != TPM_ALG_NULL)
                        return TRUE;
                }
            return FALSE;
        }
```

### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>authValue is available</td>
</tr>
<tr>
<td>FALSE</td>
<td>authValue is not available</td>
</tr>
</tbody>
</table>

```c
static BOOL
IsAuthValueAvailable(  
    TPM_HANDLE handle,       // IN: handle of entity
)
TPM_CC commandCode,  // IN: commandCode
UINT32 sessionIndex  // IN: session index

// If a policy session is required, the entity can not be authorized by
// authValue. However, at this point, the policy session requirement should
// already have been checked.
pAssert(!IsPolicySessionRequired(commandCode, sessionIndex));

switch(HandleGetType(handle))
{
  case TPM_HT_PERMANENT:
    switch(handle)
    {
      // At this point hierarchy availability has already been
      // checked so primary seed handles are always available here
      case TPM_RH_OWNER:
      case TPM_RH_ENDORSEMENT:
      case TPM_RH_PLATFORM:
        return TRUE;
      break;
      case TPM_RH_LOCKOUT:
        // At the point when authValue availability is checked, control
        // path has already passed the DA check so LockOut auth is
        // always available here
        return TRUE;
      break;
      case TPM_RH_NULL:
        // NullAuth is always available.
        return TRUE;
      break;
      default:
        // Otherwise authValue is not available.
        return FALSE;
      break;
    }
  break;
  case TPM_HT_TRANSIENT:
    // A persistent object has already been loaded and the internal
    // handle changed.
    {
      OBJECT *object;
      object = ObjectGet(handle);

      // authValue is always available for a sequence object.
      if(ObjectIsSequence(object))
        return TRUE;
    }

    // authValue is available for an object if it has its sensitive
    // portion loaded and
    // 1. userWithAuth bit is SET, or
    // 2. ADMIN role is required
    if( object->attributes.publicOnly == CLEAR
      && (object->publicArea.objectAttributes.userWithAuth == SET
       || (CommandAuthRole(commandCode, sessionIndex) == AUTH_ADMIN
       && object->publicArea.objectAttributes.adminWithPolicy
       == CLEAR)))
      return TRUE;
    else
      return FALSE;
  }
  break;
  case TPM_HT_NV_INDEX:
    // NV Index.
    {
      NV_INDEX nvIndex;
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>authPolicy is available</td>
</tr>
<tr>
<td>FALSE</td>
<td>authPolicy is not available</td>
</tr>
</tbody>
</table>

```c
static BOOL IsAuthPolicyAvailable(
    TPM_HANDLE handle,
    TPM_CC    commandCode,
    UINT32    sessionIndex
)
{
    switch(HandleGetType(handle))
    {
    case TPM_HT_PERMANENT:
        switch(handle)
        {
        // At this point hierarchy availability has already been checked.
        case TPM_RH_OWNER:
            if (gp.ownerPolicy.t.size != 0)
                return TRUE;
            else
                return FALSE;
        case TPM_RH_ENDORSEMENT:
            if (gp.endorsementPolicy.t.size != 0)
                return TRUE;
            else
```
```c
return FALSE;

case TPM_RH_PLATFORM:
    if (gc.platformPolicy.t.size != 0)
        return TRUE;
    else
        return FALSE;
    default:
        // Otherwise, authPolicy is not available.
        return FALSE;
    break;

break;

case TPM_HT_TRANSIENT:
{
    // Object handle.
    // An evict object would already have been loaded and given a
    // transient object handle by this point.
    OBJECT *object = ObjectGet(handle);
    // Policy authorization is not available for an object with only
    // public portion loaded.
    if(object->attributes.publicOnly == SET)
        return FALSE;
    // Policy authorization is always available for an object but
    // is never available for a sequence.
    if(ObjectIsSequence(object))
        return FALSE;
    else
        return TRUE;
    break;
}

case TPM_HT_NV_INDEX:
    // An NV Index.
    {
        NV_INDEX nvIndex;
        NvGetIndexInfo(handle, &nvIndex);
        // If the policy size is not zero, check if policy can be used.
        if(nvIndex.publicArea.authPolicy.t.size != 0)
            {
                // If policy session is required for this handle, always
                // uses policy regardless of the attributes bit setting
                if(IsPolicySessionRequired(commandCode, sessionIndex))
                    return TRUE;
                // Otherwise, the presence of the policy depends on the NV
                // attributes.
                if(IsWriteOperation(commandCode))
                    {
                        if (nvIndex.publicArea.attributes.TPMA_NV_POLICYWRITE == SET)
                            return TRUE;
                        else
                            return FALSE;
                    }
                else
                    {
                        if (nvIndex.publicArea.attributes.TPMA_NV_POLICYREAD ==SET)
                            return TRUE;
                        else
                            return FALSE;
                    }
            }
        return FALSE;
    }

break;

    case TPM_HT_PCR:
    // PCR handle.
    if(PCRPolicyIsAvailable(handle))
```
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.

```c
static void ComputeCpHash(
    TPMI_ALG_HASH    hashAlg,    // IN: hash algorithm
    TPM_CC           commandCode, // IN: command code
    UINT32           handleNum,   // IN: number of handles
    TPM_HANDLE       handles[],   // IN: array of handles
    UINT32           parmBufferSize, // IN: size of input parameter area
    BYTE            *parmBuffer,  // IN: input parameter area
    TPM2B_DIGEST    *cpHash,      // OUT: cpHash
    TPM2B_DIGEST    *nameHash     // OUT: name hash of command
)
{
    UINT32 i;
    HASH STATE hashState;
    TPM2B_NAME name;

    // cpHash = hash(commandCode || authName1
    //    || authName2
    //    || authName 3 ]])
    // A cpHash can contain just a commandCode only if the lone session is
    // an audit session.
    // Start cpHash.
    cpHash->t.size = CryptStartHash(hashAlg, &hashState);

    // Add commandCode.
    CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
    // Add authNames for each of the handles.
    for(i = 0; i < handleNum; i++)
    {
        name.t.size = EntityGetName(handles[i], name.t.name);
        CryptUpdateDigest2B(&hashState, &name.b);
    }

    // Add the parameters.
    CryptUpdateDigest(&hashState, parmBufferSize, parmBuffer);

    // Complete the hash.
    CryptCompleteHash2B(&hashState, &cpHash->b);

    // If the nameHash is needed, compute it here.
    if(nameHash != NULL)
    {
```

// Start name hash. hashState may be reused.
nameHash->t.size = CryptStartHash(hashAlg, &hashState);

// Adding names.
for(i = 0; i < handleNum; i++)
{
    name.t.size = EntityGetName(handles[i], name.t.name);
    CryptUpdateDigest2B(&hashState, &name.b);
}
// Complete hash.
CryptCompleteHash2B(&hashState, &nameHash->b);
return;

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[].

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_AUTH_FAIL</td>
<td>auth fails and increments DA failure count</td>
</tr>
<tr>
<td>TPM_RC_BAD_AUTH</td>
<td>auth fails but DA does not apply</td>
</tr>
</tbody>
</table>

static TPM_RC
CheckPWAuthSession(
    UINT32          sessionIndex          // IN: index of session to be processed
)
{
    TPM2B_AUTH      authValue;
    TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
    // Strip trailing zeros from the password.
    MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
    // Get the auth value and size.
    authValue.t.size = EntityGetAuthValue(associatedHandle, authValue.t.buffer);
    // Success if the digests are identical.
    if(Memory2BEqual(&s_inputAuthValues[sessionIndex].b, &authValue.b))
    {
        return TPM_RC_SUCCESS;
    }
    else // if the digests are not identical
    {
        // Invoke DA protection if applicable.
        return IncrementLockout(sessionIndex);
    }
}

6.4.4.3 ComputeCommandHMAC()

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

static void
ComputeCommandHMAC(
    UINT32          sessionIndex,    // IN: index of session to be processed
    TPM2B_DIGEST   *cpHash,         // IN: cpHash
    TPM2B_DIGEST   *hmac            // OUT: authorization HMAC
)
```c
{  
  TPM2B_TYPE(KEY, (sizeof(TPMT_HA) * 2));
  TPM2B_KEY key;
  BYTE *marshalBuffer[sizeof(TPMA_SESSION)];
  BYTE *buffer;
  UINT32 marshalSize;
  HMAC_STATE hmacState;
  TPM2B_NONCE *nonceDecrypt;
  TPM2B_NONCE *nonceEncrypt;
  SESSION *session;

  nonceDecrypt = NULL;
  nonceEncrypt = NULL;

  // Determine if extra nonceTPM values are going to be required.
  // If this is the first session (sessionIndex = 0) and it is an authorization
  // session that uses an HMAC, then check if additional session nonces are to be
  // included.
  if (sessionIndex == 0
      && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
  {
    // If there is a decrypt session and if this is not the decrypt session,
    // then an extra nonce may be needed.
    if (s_decryptSessionIndex != UNDEFINED_INDEX
        && s_decryptSessionIndex != sessionIndex)
    {
      // Will add the nonce for the decrypt session.
      session = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
      nonceDecrypt = &session->nonceTPM;
    }
    // Now repeat for the encrypt session.
    if (s_encryptSessionIndex != UNDEFINED_INDEX
        && s_encryptSessionIndex != sessionIndex
        && s_encryptSessionIndex != s_decryptSessionIndex)
    {
      // Have to have the nonce for the encrypt session.
      session = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
      nonceEncrypt = &session->nonceTPM;
    }
  }

  // Continue with the HMAC processing.
  session = SessionGet(s_sessionHandles[sessionIndex]);
  // Generate HMAC key.
  MemoryCopy2B(&key.b, &session->sessionKey.b);
  // Check if the session has an associated handle and if the associated entity
  // is the one to which the session is bound. If not, add the authValue of
  // this entity to the HMAC key.
  // If the session is bound to the object or the session is a policy session
  // with no authValue required, do not include the authValue in the HMAC key.
  // Note: For a policy session, its isBound attribute is CLEARED.
  if (s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED
      && HandleGetType(s_sessionHandles[sessionIndex])
      == TPM_HT_POLICY_SESSION
      && session->attributes.isAuthValueNeeded == CLEAR
      && !IsSessionBindEntity(s_associatedHandles[sessionIndex], session))
  {
    key.t.size = key.t.size
      + EntityGetAuthValue(s_associatedHandles[sessionIndex],
        &key.b.buffer[key.b.size]);
  }

  // if the HMAC key size for a policy session is 0, a NULL string HMAC is
// allowed.
if((HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
    && key.t.size == 0
    && s_inputAuthValues[sessionIndex].t.size == 0)
{
    hmac->t.size = 0;
    return;
}

// Start HMAC
hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);

// Add cpHash
CryptUpdateDigest2B(&hmacState, &cpHash->b);

// Add nonceCaller
CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);

// Add nonceTPM
CryptUpdateDigest2B(&hmacState, &session->nonceTPM.b);

// If needed, add nonceTPM for decrypt session
if(nonceDecrypt != NULL)
    CryptUpdateDigest2B(&hmacState, &nonceDecrypt->b);

// If needed, add nonceTPM for encrypt session
if(nonceEncrypt != NULL)
    CryptUpdateDigest2B(&hmacState, &nonceEncrypt->b);

// Add sessionAttributes
buffer = marshalBuffer;
marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
    &buffer, NULL);
CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);

// Complete the HMAC computation
CryptCompleteHMAC2B(&hmacState, &hmac->b);

return;

6.4.4.4 CheckSessionHMAC()

This function checks the HMAC of a session. It uses ComputeCommandHMAC() to compute the expected HMAC value and then compares the result with the HMAC in the authorization session. The authorization is successful if they are the same.

If the authorizations are not the same, IncrementLockout() is called. It will return TPM_RC_AUTH_FAIL if the failure caused the failureCount to increment. Otherwise, it will return TPM_RC_BAD_AUTH.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_AUTH_FAIL</td>
<td>auth failure caused failureCount increment</td>
</tr>
<tr>
<td>TPM_RC_BAD_AUTH</td>
<td>auth failure did not cause failureCount increment</td>
</tr>
</tbody>
</table>

static TPM_RC
CheckSessionHMAC(
    UINT32 sessionIndex, // IN: index of session to be processed
    TPM2B_DIGEST *cpHash // IN: cpHash of the command
)
{
    TPM2B_DIGEST hmac; // authHMAC for comparing
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
615     return TPM_RC_SUCCESS;

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.

<table>
<thead>
<tr>
<th>Error Return</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_PCR_CHANGED</td>
<td>PCR value is not current</td>
</tr>
<tr>
<td>TPM_RC_POLICY_FAIL</td>
<td>policy session fails</td>
</tr>
<tr>
<td>TPM_RC_LOCALITY</td>
<td>command locality is not allowed</td>
</tr>
<tr>
<td>TPM_RC_POLICY_CC</td>
<td>CC doesn't match</td>
</tr>
<tr>
<td>TPM_RC_EXPIRED</td>
<td>policy session has expired</td>
</tr>
<tr>
<td>TPM_RC_PP</td>
<td>PP is required but not asserted</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available for write</td>
</tr>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting</td>
</tr>
</tbody>
</table>

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     TPMI_ALG_HASH    policyAlg;
629     UINT8            locality;
630
631     // Initialize pointer to the auth session.
632     session = SessionGet(s_sessionHandles[sessionIndex]);
// See if the PCR counter for the session is still valid.
if( !SessionPCRValueIsCurrent(s_sessionHandles[sessionIndex]) )
    return TPM_RC_PCR_CHANGED;

// Get authPolicy.
policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
    &authPolicy);

// Compare policy hash algorithm.
if(policyAlg != session->authHashAlg)
    return TPM_RC_POLICY_FAIL;

// Compare timeout.
if(session->timeOut != 0)
    {
        // Cannot compare time if clock stop advancing. An TPM_RC_NV_UNAVAIL
        // or TPM_RC_NV_RATE error may be returned here.
        result = NvIsAvailable();
        if(result != TPM_RC_SUCCESS)
            return result;

        if(session->timeOut < go.clock)
            return TPM_RC_EXPIRED;
    }

// If command code is provided it must match
if(session->commandCode != 0)
    {
        if(session->commandCode != commandCode)
            return TPM_RC_POLICY_CC;
    }
else
    {
        // If command requires a DUP or ADMIN authorization, the session must have
        // command code set.
        AUTH_ROLE   role = CommandAuthRole(commandCode, sessionIndex);
        if(role == AUTH_ADMIN || role == AUTH_DUP)
            return TPM_RC_POLICY_FAIL;
    }

// Check command locality.
BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
BYTE        *buffer = sessionLocality;

// Get existing locality setting in canonical form
TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);

// See if the locality has been set
if(sessionLocality[0] != 0)
    {
        // If so, get the current locality
        locality = _plat__LocalityGet();
        if (locality < 5)
            {  
                if(*(UINT8*)&session->commandLocality != 1 << locality)
                    return TPM_RC_LOCALITY;
            }
        else if (locality > 31)
            {  
                if(*(UINT8*)&session->commandLocality != locality)
                    return TPM_RC_LOCALITY;
            }
        else
            {
                pAssert(FALSE);
            }
699 } } // end of locality check
700 }
701 } // Check physical presence.
702 if (session->attributes.isPPRequired == SET
703 && !_plat__PhysicalPresenceAsserted())
704 return TPM_RC_PP;
705
706 // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
707 // DUP role for this handle.
708 if (session->u1.cpHash.b.size != 0)
709 {
710 if (session->attributes.iscpHashDefined)
711 {
712 if (!Memory2BEqual(&session->u1.cpHash.b, &cpHash.b))
713 return TPM_RC_POLICY_FAIL;
714 }
715 else if (!Memory2BEqual(&session->u1.cpHash.b, &nameHash.b))
716 return TPM_RC_POLICY_FAIL;
717 }
718 else // Compare authPolicy.
719 if (!Memory2BEqual(&session->u2.policyDigest.b, &authPolicy.b))
720 return TPM_RC_POLICY_FAIL;
721
722 return TPM_RC_SUCCESS;
723
6.4.4.6 RetrieveSessionData()
724
This function will unmarshal the sessions in the session area of a command. The values are placed in the
arrays that are defined at the beginning of this file. The normal unmarshaling errors are possible.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SUCCESS</td>
<td>unmarshaled without error</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>the number of bytes unmarshaled is not the same as the value for</td>
</tr>
<tr>
<td></td>
<td>authorizationSize in the command</td>
</tr>
</tbody>
</table>

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
for(sessionIndex = 0; bufferSize > 0; sessionIndex++)
{
    // If maximum allowed number of sessions has been parsed, exit the loop.
    if(sessionIndex == MAX_SESSION_NUM)
      break;

    // make sure that the associated handle for each session starts out
    // unassigned
    s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;

    // First parameter: Session handle.
    result = TPMI_SH_AUTH_SESSION_Unmarshal(&s_sessionHandles[sessionIndex],
                  &sessionBuffer, &bufferSize, TRUE);
    if(result != TPM_RC_SUCCESS)
      return result + TPM_RC_S + g_rcIndex[sessionIndex];

    // Second parameter: Nonece.
    result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
                  &sessionBuffer, &bufferSize);
    if(result != TPM_RC_SUCCESS)
      return result + TPM_RC_S + g_rcIndex[sessionIndex];

    // Third parameter: sessionAttributes.
    result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
                  &sessionBuffer, &bufferSize);
    if(result != TPM_RC_SUCCESS)
      return result + TPM_RC_S + g_rcIndex[sessionIndex];

    // Fourth parameter: authValue (PW or HMAC).
    result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
                  &sessionBuffer, &bufferSize);
    if(result != TPM_RC_SUCCESS)
      return result + TPM_RC_S + g_rcIndex[sessionIndex];

    if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
    {
      // A PWAP session needs additional processing.
      // Can’t have any attributes set other than continueSession bit
      if(  s_attributes[sessionIndex].encrypt
          || s_attributes[sessionIndex].decrypt
          || s_attributes[sessionIndex].audit
          || s_attributes[sessionIndex].auditExclusive
          || s_attributes[sessionIndex].auditReset
      )
        return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

      // The nonce size must be zero.
      if(s_nonceCaller[sessionIndex].t.size != 0)
        return TPM_RC_NONCE + TPM_RC_S + g_rcIndex[sessionIndex];

      continue;
    }

    // For not password sessions...

    // Find out if the session is loaded.
    if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
      return TPM_RC_REFERENCE_S0 + sessionIndex;

    sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
    session = SessionGet(s_sessionHandles[sessionIndex]);

    // Check if the session is an HMAC/policy session.
    if(  ( session->attributes.isPolicy == SET
          && sessionType == TPM_HT_HMAC_SESSION
      )
        || ( session->attributes.isPolicy == CLEAR
          && sessionType == TPM_HT_POLICY_SESSION
      )
    )
      continue;

}
return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];

// Check that this handle has not previously been used.
for(i = 0; i < sessionIndex; i++)
{
    if (s_sessionHandles[i] == s_sessionHandles[sessionIndex])
        return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
}

// If the session is used for parameter encryption or audit as well, set // the corresponding indices.

// First process decrypt.
if(s_attributes[sessionIndex].decrypt)
{
    // Check if the commandCode allows command parameter encryption.
    if(!CommandIsDecryptAllowed(commandCode))
        return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

    // Encrypt attribute can only appear in one session
    if(s_decryptSessionIndex != UNDEFINED_INDEX)
        return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

    // All checks passed, so set the index for the session used to decrypt // a command parameter.
    s_decryptSessionIndex = sessionIndex;
}

// Now process encrypt.
if(s_attributes[s_sessionNum].encrypt)
{
    // Check if the commandCode allows response parameter encryption.
    if(!CommandIsEncryptAllowed(commandCode))
        return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

    // Encrypt attribute can only appear in one session.
    if(s_encryptSessionIndex != UNDEFINED_INDEX)
        return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

    // All checks passed, so set the index for the session used to encrypt // a response parameter.
    s_encryptSessionIndex = sessionIndex;
}

// At last process audit.
if(s_attributes[sessionIndex].audit)
{
    // Audit attribute can only appear in one session.
    if(s_auditSessionIndex != UNDEFINED_INDEX)
        return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

    // An audit session can not be policy session.
    if( HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION)
        return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

    // If this is a reset of the audit session, or the first use // of the session as an audit session, it doesn't matter what // the exclusive state is. The session will become exclusive.
    if( s_attributes[sessionIndex].auditReset == CLEAR && session->attributes.isAudit == SET) // Not first use or reset. If auditExclusive is SET, then this // session must be the current exclusive session.
if( s_attributes[sessionIndex].auditExclusive == SET &&
   g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
  return TPM_RC_EXCLUSIVE;

s_auditSessionIndex = sessionIndex;

// Initialize associated handle as undefined. This will be changed when
// the handles are processed.
s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;

// At this point either all session data has been processed or sessions limit
// has been reached. In either case, the remaining size should be zero
if(bufferSize != 0)
  return TPM_RC_SIZE + TPM_RC_S + g_rcIndex[sessionIndex+1];

// Set the number of sessions found.
*sessionCount = sessionIndex;
return TPM_RC_SUCCESS;

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available at this time</td>
</tr>
<tr>
<td>TPM_RC_LOCKOUT</td>
<td>TPM is in lockout</td>
</tr>
</tbody>
</table>

static TPM_RC
CheckLockedOut(
  BOOL lockoutAuthCheck // IN: TRUE if checking is for lockoutAuth
)
{
  TPM_RC result;

  // If NV is unavailable, and current cycle state recorded in NV is not
  // SHUTDOWN_NONE, refuse to check any authorization because we would
  // not be able to handle a DA failure.
  result = NvIsAvailable();
  if(result != TPM_RC_SUCCESS && gp.orderlyState != SHUTDOWN_NONE)
    return result;

  // Check if DA info needs to be updated in NV.
  if(s_DAPendingOnNV)
    {
      // If NV is accessible, ...
      if(result == TPM_RC_SUCCESS)
        {
          // ... write the pending DA data and proceed.
          NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED,
                         &gp.lockOutAuthEnabled);
          NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
          g_updateNV = TRUE;
        }
    }
s_DAPendingOnNV = FALSE;
else
{  // Otherwise no authorization can be checked.
    return result;
}

// Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
// is disabled...
if(lockoutAuthCheck)
{
    if(gp.lockOutAuthEnabled == FALSE)
        return TPM_RC_LOCKOUT;
}  
else
{
    // ... or if the number of failed tries has been maxed out.
    if(gp.failedTries >= gp.maxTries)
        return TPM_RC_LOCKOUT;
}
return TPM_RC_SUCCESS;

6.4.4.8 CheckAuthSession()

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

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

static TPM_RC
CheckAuthSession(
TPM_CC           commandCode,   // IN: commandCode
UINT32           sessionIndex,  // IN: index of session to be processed
TPM2B_DIGEST    *cpHash,       // IN: cpHash
TPM2B_DIGEST    *nameHash     // IN: nameHash
)
{
    TPM_RC           result;
    SESSION         *session = NULL;
    TPM_HANDLE      sessionHandle = s_sessionHandles[sessionIndex];
    TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
    TPM_HT           sessionHandleType = HandleGetType(sessionHandle);
```c
pAssert(sessionHandle != TPM_RH_UNASSIGNED);

if(sessionHandle != TPM_RS_PW)
    session = SessionGet(sessionHandle);

// If the authorization session is not a policy session, or if the policy
// session requires authorization, then check lockout.
if(
    HandleGetType(sessionHandle) != TPM_HT_POLICY_SESSION
    || session->attributes.isAuthValueNeeded
    || session->attributes.isPasswordNeeded)
{
    // See if entity is subject to lockout.
    if(!IsDAExempted(associatedHandle))
    {
        // If NV is unavailable, and current cycle state recorded in NV is not
        // SHUTDOWN_NONE, refuse to check any authorization because we would
        // not be able to handle a DA failure.
        result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
}

if(associatedHandle == TPM_RH_PLATFORM)
{
    // If the physical presence is required for this command, check for PP
    // assertion. If it isn’t asserted, no point going any further.
    if(   PhysicalPresenceIsRequired(commandCode)
        && !_plat__PhysicalPresenceAsserted())
        return TPM_RC_PP;
}

// If a policy session is required, make sure that it is being used.
if(    IsPolicySessionRequired(commandCode, sessionIndex)
       && sessionHandleType != TPM_HT_POLICY_SESSION)
    return TPM_RC_AUTH_TYPE;

// If this is a PW authorization, check it and return.
if(sessionHandle == TPM_RS_PW)
{
    if(IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
        return CheckPWAuthSession(sessionIndex);
    else
        return TPM_RC_AUTH_UNAVAILABLE;
}

// If this is a policy session, ...
if(sessionHandleType == TPM_HT_POLICY_SESSION)
{
    // ... see if the entity has a policy, ...
    if(!IsAuthPolicyAvailable(associatedHandle, commandCode, sessionIndex))
        return TPM_RC_AUTH_UNAVAILABLE;
    // ... and check the policy session.
    result = CheckPolicyAuthSession(sessionIndex, commandCode,
        cpHash, nameHash);
    if (result != TPM_RC_SUCCESS)
        return result;
}
else
{
    // For non policy, the entity being accessed must allow authorization
    // with an auth value. This is required even if the auth value is not
    // going to be used in an HMAC because it is bound.
    if(!IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
        return TPM_RC_AUTH_UNAVAILABLE;
}

// At this point, the session must be either a policy or an HMAC session.
```
session = SessionGet(s_sessionHandles[sessionIndex]);

if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
    && session->attributes.isPasswordNeeded == SET)
{
    // For policy session that requires a password, check it as PWAP session.
    return CheckPWAuthSession(sessionIndex);
}
else
{
    // For other policy or HMAC sessions, have its HMAC checked.
    return CheckSessionHMAC(sessionIndex, cpHash);
}
#endif

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available for write</td>
</tr>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting</td>
</tr>
</tbody>
</table>

static TPM_RC CheckCommandAudit(
    TPM_CC commandCode, // IN: Command code
    UINT32 handleNum,   // IN: number of element in handle array
    TPM_HANDLE handles[], // IN: array of handles
    BYTE *parmBufferStart, // IN: start of parameter buffer
    UINT32 parmBufferSize // IN: size of parameter buffer
)
{
    TPM_RC result = TPM_RC_SUCCESS;

    // If audit is implemented, need to check to see if auditing is being done
    // for this command.
    if(CommandAuditIsRequired(commandCode))
    {
        // If the audit digest is clear and command audit is required, NV must be
        // available so that TPM2_GetCommandAuditDigest() is able to increment
        // audit counter. If NV is not available, the function bails out to prevent
        // the TPM from attempting an operation that would fail anyway.
        if(gr.commandAuditDigest.t.size == 0
            || commandCode == TPM_CC_GetCommandAuditDigest)
        {
            result = NvIsAvailable();
            if(result != TPM_RC_SUCCESS)
                return result;
        }
        ComputeCpHash(gp.auditHashAlg, commandCode, handleNum,
            handles, parmBufferStart, parmBufferSize, &s_cpHashForCommandAudit, NULL);
    }
    return TPM_RC_SUCCESS;
}
#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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing Error</td>
<td>failure</td>
</tr>
</tbody>
</table>

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

// Consistency checks are done first to avoid auth failure when the command
// will not be executed anyway.
for(sessionIndex = 0; sessionIndex < s_sessionNum; sessionIndex++)
{
    // FW session must be an authorization session
    if(s_sessionHandles[sessionIndex] == TPM_RS_PW )
    {
        if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
            return TPM_RC_HANDLE + g_rcIndex[sessionIndex];
    }
    else
    {
        session = SessionGet(s_sessionHandles[sessionIndex]);
        // A trial session can not appear in session area, because it cannot
        // be used for authorization, audit or encrypt/decrypt.
        if(s_attributes[sessionIndex].isTrialPolicy == SET)
            return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

        // See if the session is bound to a DA protected entity
        if(s_attributes[sessionIndex].isDaBound == SET)
            { result = CheckLockedOut(s_attributes[sessionIndex].isLockoutBound == SET);
                if(result != TPM_RC_SUCCESS)
                    return result;
            }
        // If the current cpHash is the right one, don't re-compute.
        if(cpHashAlg != session->authHashAlg) // different so compute
            { cpHashAlg = session->authHashAlg; // save this new algID
                ComputeCpHash(session->authHashAlg, commandCode, handleNum,
                               handles, parmBufferSize, parmBufferStart,
                               &cpHash, &nameHash);
            }
        // If this session is for auditing, save the cpHash.
        if(s_attributes[sessionIndex].audit)
            s_cpHashForAudit = cpHash;
    }

    // if the session has an associated handle, check the auth
    if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
    { result = CheckAuthSession(commandCode, sessionIndex,
                                &cpHash, &nameHash);
        if(result != TPM_RC_SUCCESS)
            return RcSafeAddToResult(result,
                                      TPM_RC_S + g_rcIndex[sessionIndex]);
    }
    else
    {
        // a session that is not for authorization must either be encrypt,
        // decrypt, or audit
        if( (s_attributes[sessionIndex].audit == CLEAR &&
             s_attributes[sessionIndex].encrypt == CLEAR &&
             s_attributes[sessionIndex].decrypt == CLEAR)
            return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

        // check HMAC for encrypt/decrypt/audit only sessions
        result = CheckSessionHMAC(sessionIndex, &cpHash);
        if(result != TPM_RC_SUCCESS)
            return RcSafeAddToResult(result,
                                      TPM_RC_S + g_rcIndex[sessionIndex]);
    }
#ifdef TPM_CC_GetCommandAuditDigest
    // Check if the command should be audited.
    result = CheckCommandAudit(commandCode, handleNum, handles, parmBufferStart, parmBufferSize);
    if(result != TPM_RC_SUCCESS)
        return result;                        // No session number to reference
#endif

    // Decrypt the first parameter if applicable. This should be the last operation 
    // in session processing.
    // If the encrypt session is associated with a handle and the handle's 
    // authValue is available, then authValue is concatenated with sessionAuth to 
    // generate encryption key, no matter if the handle is the session bound entity 
    // or not.
    if(s_decryptSessionIndex != UNDEFINED_INDEX)
    {
        // Get size of the leading size field in decrypt parameter
        if(  s_associatedHandles[s_decryptSessionIndex]  != TPM_RH_UNASSIGNED
            && IsAuthValueAvailable(s_associatedHandles[s_decryptSessionIndex], commandCode, s_decryptSessionIndex)
        
        )
        
        {
            extraKey.b.size = EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex], extraKey.b.buffer);
        }
        
        else
        {
            extraKey.b.size = 0;
        }

        size = EncryptDecryptSize(commandCode);
        pAssert(size < INT16_MAX);
        result = CryptParameterDecryption( 
            s_sessionHandles[s_decryptSessionIndex], 
            &s_nonceCaller[s_decryptSessionIndex].b, 
            parmBufferSize, (UINT16)size, 
            extraKey, 
            parmBufferStart);
        if(result != TPM_RC_SUCCESS)
            return RcSafeAddToResult(result, 
                TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);

    }

    return TPM_RC_SUCCESS;
#endif

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_AUTH_MISSING</td>
<td>failure - one or more handles require auth</td>
</tr>
</tbody>
</table>

TPM_RC
CheckAuthNoSession( 
    TPM_CC           commandCode,  // IN: Command Code
    UINT32           handleNum,    // IN: number of handles in command
    TPM_HANDLE       handles[],    // IN: array of handles
    BYTE             *parmBufferStart,  // IN: start of parameter buffer
    UINT32           parmBufferSize // IN: size of parameter buffer
)
{  
  UINT32 i;
  TPM_RC result = TPM_RC_SUCCESS;
  
  // Check if the commandCode requires authorization
  for (i = 0; i < handleNum; i++)
  {
    if (CommandAuthRole(commandCode, i) != AUTH_NONE)
      return TPM_RC_AUTH_MISSING;
  }

  #ifdef TPM_CC_GetCommandAuditDigest
  // Check if the command should be audited.
  result = CheckCommandAudit(commandCode, handleNum, handles,
                              parmBufferStart, parmBufferSize);
  if (result != TPM_RC_SUCCESS) return result;
  #endif

  // Initialize number of sessions to be 0
  s_sessionNum = 0;

  return TPM_RC_SUCCESS;
}

6.4.5  Response Session Processing

6.4.5.1  Introduction

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

6.4.5.2  ComputeRpHash()

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

static void ComputeRpHash(
  TPM_ALG_ID hashAlg,  // IN: hash algorithm to compute rpHash
  TPM_CC commandCode,  // IN: commandCode
  UINT32 resParmBufferSize, // IN: size of response parameter buffer
  BYTE *resParmBuffer, // IN: response parameter buffer
  TPM2B_DIGEST *rpHash // OUT: rpHash
)
{
  TPM_RC responseCode = TPM_RC_SUCCESS;
  HASH_STATE hashState;

  // The command result in rpHash is always TPM_RC_SUCCESS.
  TPM_RC result = TPM_RC_SUCCESS;

  // Initiate hash creation.
  rpHash->t.size = CryptStartHash(hashAlg, &hashState);

  // Add hash constituents.
  CryptUpdateDigestInt(&hashState, sizeof(TPM_RC), &responseCode);
  CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
  CryptUpdateDigest(&hashState, resParmBufferSize, resParmBuffer);

  // Complete hash computation.
  CryptCompleteHash2B(&hashState, &rpHash->b);

  return;
}
6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```c
static void InitAuditSession(
    SESSION *session // session to be initialized
)
{
    // Mark session as an audit session.
    session->attributes.isAudit = SET;

    // Audit session can not be bound.
    session->attributes.isBound = CLEAR;

    // Size of the audit log is the size of session hash algorithm digest.
    session->u2.auditDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);

    // Set the original digest value to be 0.
    MemorySet(&session->u2.auditDigest.t.buffer,
        0,
        session->u2.auditDigest.t.size);

    return;
}
```

6.4.5.4 Audit()

This function updates the audit digest in an audit session.

```c
static void Audit(
    SESSION *auditSession, // IN: loaded audit session
    TPM_CC commandCode, // IN: commandCode
    UINT32 resParmBufferSize, // IN: size of response parameter buffer
    BYTE *resParmBuffer // IN: response parameter buffer
)
{
    TPM2B_DIGEST rpHash; // rpHash for response
    HASH_STATE hashState;

    // Compute rpHash
    ComputeRpHash(auditSession->authHashAlg, commandCode, resParmBufferSize, resParmBuffer, &rpHash);

    // auditDigestnew := hash (auditDigestold || cpHash || rpHash)
    CryptStartHash(auditSession->authHashAlg, &hashState);

    // Add old digest.
    CryptUpdateDigest2B(&hashState, &auditSession->u2.auditDigest.b);

    // Add cpHash and rpHash.
    CryptUpdateDigest2B(&hashState, &s_cpHashForAudit.b);
    CryptUpdateDigest2B(&hashState, &rpHash.b);

    // Finalize the hash.
```
CryptCompleteHash2B(&hashState, &auditSession->u2.auditDigest.b);
return;
#endif  TPM_CC_GetCommandAuditDigest

6.4.5.5 CommandAudit()

This function updates the command audit digest.

static void CommandAudit(
    TPM_CC commandCode,  // IN: commandCode
    UINT32 resParmBufferSize,  // IN: size of response parameter buffer
    BYTE *resParmBuffer  // IN: response parameter buffer
) {
    static void
    if(CommandAuditIsRequired(commandCode)) {
        TPM2B_DIGEST rpHash;  // rpHash for response
        HASH_STATE hashState;

        // Compute rpHash.
        ComputeRpHash(gp.auditHashAlg, commandCode, resParmBufferSize,
                       resParmBuffer, &rpHash);

        // If the digest.size is one, it indicates the special case of changing
        // the audit hash algorithm. For this case, no audit is done on exit.
        // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
        // force an update to the NV on exit so that the change in digest will
        // be recorded. So, it is safe to exit here without setting any flags
        // because the digest change will be written to NV when this code exits.
        if(gr.commandAuditDigest.t.size == 1) {
            gr.commandAuditDigest.t.size = 0;
            return;
        }

        // If the digest size is zero, need to start a new digest and increment
        // the audit counter.
        if(gr.commandAuditDigest.t.size == 0) {
            gr.commandAuditDigest.t.size = CryptGetHashDigestSize(gp.auditHashAlg);
            MemorySet(gr.commandAuditDigest.t.buffer,
                       0,
                       gr.commandAuditDigest.t.size);

            // Bump the counter and save its value to NV.
            gp.auditCounter++;
            NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
            g_updateNV = TRUE;
        }

        // auditDigestnew := hash (auditDigestold || cpHash || rpHash)

        // Start hash computation.
        CryptStartHash(gp.auditHashAlg, &hashState);

        // Add old digest.
        CryptUpdateDigest2B(&hashState, &gr.commandAuditDigest.b);

        // Add cpHash
        CryptUpdateDigest2B(&hashState, &s_cpHashForCommandAudit.b);

        return;
    }
}
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             } else
1466             {
1467                 // Check if the audit session is the current exclusive audit
1468                 // session and, if not, clear previous exclusive audit session.
1469                 if(g_exclusiveAuditSession != auditSession)
1471  g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1472 }
1473 // Report audit session exclusivity.
1474 if(g_exclusiveAuditSession == auditSession)
1475 {
1476   s_attributes[i].auditExclusive = SET;
1477 }
1478 else
1479 {
1480   s_attributes[i].auditExclusive = CLEAR;
1481 }
1482 // Extend audit log.
1483 Audit(session, commandCode, resParmBufferSize, resParmBuffer);
1484 }
1485 }
1486 // If no audit session is found in the command, and the command allows
1487 // a session then, clear the current exclusive
1488 // audit session.
1489 if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(commandCode))
1490 {
1491   g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1492 }
1493 return;
1494 }
1495
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   *resParmBuffer,  // IN: response parameter buffer
1507   TPM2B_DIGEST *hmac        // OUT: authHMAC
1508 )
1509 {
1510   TPM2B_TYPE(KEY, (sizeof(TPMT_HA) * 2));
1511   TPM2B_KEY   key;     // HMAC key
1512   BYTE       marshalBuffer[sizeof(TPMA_SESSION)];
1513   BYTE       *buffer;
1514   UINT32     marshalSize;
1515   HMAC_STATE hmacState;
1516   TPM2B_DIGEST rp_hash;
1517   // Compute rpHash.
1518   ComputeRpHash(session->authHashAlg, commandCode, resParmBufferSize,
1519                   resParmBuffer, &rp_hash);
1520   // Generate HMAC key
1521   MemoryCopy2B(&key.b, &session->sessionKey.b);
1522   // Check if the session has an associated handle and the associated entity is
1523   // the one that the session is started with.
1524   // If so, add the authValue of this entity to the HMAC key.
1525   if(   s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED
1526     // Compute rpHash.
1527     ComputeRpHash(session->authHashAlg, commandCode, resParmBufferSize,
1528                    resParmBuffer, &rp_hash);
1529     // Generate HMAC key
1530     MemoryCopy2B(&key.b, &session->sessionKey.b);
1531     // Check if the session has an associated handle and the associated entity is
1532     // the one that the session is started with.
1533     // If so, add the authValue of this entity to the HMAC key.
1534     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  // if the HMAC key size for a policy session is 0, the response HMAC is
1540  // computed according to the input HMAC
1541  if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1542      && key.t.size == 0
1543      && s_inputAuthValues[sessionIndex].t.size == 0)
1544  {
1545      hmac->t.size = 0;
1546      return;
1547  }
1548  // Start HMAC computation.
1549  hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
1550  // Add hash components.
1551  CryptUpdateDigest2B(&hmacState, &rp_hash.b);
1552  CryptUpdateDigest2B(&hmacState, &nonceTPM->b);
1553  CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
1554  // Add session attributes.
1555  buffer = marshalBuffer;
1556  marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1557  CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
1558  // Finalize HMAC.
1559  CryptCompleteHMAC2B(&hmacState, &hmac->b);
1560  return;
1561 }

6.4.5.8 BuildSingleResponseAuth()

Function to compute response for an authorization session.

static void BuildSingleResponseAuth(
1567  UINT32 sessionIndex,       // IN: session index to be processed
1568  TPM_CC commandCode,       // IN: commandCode
1569  UINT32 resParmBufferSize,  // IN: size of response parameter buffer
1570  BYTE *resParmBuffer,      // IN: response parameter buffer
1571  TPM2B_AUTH *auth)         // OUT: authHMAC
1572 }
1573 // For password authorization, field is empty.
1574 if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1575 {
1576    auth->t.size = 0;
1577  }
1578 else
1579  {
1580    // Fill in policy/HMAC based session response.
1581    SESSION *session = SessionGet(s_sessionHandles[sessionIndex]);
1582    // If the session is a policy session with isPasswordNeeded SET, the auth
1583    // field is empty.
```
if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION && session->attributes.isPasswordNeeded == SET) {
    auth->t.size = 0;
} else {
    // Compute response HMAC.
    ComputeResponseHMAC(sessionIndex, session, commandCode, &session->nonceTPM, resParmBufferSize, resParmBuffer, auth);
}

return;

6.4.5.9 UpdateTPMNonce()

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

static void UpdateTPMNonce()
{
    TPM2B_NONCE nonces[] // OUT: nonceTPM

    UINT32 i;
    for(i = 0; i < s_sessionNum; i++) {
        SESSION *session;
        // For PW session, nonce is 0.
        if(s_sessionHandles[i] == TPM_RS_PW) {
            nonces[i].t.size = 0;
            continue;
        }
        session = SessionGet(s_sessionHandles[i]);
        // Update nonceTPM in both internal session and response.
        CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
        nonces[i] = session->nonceTPM;
    }
    return;

6.4.5.10 UpdateInternalSession()

Updates internal sessions:

r) Restarts session time.

s) Clears a policy session since nonce is rolling.

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

Family "02"  Published  Page 65
Level 00 Revision 00.96 Copyright © TCG 2006-2013 March 15, 2013
// Close internal session.
SessionFlush(s_sessionHandles[i]);
}
else
{
    // If nonce is rolling in a policy session, the policy related data
    // will be re-initialized.
    if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION)
    {
        SESSION *session = SessionGet(s_sessionHandles[i]);
        // When the nonce rolls it starts a new timing interval for the
        // policy session.
        SessionResetPolicyData(session);
        session->startTime = go.clock;
    }
}
return;
}

6.4.5.11 BuildResponseSession()

Function to build Session buffer in a response.

void BuildResponseSession(
    TPM_ST tag,          // IN: tag
    TPM_CC commandCode,  // IN: commandCode
    UINT32 resHandleSize, // IN: size of response handle buffer
    UINT32 resParmSize,   // IN: size of response parameter buffer
    UINT32 *resSessionSize // OUT: response session area
)
{
    BYTE *resParmBuffer;
    TPM2B_NONCE responseNonces[MAX_SESSION_NUM];
    // Compute response parameter buffer start.
    resParmBuffer = MemoryGetResponseBuffer(commandCode) +
                    sizeof(TPM_ST) +
                    sizeof(UINT32) +
                    sizeof(TPM_RC) +
                    resHandleSize;
    // For TPM_ST_SESSIONS, there is parameterSize field.
    if(tag == TPM_ST_SESSIONS)
        resParmBuffer += sizeof(UINT32);
    // Session nonce should be updated before parameter encryption
    if(tag == TPM_ST_SESSIONS)
    {
        UpdateTPMNonce(responseNonces);
        // Encrypt first parameter if applicable. Parameter encryption should
        // happen after nonce update and before any rpHash is computed.
        // If the encrypt session is associated with a handle, the authValue of
        // this handle will be concatenated with sessionAuth to generate
        // encryption key, no matter if the handle is the session bound entity
        // or not. The authValue is added to sessionAuth only when the authValue
        // is available.
        if(s_encryptSessionIndex != UNDEFINED_INDEX)
        {
            UINT32 size;
            TPM2B_AUTH extraKey;
            // Get size of the leading size field
            if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED

\begin{verbatim}
1697    /// IsAuthValueAvailable(s_associatedHandles[s_encryptSessionIndex],
1698    /// commandCode, s_encryptSessionIndex)
1699    }
1700    }
1701    }
1702    }
1703    }
1704    }
1705    }
1706    }
1707    }
1708    }
1709    }
1710    }
1711    }
1712    }
1713    }
1714    }
1715    }
1716    }
1717    }
1718    }
1719    }
1720    }
1721    }
1722    }
1723    }
1724    }
1725    }
1726    }
1727    }
1728    }
1729    }
1730    }
1731    }
1732    }
1733    }
1734    }
1735    }
1736    }
1737    }
1738    }
1739    }
1740    }
1741    }
1742    }
1743    }
1744    }
1745    }
1746    }
1747    }
1748    }
1749    }
1750    }
1751    }
1752    }
1753    }
1754    }
1755    }
1756    }
1757    }
1758    }
1759    }
1760    }
1761    }
1762    
\end{verbatim}
    *resSessionSize += TPMA_SESSION.Marshal(&s_attributes[i],
    &buffer, NULL);
    *resSessionSize += TPM2B_DIGEST.Marshal(&responseAuths[i],
    &buffer, NULL);
}

// Update internal sessions after completing response buffer computation.
    UpdateInternalSession();

else

    // Process command with no session.
    *resSessionSize = 0;
}

return;}
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)

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

7.2.1.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

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

```c
TPM_RC

FillInAttestInfo()

TPMI_DH_OBJECT signHandle, // IN: handle of signing object
TPM_SIG_SCHEME *scheme, // IN/OUT: scheme to be used for signing
TPM2B_DATA *data, // IN: qualifying data
TPMS_ATTEST *attest // OUT: attest structure

{  
TPM_RC result;
TPMI_RH_HIERARCHY signHierarchy;
result = CryptSelectSignScheme(signHandle, scheme);
if(result != TPM_RC_SUCCESS)
  return result;

// Magic number
attest->magic = TPM_GENERATED_VALUE;

if(signHandle == TPM_RH_NULL)
  {
    BYTE *buffer;
    // For null sign handle, the QN is TPM_RH_NULL
    buffer = attest->qualifiedSigner.t.name;
    attest->qualifiedSigner.t.size = TPM_HANDLE_Marshal(&signHandle, &buffer, NULL);
  }
else
  {
    // Certifying object qualified name
    // if the scheme is anonymous, this is an empty buffer
    if(CryptIsSchemeAnonymous(scheme->scheme))
      attest->qualifiedSigner.t.size = 0;
    else
      ObjectGetQualifiedName(signHandle, &attest->qualifiedSigner);
  }
// current clock in plain text
```
TimeFillInfo(&attest->clockInfo);

// Firmware version in plain text
attest->firmwareVersion = ((UINT64) gp.firmwareV1 << (K*sizeof(UINT32) * 8));
attest->firmwareVersion += gp.firmwareV2;

// Get the hierarchy of sign object. For NULL sign handle, the hierarchy will be TPM RH NULL
signHierarhcy = EntityGetHierarchy(signHandle);
if(signHierarhcy != TPM_RH_PLATFORM && signHierarhcy != TPM_RH_ENDORSEMENT)
{
    // For sign object is not in platform or endorsement hierarchy, obfuscate the clock and firmwareVersion information
    UINT64          obfuscation[2];
    TPMI_ALG_HASH   hashAlg;

    // Get hash algorithm
    if(signHandle == TPM_RH_NULL || signHandle == TPM_RH_OWNER)
    {
        hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
    }
    else
    {
        OBJECT          *signObject = NULL;
        signObject = ObjectGet(signHandle);
        hashAlg = signObject->publicArea.nameAlg;
    }
    KDFa(hashAlg, &gp.shProof.b, "OBFUSCATE",
         &attest->qualifiedSigner.b, NULL, 128, (BYTE *)&obfuscation[0], NULL);

    // Obfuscate data
    attest->firmwareVersion += obfuscation[0];
    attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
    attest->clockInfo.restartCount += (UINT32)obfuscation[1];
}

// External data
if(CryptIsSchemeAnonymous(scheme->scheme))
    attest->extraData.t.size = 0;
else
{
    // If we move the data to the attestation structure, then we will not use it in the signing operation except as part of the signed data
    attest->extraData = *data;
    data->t.size = 0;
}
return TPM_RC_SUCCESS;

7.2.1.2 SignAttestInfo()

Sign a TPMS_ATTEST structure. If signHandle is TPM RH NULL, a null signature is returned.

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

TPM_RC
90  \textbf{SignAttestInfo}(
91  \texttt{TPMI\_DH\_OBJECT} signHandle, \hspace{1em} // IN: handle of sign object
92  \texttt{TPMT\_SIG\_SCHEME} \*scheme, \hspace{1em} // IN: sign scheme
93  \texttt{TPMS\_ATTEST} \*certifyInfo, \hspace{1em} // IN: the data to be signed
94  \texttt{TPM2B\_DATA} \*qualifyingData, \hspace{1em} // IN: extra data for the signing
95  // process
96  \texttt{TPM2B\_ATTEST} \*attest, \hspace{1em} // OUT: marshaled attest blob to
97  // be signed
98  \texttt{TPMT\_SIGNATURE} \*signature \hspace{1em} // OUT: signature
99  )
100 
101  TPM\_RC result;
102  TPM\_ALG\_HASH hashAlg;
103  BYTE \*buffer;
104  HASH\_STATE hashState;
105  TPM2B\_DIGEST digest;
106
107  // Marshal TPMS\_ATTEST structure for hash
108  buffer = attest->t.attestationData;
109  attest->t.size = TPMS\_ATTEST\_Marshal(certifyInfo, \&buffer, NULL);
110
111  if(signHandle == TPM\_RH\_NULL)
112  {
113    signature->sigAlg = TPM\_ALG\_NULL;
114  }
115  else
116  {
117    // Attestation command may cause the orderlyState to be cleared due to
118    // the reporting of clock info. If this is the case, check if NV is
119    // available first
120    if(gp.orderlyState != SHUTDOWN\_NONE)
121    {
122      // The command needs NV update. Check if NV is available.
123      // A TPM\_RC\_NV\_UNAVAILABLE or TPM\_RC\_NV\_RATE error may be returned at
124      // this point
125      result = NvIsAvailable();
126      if(result != TPM\_RC\_SUCCESS)
127        return result;
128    }
129
130    // Compute hash
131    hashAlg = scheme->details.any.hashAlg;
132    digest.t.size = CryptStartHash(hashAlg, \&hashState);
133    CryptUpdateDigest(\&hashState, attest->t.size, attest->t.attestationData);
134    CryptCompleteHash2B(\&hashState, \&digest.b);
135
136    // If there is qualifying data, need to rehash the the data
137    // hash(qualifyingData || hash(attestationData))
138    if(qualifyingData->t.size != 0)
139    {
140      CryptStartHash(hashAlg, \&hashState);
141      CryptUpdateDigest(\&hashState, qualifyingData->t.size, qualifyingData->t.buffer);
142      CryptUpdateDigest(\&hashState, digest.t.size, digest.t.buffer);
143      CryptCompleteHash2B(\&hashState, \&digest.b);
144    }
145
146    // Sign the hash. A TPM\_RC\_VALUE, TPM\_RC\_SCHEME, or
147    // TPM\_RC\_ATTRIBUTES error may be returned at this point
148    return CryptSign(signHandle,
149      scheme, 
150      \&digest,
151      \&signature);
7.3 Context Management Command Support (Context_spt.c)

This function retrieves the symmetric protection key for context encryption. It is used by TPM2_ContextSave() and TPM2_ContextLoad() to create the symmetric encryption key and IV.

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

7.3.1.1 ComputeContextProtectionKey()

void ComputeContextProtectionKey(
    TPM2B_AUTH *proof = NULL, // the proof value to use. Is null for everything but a primary object in the Endorsement Hierarchy
    BYTE kdfResult[sizeof(TPMU_HA) * 2]; // Value produced by the KDF
) {
    UINT16 symKeyBits; // number of bits in the parent's symmetric key
    TPM2B_DATA sequence2B, handle2B;
    // Get proof value
    proof = HierarchyGetProof(contextBlob->hierarchy);
    // Get sequence value in 2B format
    sequence2B.t.size = sizeof(contextBlob->sequence);
    MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence, sizeof(contextBlob->sequence));
    // Get handle value in 2B format
    handle2B.t.size = sizeof(contextBlob->savedHandle);
    MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle, sizeof(contextBlob->savedHandle));
    // Get the symmetric encryption key size
    symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
    symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
    // Get the size of the IV for the algorithm
    iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);
    // KDFa to generate symmetric key and IV value
    KDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, "CONTEXT", &sequence2B.b, &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL);
    // Copy part of the returned value as the key
    MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size);
    // Copy the rest as the IV
    MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
    return;
}
```
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.

```c
void ComputeContextIntegrity(
    TPMS_CONTEXT *contextBlob,  // IN: context blob
    TPM2B_DIGEST *integrity   // OUT: integrity
)
{
    HMAC_STATE hmacState;
    TPM2B_AUTH *proof;
    UINT16 integritySize;

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

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

    // Adding total reset counter
    CryptUpdateDigestInt(&hmacState, sizeof(gp.totalResetCount),
        &gp.totalResetCount);

    // Adding clearCount
    if (contextBlob->savedHandle == 0x80000002)
        CryptUpdateDigestInt(&hmacState, sizeof(gr.clearCount), &gr.clearCount);

    // Adding sequence
    CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->sequence),
        &contextBlob->sequence);

    // Adding handle
    CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->savedHandle),
        &contextBlob->savedHandle);

    // Compute integrity size at the beginning of context blob
    integritySize = sizeof(integrity->t.size)
        + CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);

    // Adding sensitive contextData, skip the leading integrity area
    CryptUpdateDigest(&hmacState, contextBlob->contextBlob->contextBlob.t.size - integritySize,
        contextBlob->contextBlob.t.buffer + integritySize);

    // Complete HMAC
    CryptCompleteHMAC2B(&hmacState, &integrity->b);
    return;
}
```

7.4 Policy Command Support (Policy_spt.c)

7.4.1.1 ValidatePolicyID()

Validate `nonceTPM` parameter for TPM2_PolicySigned(), and TPM2_PolicySecret().
Error Returns | Meaning
--- | ---
TPM_RC_VALUE | if fails

```c
TPM_RC
ValidatePolicyID(
    TPM2B_NONCE   *nonceTPM,  // IN: nonceTPM
    SESSION       *session)   // IN: policy session
{
    if(nonceTPM->t.size != 0)
    {
        if(!Memory2BEqual(nonceTPM->b, &session->nonceTPM.b))
            return TPM_RC_VALUE;
    }
    return TPM_RC_SUCCESS;
}
```

7.4.1.2 ValidateExpiration()

Validate expiration parameter for TPM2_PolicySigned() and TPM2_PolicySecret()

Error Returns | Meaning
--- | ---
TPM_RC_VALUE | if fails

```c
TPM_RC
ValidateExpiration(
    UINT32               expiration, // IN: expiration in millisecond
    SESSION             *session) // IN: policy session
){
    TPM_RC          result = TPM_RC_SUCCESS;
    if(expiration != 0)
    {
        // Cannot compare time if clock stop advancing. A TPM_RC_NV_UNAVAILABLE
        // or TPM_RC_NV_RATE error may be returned here.
        result = NvIsAvailable();
        if(result != TPM_RC_SUCCESS)
            return result;
        if((UINT64) expiration * 1000 < go.clock - session->startTime)
            return TPM_RC_EXPIRED;
    }
    return TPM_RC_SUCCESS;
}
```

7.4.1.3 UpdateTimeout()

Update timeout in a policy session

```c
void
UpdateTimeout(
    UINT64               timeout,   // IN: the new timeout value
    SESSION             *session)  // IN: the session
){
    // If the timeout has not been set, then set it to the new value
    if(session->timeOut == 0)
        session->timeOut = timeout;
```
7.4.1.4 PolicyUpdate()

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

```c
void PolicyUpdate(
    TPM_CC commandCode, // IN: command code
    TPM2B_NAME *name,    // IN: name of entity
    TPM2B_NONCE *ref,    // IN: the reference data
    SESSION *session     // IN/OUT: policy session to be updated
) {
    HASH_STATE hashState;
    UINT16 policyDigestSize;
    // Start hash
    policyDigestSize = CryptStartHash(session->authHashAlg, &hashState);
    // policyDigest size should always be the digest size of session hash alg.
    pAssert(session->u2.policyDigest.t.size == policyDigestSize);
    // add old digest
    CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);
    // add commandCode
    CryptUpdateDigestInt(&hashState, sizeof(commandCode), &commandCode);
    // add name if applicable
    if (name != NULL)
        CryptUpdateDigest2B(&hashState, &name->b);
    // Complete the digest and get the results
    CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);
    // Start second hash computation
    CryptStartHash(session->authHashAlg, &hashState);
    // add policyDigest
    CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);
    // add policyRef
    if (ref != NULL)
        CryptUpdateDigest2B(&hashState, &ref->b);
    // Complete second digest
    CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);
    return;
}
```

7.5 NV Command Support (NV_spt.c)

```c
#include "InternalRoutines.h"
#include "NV_spt_fp.h"
```
Common routine for validating a read Used by TPM2_NV_Read(), TPM2_NV_ReadLock() and TPM2_PolicyNV()

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_AUTHORIZATION</td>
<td>'authHandle' is not allowed to authorize read of the index</td>
</tr>
<tr>
<td>TPM_RC_NV_LOCKED</td>
<td>Read locked</td>
</tr>
<tr>
<td>TPM_RC_NV_UNINITIALIZED</td>
<td>Try to read an uninitialized index</td>
</tr>
</tbody>
</table>

```
TPM_RC

NvReadAccessChecks(
    TPM_HANDLE authHandle, // IN: the handle that provided the
    // authorization
    TPM_HANDLE nvHandle  // IN: the handle of the NV index to be
    // written
)

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

// If data is read locked, returns an error
if(nvIndex.publicArea.attributes.TPMA_NV_READLOCKED == SET)
    return TPM_RC_NV_LOCKED;

// If the index has not been written, then the value cannot be read
if(nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
    return TPM_RC_NV_UNINITIALIZED;

// If the authorization was provided by the owner or platform, then check
// that the attributes allow the read. If the authorization handle
// is the same as the index, then the checks were made when the authorization
// was checked..
if(authHandle == TPM_RH_OWNER)
{
    // If Owner provided auth then ONWERWRITE must be SET
    if(!nvIndex.publicArea.attributes.TPMA_NV_OWNERREAD)
        return TPM_RC_NV_AUTHORIZATION;
}
else if(authHandle == TPM_RH_PLATFORM)
{
    // If Platform provided auth then PPWRITE must be SET
    if(!nvIndex.publicArea.attributes.TPMA_NV_PPREAD)
        return TPM_RC_NV_AUTHORIZATION;

    // If neither Owner nor Platform provided auth, make sure that it was
    // provided by this index.
}
else
{
    // make sure that the handles match
    if(authHandle != nvHandle)
        return TPM_RC_NV_AUTHORIZATION;

    // If the hierarchy that the object was created is disabled, only
    // another hierarchy handle can be used to authorize the access.
    if(  ( nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET
            && g_phEnable == CLEAR)
        || ( nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR
            && gc.shEnable == CLEAR)
    )
        return TPM_RC_NV_AUTHORIZATION;
}
```
Common routine for validating a write. Used by TPM2_NV_Write(), TPM2_NV_Increment(), TPM2_SetBits(), and TPM2_NV_WriteLock().

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_AUTHORIZATION</td>
<td>Authorization fails</td>
</tr>
<tr>
<td>TPM_RC_NV_LOCKED</td>
<td>Write locked</td>
</tr>
</tbody>
</table>

```c
return TPM_RC_SUCCESS;
```

```c
TPM_RC
NvWriteAccessChecks(
    TPM_HANDLE authHandle, // IN: the handle that provided the
    // authorization
    TPM_HANDLE nvHandle // IN: the handle of the NV index to be
    // written
)
{
    NV_INDEX nvIndex;

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

    // If data is write locked, returns an error
    if (nvIndex.publicArea.attributes.TPMA_NWRITELOCKED == SET)
        return TPM_RC_NV_LOCKED;

    // If the authorization was provided by the owner or platform, then check
    // that the attributes allow the write. If the authorization handle
    // is the same as the index, then the checks were made when the authorization
    // was checked...
    if (authHandle == TPM_RH_OWNER)
    {
        // If Owner provided auth then ONWERWRITE must be SET
        if (!nvIndex.publicArea.attributes.TPMA_NV_OWNERWRITE)
            return TPM_RC_NV_AUTHORIZATION;
    }
    else if (authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided auth then PPWRITE must be SET
        if (!nvIndex.publicArea.attributes.TPMA_NV_PPWRITE)
            return TPM_RC_NV_AUTHORIZATION;
    }
    else
    {
        // make sure that the handles match
        if (authHandle != nvHandle)
            return TPM_RC_NV_AUTHORIZATION;

        // If the hierarchy that the object was created is disabled, only
        // another hierarchy handle can be used to authorize the access.
        if (   (   nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET
            && g_phEnable == CLEAR)
            || (   nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR
            && gc.shEnable == CLEAR)
        )
            return TPM_RC_NV_AUTHORIZATION;
    }

    return TPM_RC_SUCCESS;
```
Object Command Support (Object_spt.c)

Check if the crypto sets in two public areas are equal

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ASYMMETRIC</td>
<td>mismatched parameters</td>
</tr>
<tr>
<td>TPM_RC_HASH</td>
<td>mismatched name algorithm</td>
</tr>
<tr>
<td>TPM_RC_TYPE</td>
<td>mismatched type</td>
</tr>
</tbody>
</table>

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

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

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

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

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

    return TPM_RC_SUCCESS;
}
```

7.6.1.1 AreAttributesForParent()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>properties are those of a parent</td>
</tr>
<tr>
<td>FALSE</td>
<td>properties are not those of a parent</td>
</tr>
</tbody>
</table>

```c
BOOL AreAttributesForParent(
    OBJECT      *parentObject // IN: parent handle
) 
```
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().

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

TPM_RC

SchemeChecks(

54 TPMI_DH_OBJECT parentHandle,  // IN: input parent handle  
55 TPMT_PUBLIC *publicArea       // IN: public area of the object  
56 )

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     else if(publicArea->objectAttributes.encrypt == SET  
82             && keyScheme->scheme != TPM_ALG_NULL)  
83         return TPM_RC_SCHEME;  
84     else if(CryptAreKeySizesConsistent(publicArea))  
85         return TPM_RC_SCHEME;  
86     else  
87         return TPM_RC_SCHEME;  
88 }
return TPM_RC_SCHEME;

// A restrict sign key must have a non-NULL scheme
if (publicArea->objectAttributes.restricted == SET
    && publicArea->objectAttributes.sign == SET
    && keyScheme->scheme == TPM_ALG_NULL)
return TPM_RC_SCHEME;

// Keys must have a valid sign or decrypt scheme, or a TPM_ALG_NULL
// scheme
if (keyScheme->scheme != TPM_ALG_NULL
    && (publicArea->objectAttributes.sign == SET
        && !CryptIsSignScheme(keyScheme->scheme)
        |
        publicArea->objectAttributes.decrypt == SET
        && !CryptIsDecryptScheme(keyScheme->scheme)
    ))
return TPM_RC_SCHEME;

// Special checks for an ECC key
if (publicArea->type == TPM_ALG_ECC)
{
    TPM_ECC_CURVE  curveID = publicArea->parameters.eccDetail.curveID;
    const TPMT_ECC_SCHEME  *curveScheme = CryptGetCurveSignScheme(curveID);
    // The curveId must be valid or the unmarshaling is busted.
pAssert(curveScheme != NULL);

    // If the curveID requires a specific scheme, then the key must select
    // the same scheme
    if (curveScheme->scheme != TPM_ALG_NULL)
    {
        if (keyScheme->scheme != curveScheme->scheme)
return TPM_RC_SCHEME;
        // The scheme can allow any hash, or not...
        if (curveScheme->details.any.hashAlg != TPM_ALG_NULL
            && (keyScheme->details.anySig.hashAlg != curveScheme->details.any.hashAlg)
        )
return TPM_RC_SCHEME;
        // For now, the KDF must be TPM_ALG_NULL
    if (publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
return TPM_RC_KDF;
    }

    // Checks for a storage key (restricted + decryption)
    if (publicArea->objectAttributes.restricted == SET
        && publicArea->objectAttributes.decrypt == SET)
    {
        // A storage key must have a valid protection key
        if (publicArea->parameters.asymDetail.symmetric.algorithm
            == TPM_ALG_NULL)
return TPM_RC_SYMMETRIC;
        // A storage key must have a null scheme
        if (publicArea->parameters.asymDetail.scheme.scheme != TPM_ALG_NULL)
return TPM_RC_SCHEME;
        // A storage key must match its parent algorithms unless
        // it is duplicable or a primary (including Temporary Primary Objects)
        if (HandleGetType(parentHandle) != TPM_HT_PERMANENT
            && publicArea->objectAttributes.fixedParent == SET
        )
{  
    // If the object to be created is a storage key, and is fixedParent,
    // its crypto set has to match its parent's crypto set. TPM_RC_TYPE,
    // TPM_RC_HASH or TPM_RCASYMMETRIC may be returned at this point
    return EqualCryptSet(publicArea,
        &(ObjectGet(parentHandle)->publicArea));
}

else
{
    // Non-storage keys must have TPM_ALG_NULL for the symmetric algorithm
    if((publicArea->parameters.asymDetail.symmetric.algorithm
        != TPM_ALG_NULL)
        return TPM_RCASYMMETRIC;

    } // End of asymmetric decryption key checks

} // End of asymmetric checks

else if(publicArea->type == TPM_ALGKEYEDHASH)
{
    TPMHKEYEDHASH_SCHEME *scheme
    = &publicArea->parameters.keyedHashDetail.scheme;
    // If both sign and decrypt are set the scheme must be TPM_ALG_NULL
    // and the scheme selected when the key is used.
    // If neither sign nor decrypt is set, the scheme must be TPM_ALG_NULL
    // because this is a data object.
    if((publicArea->objectAttributes.sign == SET
        && publicArea->objectAttributes.decrypt == SET)
        || (  
            publicArea->objectAttributes.sign == CLEAR
            && publicArea->objectAttributes.decrypt == CLEAR
        )
    )
    {
        if(scheme->scheme != TPM_ALG_NULL)
            return TPM_RCSCHEME;
        return TPM_RC_SUCCESS;
    }
    // If this is a decryption key, make sure that is is XOR and that there
    // is a KDF
    else if(publicArea->objectAttributes.decrypt)
    {
        if(scheme->scheme != TPM_ALG_XOR
            || scheme->details.xor.hashAlg == TPM_ALG_NULL)
            return TPM_RCSCHEME;
        if(scheme->details.xor.kdf == TPM_ALG_NULL)
            return TPM_RC_KDF;
        return TPM_RC_SUCCESS;
    }
    // only supported signing scheme for keyedHash object is HMAC
    if((scheme->scheme != TPM_ALG_HMAC
        || scheme->details.hmac.hashAlg == TPM_ALG_NULL)
        return TPM_RCSCHEME;
    }
    // end of the checks for keyedHash
    return TPM_RC_SUCCESS;
}

else if (publicArea->type == TPM_ALG_SYMCIPHER)
{
    // Must be a decrypting key and may not be a signing key
    if(publicArea->objectAttributes.decrypt == CLEAR
        || publicArea->objectAttributes.sign == SET
    )
        return TPM_RC_ATTRIBUTES;
}
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().

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ASYMMETRIC</td>
<td>non-duplicable storage key and its parent have different public params</td>
</tr>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>*fixedTPM, fixedParent, or encryptedDuplication attributes are inconsistent between themselves or with those of the parent object; inconsistent restricted, decrypt and sign attributes; attempt to inject sensitive data for an asymmetric key; attempt to create a symmetric cipher key that is not a decryption key</td>
</tr>
<tr>
<td>TPM_RC_HASH</td>
<td>non-duplicable storage key and its parent have different name algorithm</td>
</tr>
<tr>
<td>TPM_RC_KDF</td>
<td>incorrect KDF specified for decrypting keyed hash object</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>invalid key size values in an asymmetric key public area</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>inconsistent attributes decrypt, sign, restricted and key’s scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>authPolicy size does not match digest size of the name algorithm in publicArea</td>
</tr>
<tr>
<td>TPM_RC_SYMMETRIC</td>
<td>a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL</td>
</tr>
<tr>
<td>TPM_RC_TYPE</td>
<td>unexpected object type; or non-duplicable storage key and its parent have different types</td>
</tr>
</tbody>
</table>

```c
TPM_RC
PublicAttributesValidation(
    BOOL                 load,            // IN: TRUE if load checks, FALSE if
    TPMI_DH_OBJECT       parentHandle,    // IN: input parent handle
    TPMT_PUBLIC         *publicArea       // IN: public area of the object
)
{
    OBJECT              *parentObject = NULL;

    if(HandleGetType(parentHandle) != TPM_HT_PERMANENT)
        parentObject = ObjectGet(parentHandle);

    // Check authPolicy digest consistency
    if(   publicArea->authPolicy.t.size != 0
        && (   publicArea->authPolicy.t.size
        != CryptGetHashDigestSize(publicArea->nameAlg)
        )
    )
        return TPM_RC_SIZE;

    // If the parent is fixedTPM (including a Primary Object) the object must have
    // the same value for fixedTPM and fixedParent
    if(   parentObject == NULL
        || parentObject->publicArea.objectAttributes.fixedTPM == SET)
    {
        }```
if (publicArea->objectAttributes.fixedParent != publicArea->objectAttributes.fixedTPM)
    return TPM_RC_ATTRIBUTES;
else
    // The parent is not fixedTPM so the object can't be fixedTPM
    if (publicArea->objectAttributes.fixedTPM == SET)
        return TPM_RC_ATTRIBUTES;
    // A restricted object cannot be both sign and decrypt and it can't be neither
    // sign not decrypt
    if (publicArea->objectAttributes.restricted == SET &&
        publicArea->objectAttributes.decrypt == publicArea->objectAttributes.sign)
        return TPM_RC_ATTRIBUTES;
    // A fixedTPM object can not have encryptedDuplication bit SET
    if (publicArea->objectAttributes.fixedTPM == SET &&
        publicArea->objectAttributes.encryptedDuplication == SET)
        return TPM_RC_ATTRIBUTES;
    // If a parent object has fixedTPM CLEAR, the child must have the
    // same encryptedDuplication value as parent.
    // Primary objects are considered to have a fixedTPM parent (the seeds).
    if (parentObject != NULL &&
        parentObject->publicArea.objectAttributes.fixedTPM == CLEAR)
        // Get here if parent is not fixed TPM
        if (publicArea->objectAttributes.encryptedDuplication !=
            publicArea->objectAttributes.encryptedDuplication)
            return TPM_RC_ATTRIBUTES;
    return SchemeChecks(load, parentHandle, publicArea);
}

7.6.1.4 FillInCreationData()

Fill in creation data for an object.

void FillInCreationData(
    TPMI_DH_OBJECT       parentHandle, // IN: handle of parent
    TPMI_ALG_HASH        nameHashAlg, // IN: name hash algorithm
    TPML_PCR_SELECTION  *creationPCR, // IN: PCR selection
    TPM2B_DATA          *outsideData, // IN: outside data
    TPM2B_CREATION_DATA *outCreation, // OUT: creation data for output
    TPM2B_DIGEST        *creationDigest // OUT: creation digest
)
{
    BYTE                 creationBuffer[ sizeof(TPMS_CREATION_DATA)];
    BYTE                *buffer;
    HASH_STATE hashState;
    // Fill in TPMS_CREATION_DATA in outCreation
    // Compute PCR digest
    PCRComputeCurrentDigest(nameHashAlg, creationPCR,
        &outCreation->t.creationData.pcrDigest);
    // Put back PCR selection list
    outCreation->t.creationData.pcrSelect = *creationPCR;
}
// Get locality
outCreation->t.creationData.locality = LocalityGetAttributes(_plat__LocalityGet());

outCreation->t.creationData.parentNameAlg = TPM_ALG_NULL;

// If the parent is is either a primary seed or TPM_ALG_NULL, then the Name
// and QN of the parent are the parent's handle.
if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
{
    BYTE *buffer = &outCreation->t.creationData.parentName.t.name[0];
    outCreation->t.creationData.parentName.t.size =
        TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);

    // Parent qualified name of a Temporary Object is the same as parent's
    // name
    MemoryCopy2B(&outCreation->t.creationData.parentQualifiedName.b,
        &outCreation->t.creationData.parentName.b);
}
else // Regular object
{
    OBJECT *parentObject = ObjectGet(parentHandle);

    // Set name algorithm
    outCreation->t.creationData.parentNameAlg =
        parentObject->publicArea.nameAlg;

    // Copy parent name
    outCreation->t.creationData.parentName = parentObject->name;

    // Copy parent qualified name
    outCreation->t.creationData.parentQualifiedName =
        parentObject->qualifiedName;
}

// Copy outside information
outCreation->t.creationData.outsideInfo = *outsideData;

// Marshal creation data to canonical form
buffer = creationBuffer;
outCreation->t.size = TPMS_CREATION_DATA_Marshal(&outCreation->t.creationData,
    &buffer, NULL);

// Compute hash for creation field in public template
creationDigest->t.size = CryptStartHash(nameHashAlg, &hashState);
CryptUpdateDigest(&hashState, outCreation->t.size, creationBuffer);
CryptCompleteHash2B(&hashState, &creationDigest->b);
return;
}

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>static UINT16 GetIV2BSize(</td>
<td></td>
</tr>
<tr>
<td>TPM_HANDLE protectorHandle // IN: the protector handle</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
</tbody>
</table>
Trusted Platform Module Library

Part 4: Supporting Routines

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.

```c
TPM2B_SEED*
GetSeedForKDF(
    TPM_HANDLE           protectorHandle, // IN: the protector handle
    TPM2B_SEED          *seedIn     // IN: the optional input seed
)
{
    OBJECT              *protector = NULL; // Pointer to the protector
    TPM_ALG_ID          symAlg;
    UINT16              keyBits;
    // Determine the symmetric algorithm and size of key
    if(protectorHandle == TPM_RH_NULL)
    {
        // Use the context encryption algorithm and key size
        symAlg = CONTEXT_ENCRYPT_ALG;
        keyBits = CONTEXT_ENCRYPT_KEY_BITS;
    }
    else
    {
        protector = ObjectGet(protectorHandle);
        symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
        keyBits= protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
    }
    // The IV size is a UINT16 size field plus the block size of the symmetric
    // algorithm
    return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
}
```

7.6.1.7  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.

```c
static void
ComputeProtectionKeyParms(
    TPM_HANDLE protectorHandle,  // IN: the protector handle
    TPM_ALG_ID hashAlg,          // IN: hash algorithm for KDFa
    TPM2B_NAME *name,            // IN: name of the object
    TPM2B_SEED *seedIn,          // IN: optional seed for duplication
    TPM_ALG_ID *symAlg,          // OUT: the symmetric algorithm
    UINT16 *keyBits,             // OUT: the symmetric key size in bits
    TPM2B_SYM_KEY *symKey        // OUT: the symmetric key
)
{
    TPM2B_SEED *seed = NULL;
    OBJECT *protector = NULL;  // Pointer to the protector

    // Determine the algorithms for the KDF and the encryption/decryption
    // For TPM_RH_NULL, using context settings
    if (protectorHandle == TPM_RH_NULL)
    {
        // Use the context encryption algorithm and key size
        *symAlg = CONTEXT_ENCRYPT_ALG;
        symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
        *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
    }
    else
    {
        TPMT_SYM_DEF_OBJECT *symDef;
        protector = ObjectGet(protectorHandle);
        symDef = &protector->publicArea.parameters.asymDetail.symmetric;
        *symAlg = symDef->algorithm;
        *keyBits = symDef->keyBits.sym;
        symKey->t.size = (*keyBits + 7) / 8;
    }

    // Get seed for KDF
    seed = GetSeedForKDF(protectorHandle, seedIn);

    // KDFa to generate symmetric key and IV value
    KDFa(hashAlg, (TPM2B *)seed, "STORAGE", (TPM2B *)name, NULL,
    symKey->t.size * 8, symKey->t.buffer, NULL);

    return;
}
```

### 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.

```c
static void
ComputeOuterIntegrity(
    TPM2B_NAME *name,            // IN: the name of the object
    TPM_HANDLE protectorHandle,   // IN: The handle of the object
    // that provides protection. For
    // object, it is parent handle.
    // For credential, it is the handle
    // of encrypt object. For a
```
7.6.1.9 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap.

```c
static void ComputeInnerIntegrity(
    TPM_ALG_ID          hashAlg,  // IN: hash algorithm for inner wrap
    TPM2B_NAME          *name,    // IN: the name of the object
    UINT16              dataSize, // IN: the size of sensitive data
    BYTE                *sensitiveData, // IN: sensitive data
    TPM2B_DIGEST        *integrity  // OUT: inner integrity
)
{
    HASH_STATE          hashState;

    // Start hash and get the size of the digest which will become the integrity
    integrity->t.size = CryptStartHash(hashAlg, &hashState);

    // Adding the marshaled sensitive area to the integrity value
    CryptUpdateDigest(&hashState, dataSize, sensitiveData);

    // Compute HMAC
    CryptCompleteHMAC2B(&hashState, &integrity->b);

    return;
}
```
// Adding name
CryptUpdateDigest2B(&hashState, &name->b);

// Compute hash
CryptCompleteHash2B(&hashState, &integrity->b);

return;
}
}

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 assume the sensitive data starts at address (innerBuffer + integrity size). This function integrity at the beginning of the inner buffer it returns the total size of buffer with the inner wrap.

static UINT16 ProduceInnerIntegrity(
    TPM2B_NAME              *name,
    // IN: the name of the object
    TPM_ALG_ID              hashAlg,
    // IN: hash algorithm for inner wrap
    UINT16                  dataSize,
    // IN: the size of sensitive data,
    // excluding the leading integrity buffer size
    BYTE                    *innerBuffer
    // IN/OUT: inner buffer with sensitive data in it. At input, the leading bytes of this buffer is reserved for integrity
)
{
    BYTE *sensitiveData; // pointer to the sensitive data
    TPM2B_DIGEST integrity;
    UINT16 integritySize;
    BYTE *buffer; // Auxiliary buffer pointer

    // sensitiveData points to the beginning of sensitive data in innerBuffer
    integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
    sensitiveData = innerBuffer + integritySize;

    ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);

    // Add integrity at the beginning of inner buffer
    buffer = innerBuffer;
    TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);

    return dataSize + integritySize;
}

7.6.1.11 CheckInnerIntegrity()

This function check integrity of inner blob.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_INTEGRITY</td>
<td>if the outer blob integrity is bad</td>
</tr>
<tr>
<td>unmarshal errors</td>
<td>unmarshal errors while unmarshaling integrity</td>
</tr>
</tbody>
</table>

static TPM_RC CheckInnerIntegrity();
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
BYTE *outerBuffer // IN/OUT: outer buffer with
    // sensitive data in it
}

TPM_ALG_ID symAlg;
UINT16 keyBits;
TPM2B_SYM_KEY symKey;
TPM2B_IV ivRNG; // IV from RNG
TPM2B_IV *iv = NULL;
UINT16 ivSize = 0; // size of iv area, including the size field
BYTE *sensitiveData; // pointer to the sensitive data

TPM2B_DIGEST integrity;
UINT16 integritySize;
BYTE *buffer; // Auxiliary buffer pointer

// Compute the beginning of sensitive data. The outer integrity should
// always exist if this function function is called to make an outer wrap
integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
sensitiveData = outerBuffer + integritySize;

// If iv is used, adjust the pointer of sensitive data and add iv before it
if(useIV)
{
    ivSize = GetIV2BSize(protector);

    // Generate IV from RNG. The iv data size should be the total IV area
    // size minus the size of size field
    ivRNG.t.size = ivSize - sizeof(UINT16);
    CryptGenerateRandom(ivRNG.t.size, ivRNG.t.buffer);

    // Marshal IV to buffer
    buffer = sensitiveData;
    TPM2B_IVMarshal(&ivRNG, &buffer, NULL);

    // adjust sensitive data starting after IV area
    sensitiveData += ivSize;

    // Use iv for encryption
    iv = &ivRNG;
}

// Compute symmetric key parameters for outer buffer encryption
ComputeProtectionKeyParms(protector, hashAlg, name, seed,
    &symAlg, &keyBits, &symKey);

// Encrypt inner buffer in place
CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
    TPM_ALG_CFB, symKey.t.buffer, iv, dataSize,
    sensitiveData);

// Compute outer integrity. Integrity computation includes the optional IV
// area
ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
    outerBuffer + integritySize, &integrity);

// Add integrity at the beginning of outer buffer
buffer = outerBuffer;
TPM2B_DIGESTMarshal(&integrity, &buffer, NULL);

// return the total size in outer wrap
return dataSize + integritySize + ivSize;
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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_INSUFFICIENT</td>
<td>error during sensitive data unmarshaling</td>
</tr>
<tr>
<td>TPM_RC_INTEGRITY</td>
<td>sensitive data integrity is broken</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>error during sensitive data unmarshaling</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>IV size for CFB does not match the encryption algorithm block size</td>
</tr>
</tbody>
</table>

```c
TPM_RC UnwrapOuter(
    TPM_HANDLE protector, // IN: The handle of the object
    TPM2B_NAME *name, // IN: the name of the object
    TPM_ALG_ID hashAlg, // IN: hash algorithm for outer wrap
    TPM2B_SEED *seed, // IN: an external seed may be provided for duplication blob.
    TPM2B_DIGEST integrityToCompare;

    TPM_RC result;
    TPM_ALG_ID symAlg;
    TPM2B_SYM_KEY symKey;
    UINT16 keyBits;
    TPM2B_IV ivIn; // input IV retrieved from input buffer
    TPM2B_IV *iv = NULL;
    BYTE *sensitiveData; // pointer to the sensitive data
    TPM2B_DIGEST integrity;
    INT32 size;

    // Unmarshal integrity
    sensitiveData = outerBuffer;
    size = (INT32) dataSize;
    result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Compute integrity to compare
    ComputeOuterIntegrity(name, protector, hashAlg, seed,
        (UINT16) size, sensitiveData,
        &integrityToCompare);

    // Compare outer blob integrity
    if(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
```
// Get the symmetric algorithm parameters used for encryption
ComputeProtectionKeyParms(protector, hashAlg, name, seed,
   &symAlg, &keyBits, &symKey);

// Retrieve IV if it is used
if(useIV)
   {
      result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
      if(result != TPM_RC_SUCCESS)
         return result;
      // The input iv size for CFB must match the encryption algorithm block size
      if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
         return TPM_RC_VALUE;
      iv = &ivIn;
   }

// Decrypt private in place
CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
   TPM_ALG_CFB, symKey.t.buffer, iv,
   (UINT16) size, sensitiveData);
return TPM_RC_SUCCESS;

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.

void SensitiveToPrivate(
   TPM_T SENSITIVE *sensitive, // IN: sensitive structure
   TPM2B_NAME *name, // IN: the name of the object
   TPM_HANDLE parentHandle, // IN: The parent's handle
   TPM_ALG_ID nameAlg, // IN: hash algorithm in public area. This parameter is used when parentHandle is NULL, in which case the object is temporary.
   TPM2B_PRIVATE *outPrivate // OUT: output private structure
)
{
   BYTE *buffer; // Auxiliary buffer pointer
   BYTE *sensitiveData; // pointer to the sensitive data
   UINT16 dataSize; // data blob size
   TPM_ALG_HASH hashAlg; // hash algorithm for integrity
   UINT16 integritySize;
   UINT16 ivSize;
   pAssert(name != NULL && name->t.size != 0);

   // Find the hash algorithm for integrity computation
   if(parentHandle == TPM_RH_NULL)
      {
         // For Temporary Object, using self name algorithm
         hashAlg = nameAlg;
   
   return TPM_RC_SUCCESS;
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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_INTEGRITY</td>
<td>if the private area integrity is bad</td>
</tr>
<tr>
<td>TPM_RC_SENSITIVE</td>
<td>unmarshal errors while unmarshaling TPMS_ENCRYPT from input private</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>outer wrapper does not have an iV of the correct size</td>
</tr>
</tbody>
</table>

TPM RC
823 PrivateToSensitive(  
824    TPM2B_PRIVATE *inPrivate,  // IN: input private structure}
TPM2B_NAME *name,  // IN: the name of the object
TPM_HANDLE parentHandle,  // IN: The parent's handle
TPM_ALG_ID nameAlg,    // IN: hash algorithm in public

// area. It is passed separately because we only pass name, rather than the whole public
// area of the object. This parameter is used in the following two cases: 1. primary objects. 2. duplication
// cases, this parameter will be ignored
TPMT_SENSITIVE *sensitive  // OUT: sensitive structure

{  TPM_RC result;
BYTE *buffer;
INT32 size;
BYTE *sensitiveData;  // pointer to the sensitive data
UINT16 dataSize;
UINT16 dataSizeInput;
TPMI_ALG_HASH hashAlg;  // hash algorithm for integrity
OBJECT *parent = NULL;

UINT16 integritySize;
UINT16 ivSize;

// Make sure that name is provided
pAssert(name != NULL && name->t.size != 0);

// Find the hash algorithm for integrity computation
if(parentHandle == TPM_RH_NULL)
{
    // For Temporary Object, using self name algorithm
    hashAlg = nameAlg;
}
else
{
    // Otherwise, using parent's name algorithm
    hashAlg = ObjectGetNameAlg(parentHandle);
}

// unwrap outer
result = UnwrapOuter(parentHandle, name, hashAlg, NULL, TRUE,
inPrivate->t.size, inPrivate->t.buffer);

if(result != TPM_RC_SUCCESS)

    return result;

// Compute the inner integrity size.
integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);

// Get iv size
ivSize = GetIV2BSize(parentHandle);

// The starting of sensitive data and data size without outer wrapper
sensitiveData = inPrivate->t.buffer + integritySize + ivSize;
dataSize = inPrivate->t.size - integritySize - ivSize;

// Unmarshal input data size
buffer = sensitiveData;
size = (INT32)dataSize;

result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
if(result != TPM_RC_SUCCESS)

    return result; -
7.6.1.16 SensitiveToDuplicate()

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

1) marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE
2) apply inner wrap to the sensitive area if required
3) apply outer wrap if required

```c
void SensitiveToDuplicate(  
    TPMT_SENSITIVE          *sensitive,   // IN: sensitive structure  
    TPM2B_NAME              *name,       // IN: the name of the object  
    TPM_HANDLE              parentHandle,  // IN: The new parent's handle  
    TPM_ALG_ID              nameAlg,     // IN: hash algorithm in public area. It is passed separately  
                                // because we only pass name, rather than the whole public area of the object.  
    TPM2B_SEED              *seed,        // IN: the external seed.  
                                // If external seed is provided with size of 0, no outer wrap should be applied  
                                // because we only pass name, rather than the whole public area of the object.  
    TPMT_SYM_DEF_OBJECT     *symDef,      // IN: Symmetric key definition.  
                                // If the symmetric key algorithm is NULL, no inner wrap should be applied  
    TPM2B_DATA              *innerSymKey, // IN: a symmetric key may be provided to encrypt the inner wrap of a duplication blob.  
    TPM2B_PRIVATE           *outPrivate  // OUT: output private structure  
)  
{
```

```c
    BYTE            *buffer;   // Auxiliary buffer pointer
    BYTE            *sensitiveData; // pointer to the sensitive data
    TPM_ALG_HASH    outerHash = TPM_ALG_NULL; // The hash algorithm for outer wrap
    TPM_ALG_HASH    innerHash = TPM_ALG_NULL; // The hash algorithm for inner wrap
    UINT16          dataSize;   // Data blob size
    BOOL            doInnerWrap = FALSE;
    BOOL            doOuterWrap = FALSE;
```

```c
    // Make sure that name is provided
    pAssert(name != NULL && name->t.size != 0);
```
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     else
1002     {
1003         // assume the input key size should matches the symmetric definition
1004         pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1005     }
1006 }
1007
1008 // TPM generates symmetric encryption. Set the flag to FALSE
1009 symKeyInput = FALSE;
1010
// Encrypt inner buffer in place
CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
    symDef->keyBits.sym, TPM_ALG_CFB,
    innerSymKey->t.buffer, NULL, dataSize,
    innerBuffer);

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

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

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

    return;

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_INSUFFICIENT</td>
<td>unmarshaling sensitive data from inPrivate failed</td>
</tr>
<tr>
<td>TPM_RC_INTEGRITY</td>
<td>inPrivate data integrity is broken</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>unmarshaling sensitive data from inPrivate failed</td>
</tr>
</tbody>
</table>

TPM_RC

DuplicateToSensitive(

    TPM2B_PRIVATE *inPrivate,  // IN: input private structure
    TPM2B_NAME *name,         // IN: the name of the object
    TPM_HANDLE parentHandle, // IN: The parent's handle
    TPM_ALG_ID nameAlg,      // IN: hash algorithm in public
    TPM2B_SEED *seed,        // IN: an external seed may be
    TPM2B_DATA *innerSymKey, // IN: a symmetric key may be
    TPMT_SYM_DEF_OBJECT *symDef, // IN: Symmetric key definition.
    TPMT_SENSITIVE *sensitive // OUT: sensitive structure
)
1056 )
1057 {
1058     TPM_RC result;
1059     BYTE *buffer;
1060     INT32 size;
1061     BYTE *sensitiveData; // pointer to the sensitive data
1062     UINT16 dataSize;
1063     UINT16 dataSizeInput;
1064
1065     // Make sure that name is provided
1066     pAssert(name != NULL && name->t.size != 0);
1067
1068     // Make sure symDef and innerSymKey are not NULL
1069     pAssert(symDef != NULL && innerSymKey != NULL);
1070
1071     // Starting of sensitive data
1072     sensitiveData = inPrivate->t.buffer;
1073     dataSize = inPrivate->t.size;
1074
1075     // Find out if inner wrap is applied
1076     if(seed->t.size != 0)
1077     {
1078         TPMI_ALG_HASH outerHash = TPM_ALG_NULL;
1079
1080         // Use parent nameAlg as outer hash algorithm
1081         outerHash = ObjectGetNameAlg(parentHandle);
1082         result = UnwrapOuter(parentHandle, name, outerHash, seed, FALSE,
1083             dataSize, sensitiveData);
1084         if(result != TPM_RC_SUCCESS)
1085             return result;
1086
1087         // Adjust sensitive data pointer and size
1088         sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1089         dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1090     }
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;
result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
if(result != TPM_RC_SUCCESS) return result;
if((dataSizeInput + sizeof(UINT16)) != dataSize) return TPM_RC_SIZE;
// Unmarshal sensitive buffer to sensitive structure
result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
if(result != TPM_RC_SUCCESS) return result;
if(size != 0) return TPM_RC_SIZE;

// Always remove trailing zeros at load so that it is not necessary to check
// each time auth is checked.
MemoryRemoveTrailingZeros(&(sensitive->authValue));
return TPM_RC_SUCCESS;

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

void SecretToCredential(
    TPM2B_DIGEST *secret, // IN: secret information
    TPM2B_NAME *name, // IN: the name of the object
    TPM2B_SEED *seed, // IN: an external seed.
    TPM_HANDLE protector, // IN: The protector's handle
    TPM2B_ID_OBJECT *outIDObject // OUT: output credential
)
{
    BYTE *buffer; // Auxiliary buffer pointer
    BYTE *sensitiveData; // pointer to the sensitive data
    TPM_ALG_HASH outerHash; // The hash algorithm for outer wrap
    UINT16 dataSize; // data blob size

    pAssert(secret != NULL && outIDObject != NULL);

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

    // Marshal secret area to credential buffer, leave space for integrity
    sensitiveData = outIDObject->t.credential + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);

    // Marshal secret area
    buffer = sensitiveData;
dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);

    // Apply outer wrap
    outIDObject->t.size = ProduceOuterWrap(protector,
        name,
        outerHash,
        seed,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
        &buffer,
        &outerHash,
        FALSE,
        dataSize,
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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_INSUFFICIENT</td>
<td>error during credential unmarshaling</td>
</tr>
<tr>
<td>TPM_RC_INTEGRITY</td>
<td>credential integrity is broken</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>error during credential unmarshaling</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>IV size does not match the encryption algorithm block size</td>
</tr>
</tbody>
</table>

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

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

    // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
    result = UnwrapOuter(protector, name, outerHash, seed, FALSE,  
        inIDObject->t.size, inIDObject->t.credential);  
    if(result != TPM_RC_SUCCESS)  
        return result;  

    // Compute the beginning of sensitive data
    sensitiveData = inIDObject->t.credential  
        + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);  
    dataSize = inIDObject->t.size  
        - (sizeof(UINT16) + CryptGetHashDigestSize(outerHash));  

    // Unmarshal secret buffer to TPM2B_DIGEST structure
    buffer = sensitiveData;  
    size = (INT32) dataSize;  
    result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);  
    if(result != TPM_RC_SUCCESS)  
        return result;  
    if(size != 0)  
        return TPM_RC_SIZE;  
    return TPM_RC_SUCCESS;
```
8 Subsystem

8.1 CommandAudit.c

8.1.1 Introduction

This file contains the functions that support command audit.

8.1.2 Includes

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

8.1.3 Functions

8.1.3.1 CommandAuditPreInstall_Init()

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

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

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

    // TPM_CC_SetCommandCodeAuditStatus always being audited
    if(CommandIsImplemented(TPM_CC_SetCommandCodeAuditStatus))
        CommandAuditSet(TPM_CC_SetCommandCodeAuditStatus);

    // Set initial command audit hash algorithm to be context integrity hash
    // algorithm
    gp.auditHashAlg = CONTEXT_INTEGRITY_HASH_ALG;

    // Set up audit counter to be 0
    gp.auditCounter = 0;

    // Write command audit persistent data to NV
    NvWriteReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
    NvWriteReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
    NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);

    return;
}
```

8.1.3.2 CommandAuditStartup()

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

```c
void CommandAuditStartup(
    STARTUP_TYPE                    type
                                    // IN: start up type
)
{
    if(type == SU_RESET)
```
8.1.3.3 CommandAuditSet()

This function will SET the audit flag for a command. This function will not SET the audit flag for a command that is not implemented. This ensures that the audit status is not SET when TPM2_GetCapability() is used to read the list of audited commands.

This function is only used by TPM2_SetCommandCodeAuditStatus().

The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>the command code audit status was changed</td>
</tr>
<tr>
<td>FALSE</td>
<td>the command code audit status was not changed</td>
</tr>
</tbody>
</table>

```c
BOOL CommandAuditSet(
    TPM_CC commandCode // IN: command code
) {
    UINT32 bitPos;
    // Only SET a bit if the corresponding command is implemented
    if(CommandIsImplemented(commandCode))
        // Can’t audit shutdown
        if(commandCode != TPM_CC_Shutdown)
            bitPos = commandCode - TPM_CC_FIRST;
            if(!BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
                // Set bit
                BitSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
                return TRUE;
    return FALSE;
}
```

8.1.3.4 CommandAuditClear()

This function will CLEAR the audit flag for a command. It will not CLEAR the audit flag for TPM_CC_SetCommandCodeAuditStatus().

This function is only used by TPM2_SetCommandCodeAuditStatus().

The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.
### CommandAuditClear()

```c
BOOL CommandAuditClear(
    TPM_CC commandCode // IN: command code
)
{
    UINT32 bitPos;
    // Do nothing if the command is not implemented
    if (CommandIsImplemented(commandCode))
    {
        // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be cleared
        if (commandCode != TPM_CC_SetCommandCodeAuditStatus)
        {
            bitPos = commandCode - TPM_CC_FIRST;
            if (BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
            {
                // Clear bit
                BitClear(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
                return TRUE;
            }
        }
        // No change
        return FALSE;
    }
}
```

#### 8.1.3.5 CommandAuditIsRequired()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if command is audited</td>
</tr>
<tr>
<td>FALSE</td>
<td>if command is not audited</td>
</tr>
</tbody>
</table>

```c
BOOL CommandAuditIsRequired(
    TPM_CC commandCode // IN: command code
)
{
    UINT32 bitPos;
    bitPos = commandCode - TPM_CC_FIRST;
    // Check the bit map. If the bit is SET, command audit is required
    if (!((gp.auditComands[bitPos/8] & (1 << (bitPos % 8))) != 0))
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

#### 8.1.3.6 CommandAuditCapGetCCList()

This function returns a list of commands that have their audit bit SET.
The list starts at the input `commandCode`.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more command code available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available command code has been returned</td>
</tr>
</tbody>
</table>

```
TPMI_YES_NO
CommandAuditCapGetCCList(
    TPM_CC commandCode, // IN: start command code
    UINT32 count,       // IN: count of returned TPM_CC
    TPML_CC *commandList // OUT: list of TPM_CC
)
{
    TPMI_YES_NO more = NO;
    UINT32 i;

    // Initialize output handle list
    commandList->count = 0;

    // The maximum count of command we may return is MAX_CAP_CC
    if(count > MAX_CAP_CC) count = MAX_CAP_CC;

    // If the command code is smaller than TPM_CC_FIRST, start from TPM_CC_FIRST
    if(commandCode < TPM_CC_FIRST) commandCode = TPM_CC_FIRST;

    // Collect audit commands
    for(i = commandCode; i <= TPM_CC_LAST; i++)
    {
        if(CommandAuditIsRequired(i))
        {
            if(commandList->count < count)
            {
                // If we have not filled up the return list, add this command
                // code to it
                commandList->commandCodes[commandList->count] = i;
                commandList->count++;
            }
            else
            {
                // If the return list is full but we still have command
                // available, report this and stop iterating
                more = YES;
                break;
            }
        }
    }
    return more;
}
```

### 8.1.3.7 CommandAuditGetDigest

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

```
void
CommandAuditGetDigest(
    TPM2B_DIGEST *digest       // OUT: command digest
)
{
    // Calculate digest
    TPM2B_DIGEST digest = ...
```
### 8.2 DA.c

#### 8.2.1 Introduction

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

#### 8.2.2 Includes and Data Definitions

```c
#define DA_C
#include "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().

```c
void DAPreInstall_Init(void)
{
    gp.failedTries = 0;
    gp.maxTries = 3;
    gp.recoveryTime = 1000; // in seconds (~16.67 minutes)
    gp.lockoutRecovery = 1000; // in seconds
    gp.lockOutAuthEnabled = TRUE; // Use of lockoutAuth is enabled
    // Record persistent DA parameter changes to NV
    NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
    NvWriteReserved(NV_MAX_TRIES, &gp.maxTries);
    NvWriteReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
    NvWriteReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
    NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
    return;
}
```
8.2.2.2 DAStartup()

This function is called by TPM2_Startup() to initialize the DA parameters. In the case of Startup(CLEAR), use of lockoutAuth will be enabled if the lockout recovery time is 0. Otherwise, lockoutAuth will not be enabled until the TPM has been continuously powered for the lockoutRecovery time.

This function requires that NV be available and not rate limiting.

```c
21 void 22 DAStartup( 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 // If DA has not been disabled and the previous shutdown is not orderly 37 // failedTries is not already at its maximum then increment 'failedTries' 38 if(!gp.recoveryTime != 0 39 && g_prevOrderlyState == SHUTDOWN_NONE 40 && gp.failedTries < gp.maxTries) 41 { 42 gp.failedTries++; 43 // Record the change to NV 44 NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries); 45 } 46 } 47 // Reset self healing timers 48 s_selfHealTimer = g_time; 49 s_lockoutTimer = g_time; 50 51 return;
52 }

8.2.2.3 DARegisterFailure

This function is called when a authorization failure occurs on an entity that is subject to dictionary-attack protection. When a DA failure is triggered, register the failure by resetting the relevant self-healing timer to the current time.

```c
54 void 55 DARegisterFailure( 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 DASelfHeal()

This function is called to check if sufficient time has passed to allow decrement of failedTries or to re-enable use of lockoutAuth.

This function should be called when the time interval is updated.

```c
void DASelfHeal(void)
{
    // Regular auth self healing logic
    // If no failed authorization tries, do nothing. Otherwise, try to
decrease failedTries
    if (gp.failedTries != 0)
    {
        // if recovery time is 0, DA logic has been disabled. Clear failed tries
        // immediately
        if (gp.recoveryTime == 0)
        {
            gp.failedTries = 0;
        // Update NV record
        NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
        }
    else
    {
        UINT64 decreaseCount;
        // In the unlikely event that failedTries should become larger than
        // maxTries
        if (gp.failedTries > gp.maxTries)
        {
            gp.failedTries = gp.maxTries;
        // How much can failedTried be decreased
        decreaseCount = ((g_time - s_selfHealTimer) / 1000) / gp.recoveryTime;
        if (gp.failedTries <= (UINT32) decreaseCount)
        // should not set failedTries below zero
        gp.failedTries = 0;
        else
        {
            gp.failedTries -= (UINT32) decreaseCount;
        // the cast prevents overflow of the product
        s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
        if (decreaseCount != 0)
        // If there was a change to the failedTries, record the changes
        // to NV
        NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
        }
    }
    
    // LockoutAuth self healing logic
    // If lockoutAuth is enabled, do nothing. Otherwise, try to see if we
    // may enable it
    if (!gp.lockOutAuthEnabled)
    {
        // if lockout authorization recovery time is 0, a reboot is required to
        // re-enable use of lockout authorization. Self-healing would not
        // apply in this case.
        if (gp.lockoutRecovery != 0)
        {
            if (((g_time - s_lockoutTimer)/1000) >= gp.lockoutRecovery)
            {
                gp.lockOutAuthEnabled = TRUE;
                // Record the changes to NV
```
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

```
#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.

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

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

    // Initialize owner, endorsement and lockout auth
    gp.ownerAuth.t.size = 0;
    gp.endorsementAuth.t.size = 0;
    gp.lockoutAuth.t.size = 0;

    // Initialize owner and endorsement policy
    gp.ownerAlg = TPM_ALG_NULL;
    gp.ownerPolicy.t.size = 0;
    gp.endorsementAlg = TPM_ALG_NULL;
    gp.endorsementPolicy.t.size = 0;

    // Initialize ehProof, shProof and phProof
    gp.phProof.t.size = PROOF_SIZE;
    gp.shProof.t.size = PROOF_SIZE;
    gp.ehProof.t.size = PROOF_SIZE;
    CryptGenerateRandom(gp.phProof.t.size, gp.phProof.t.buffer);
    CryptGenerateRandom(gp.shProof.t.size, gp.shProof.t.buffer);
    CryptGenerateRandom(gp.ehProof.t.size, gp.ehProof.t.buffer);

    // Write hierarchy data to NV
    NvWriteReserved(NV_DISABLE_CLEAR, &gp.disableClear);
    NvWriteReserved(NV_EP_SEED, &gp.EPSeed);
    NvWriteReserved(NV_SP_SEED, &gp.SPSeed);
```
8.3.2.2 HierarchyStartup()

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

```c
void HierarchyStartup(
    STARTUP_TYPE type, // IN: start up type
)
{
    // phEnable is SET on any startup
    g_phEnable = TRUE;

    // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
    // TPM_RESTART
    if(type != SU_RESUME)
    {
        gc.platformAuth.t.size = 0;
        gc.platformPolicy.t.size = 0;

        // enable the storage and endorsement hierarchies
gc.shEnable = gc.ehEnable = TRUE;
    }

    // nullProof and nullSeed is updated at every TPM_RESET
    if(type == SU_RESET)
    {
        gr.nullProof.t.size = PROOF_SIZE;
        CryptGenerateRandom(gr.nullProof.t.size,
                            gr.nullProof.t.buffer);

        gr.nullSeed.t.size = PRIMARY_SEED_SIZE;
        CryptGenerateRandom(PRIMARY_SEED_SIZE, gr.nullSeed.t.buffer);
    }

    return;
}
```

8.3.2.3 HierarchyGetProof()

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

```c
TPM2B_AUTH *
HierarchyGetProof(
    TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant
)
{
    switch(hierarchy)
    {
        case TPM_RH_PLATFORM:
```
8.3.2.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```c
TPM2B_SEED *
HierarchyGetPrimarySeed(
    TPMI_RH_HIERARCHY hierarchy // IN: hierarchy
)
{
    switch(hierarchy)
    {
        case TPM_RH_PLATFORM:
            return &gp.PPSeed;
            break;
        case TPM_RH_OWNER:
            return &gp.SPSeed;
            break;
        case TPM_RH_ENDORSEMENT:
            return &gp.EPSeed;
            break;
        case TPM_RH_NULL:
            return &gr.nullSeed;
            break;
        default:
            pAssert(FALSE);
            return NULL;
            break;
    }
}
```

8.3.2.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE: The TPM_RH_NULL hierarchy is always enabled.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>hierarchy is enabled</td>
</tr>
<tr>
<td>FALSE</td>
<td>hierarchy is disabled</td>
</tr>
</tbody>
</table>

```c
BOOL HierarchyIsEnabled(TPMI_RH_HIERARCHY hierarchy) // IN: hierarchy
{
    switch(hierarchy)
    {
        case TPM_RH_PLATFORM:
            if(g_phEnable)
                return TRUE;
            break;
        case TPM_RH_OWNER:
            if(gc.shEnable)
                return TRUE;
            break;
        case TPM_RH_ENDORSEMENT:
            if(gc.ehEnable)
                return TRUE;
            break;
        case TPM_RH_NULL:
            return TRUE;
            break;
        default:
            pAssert(FALSE);
            break;
    }
    return FALSE;
}
```

### 8.4 NV.c

#### 8.4.1 Introduction

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

#### 8.4.2 Includes, Defines and Data Definitions

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

NV Index/evict object iterator value
```
```c
typedef UINT32 NV_ITER;         // type of a NV iterator
#define NV_ITER_INIT 0xFFFFFFFF  // initial value to start an 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().

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

    func_return = _plat__IsNvAvailable();
    if (func_return == 0)
    {
        s_NvIsAvailable = TPM_RC_SUCCESS;
    }
    else if (func_return == 1)
    {
        s_NvIsAvailable = TPM_RC_NV_UNAVAILABLE;
    }
    else
    {
        s_NvIsAvailable = TPM_RC_NV_RATE;
    }
    return;
}
```

8.4.3.2 NvIsAvailable()

This function returns the NV availability parameter.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SUCCESS</td>
<td>NV is available</td>
</tr>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is unavailable because of rate limit</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is inaccessible</td>
</tr>
</tbody>
</table>

```c
TPM_RC NvIsAvailable(void)
{
    return s_NvIsAvailable;
}
```

8.4.3.3 NvCommit

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

```c
BOOL NvCommit(void)
{
    if (_plat__NvCommit() == 0)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```
8.4.3.4 NvReadMaxCount()

This function returns the max NV counter value.

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

8.4.3.5 NvWriteMaxCount()

This function updates the max counter value to NV memory.

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

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.

```c
static UINT32
NvNext(NV_ITER *iter)
{
    NV_ITER currentIter;
    // If iterator is at the beginning of list
    if(*iter == NV_ITER_INIT)
    {
        // Initialize iterator
        *iter = s_evictNvStart;
    }
```
69 // If iterator reaches the end of NV space, or iterator indicates list end
70 if(*iter + sizeof(UINT32) > s_evictNvEnd || *iter == 0)
71     return 0;
72
73 // Save the current iter offset
74 currentIter = *iter;
75
76 // Adjust iter pointer pointing to next entity
77 // Read pointer value
78 _plat__NvMemoryRead(*iter, sizeof(UINT32), iter);
79 if(*iter == 0) return 0;
80
81 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     while((currentAddr = NvNext(&iter)) != 0)
92         endAddr = currentAddr;
93     if(endAddr != s_evictNvStart)
94         {
95             // Read offset
96             endAddr -= sizeof(UINT32);
97             _plat__NvMemoryRead(endAddr, sizeof(UINT32), &endAddr);
98         }
99     return endAddr;
102 }

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);
8.4.4.6 NvGetCounterSize

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

```c
static UINT32
NvGetCounterSize (void)
{
    // It takes an offset field, a handle and the sizeof(NV_INDEX) and
    // sizeof(UINT64) for counter data
    return sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + sizeof(UINT64) + sizeof(UINT32);
}
```

8.4.4.7 NvTestSpace()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>space available</td>
</tr>
<tr>
<td>FALSE</td>
<td>no enough space</td>
</tr>
</tbody>
</table>

```c
static BOOL
NvTestSpace(  
    UINT32 size,  // IN: size of the entity to be added  
    BOOL isIndex // IN: TRUE if the entity is an index  
)
{
    UINT32 remainByte = NvGetFreeByte();
    if(isIndex)
    {
        // Get the number of persistent objects allocated
        UINT32 persistentNum = NvCapGetPersistentNumber();
        // If we have not allocated the requisite number of evict objects, then we
        // need to reserve space for them.
        // NOTE: some of this is not written as simply as it might seem because
        // the values are all unsigned and subtracting needs to be done carefully
        // so that an underflow doesn’t cause problems.
        if(persistentNum < MIN_EVICT_OBJECTS)
        {
            UINT32 needed = (MIN_EVICT_OBJECTS - persistentNum)
                            * NvGetEvictObjectSize();
            if(needed > remainByte)
                remainByte = 0;
            else
                remainByte -= needed;
        }
        // if the requisite number of evict objects have been allocated then
        // no need to reserve additional space
        // This checks for the size of the value being added plus the index value.
        // NOTE: This does not check to see if the end marker can be placed in
        // memory because the end marker will not be written if it will not fit.
        return (size + sizeof(UINT32) <= remainByte);
    }
```
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).

```c
static void NvAdd(
    UINT32 totalSize, // IN: total size needed for this entity
    // For evict object, totalSize is the same as bufferSize. For NV Index, totalSize is bufferSize plus index data size
    UINT32 bufferSize, // IN: size of initial buffer
    BYTE *entity // IN: initial buffer
)
{
    UINT32 endAddr;
    UINT32 nextAddr;
    UINT32 listEnd = 0;

    // Get the end of data list
    endAddr = NvGetEnd();

    // Calculate the value of next pointer, which is the size of a pointer + the entity data size
    nextAddr = endAddr + sizeof(UINT32) + totalSize;

    // Write next pointer
    _plat__NvMemoryWrite(endAddr, sizeof(UINT32), &nextAddr);

    // Write entity data
    _plat__NvMemoryWrite(endAddr + sizeof(UINT32), bufferSize, entity);

    // Write the end of list if it is not going to exceed the NV space
    if(nextAddr + sizeof(UINT32) <= s_evictNvEnd)
        _plat__NvMemoryWrite(nextAddr, sizeof(UINT32), &listEnd);

    // Set the flag so that NV changes are committed before the command completes.
    g_updateNV = TRUE;
}
```

8.4.4.9 NvDelete()

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

```c
static void NvDelete(
    UINT32 entityAddr // IN: address of entity to be deleted
)
{
    UINT32 next;
    UINT32 entrySize;
    UINT32 entryAddr = entityAddr - sizeof(UINT32);
    UINT32 listEnd = 0;

    // Get the offset of the next entry.
    _plat__NvMemoryRead(entryAddr, sizeof(UINT32), &next);

    // The size of this entry is the difference between the current entry and the next entry
    entrySize = next - entryAddr;
```
// Move each entry after the current one to fill the freed space.
// Stop when we have reached the end of all the indexes. There are two
// ways to detect the end of the list. The first is to notice that there
// is no room for anything else because we are at the end of NV. The other
// indication is that we find an end marker.

// The loop condition checks for the end of NV.
while(next + sizeof(UINT32) <= s_evictNvEnd) {
    UINT32      size, oldAddr, newAddr;

    // Now check for the end marker
    _plat__NvMemoryRead(next, sizeof(UINT32), &oldAddr);
    if(oldAddr == 0)
        break;

    size = oldAddr - next;

    // Move entry
    _plat__NvMemoryMove(next, next - entrySize, size);

    // Update forward link
    newAddr = oldAddr - entrySize;
    _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &newAddr);
    next = oldAddr;
}

// Mark the end of list
_plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &listEnd);

// Set the flag so that NV changes are committed before the command completes.
g_updateNV = TRUE;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>space available</td>
</tr>
<tr>
<td>FALSE</td>
<td>no enough space</td>
</tr>
</tbody>
</table>

static BOOL NvTestRAMSpace(
    UINT32 size  // IN: size of the data to be added to RAM
) {
    if(s_ramIndexSize + size + sizeof(TPM_HANDLE) + sizeof(UINT32)
       <= RAM_INDEX_SPACE)
        return TRUE;
    else
return FALSE;
}

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.

static UINT32
NvGetRamIndexOffset(
    TPMI_RH_NV_INDEX handle,        // IN: NV handle
)
{
    UINT32 currAddr = 0;
    while (currAddr < s_ramIndexSize)
    {
        TPMI_RH_NV_INDEX currHandle;
        UINT32 currSize;
        currHandle = * (TPM_HANDLE *) &s_ramIndex[currAddr + sizeof(UINT32)];
        // Found a match
        if (currHandle == handle)
        {
            // data buffer follows the handle and size field
            return currAddr + sizeof(TPMI_RH_NV_INDEX) + sizeof(UINT32);
        }
        currSize = * (UINT32 *) &s_ramIndex[currAddr];
        currAddr += sizeof(UINT32) + currSize;
    }
    // We assume the index data is existing in RAM space
    pAssert(FALSE);
    return 0;
}

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.

static void
NvAddRAM(
    TPMI_RH_NV_INDEX handle,        // IN: NV handle
    UINT32 size                    // IN: size of data
)
{
    // Add data space at the end of reserved RAM buffer
    *(UINT32 *) &s_ramIndex[s_ramIndexSize] = size + sizeof(TPMI_RH_NV_INDEX);
    *(TPMI_RH_NV_INDEX *) &s_ramIndex[s_ramIndexSize + sizeof(UINT32)] = handle;
    s_ramIndexSize += sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX) + size;
    pAssert(s_ramIndexSize <= RAM_INDEX_SPACE);
    // Update NV version of s_ramIndexSize
    _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
    // Write reserved RAM space to NV to reflect the newly added NV Index
    _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
    return;
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.

```c
static void NvDeleteRAM(TPMI_RH_NV_INDEX handle) // IN: NV handle
{
    UINT32 nodeOffset;
    UINT32 nextNode;
    UINT32 size;

    nodeOffset = NvGetRAMIndexOffset(handle);

    // Move the pointer back to get the size field of this node
    nodeOffset -= sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX);

    // Get node size
    size = *(UINT32 *)&s_ramIndex[nodeOffset];

    // Get the offset of next node
    nextNode = nodeOffset + sizeof(UINT32) + size;

    // Move data
    MemoryMove(s_ramIndex + nodeOffset, s_ramIndex + nextNode,
               s_ramIndexSize - nextNode);

    // Update RAM size
    s_ramIndexSize -= size + sizeof(UINT32);

    // Update NV version of s_ramIndexSize
    _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);

    // Write reserved RAM space to NV to reflect the newly delete NV Index
    _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);

    return;
}
```

8.4.6 Utility Functions

8.4.6.1 NvInitStatic()

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

```c
static void NvInitStatic(void)
{
    UINT16 i;
    UINT32 reservedAddr;

    s_reservedSize[NV_DISABLE_CLEAR] = sizeof(gp.disableClear);
    s_reservedSize[NV_OWNER_ALG] = sizeof(gp.ownerAlg);
    s_reservedSize[NV_ENDORSEMENT_ALG] = sizeof(gp.endorsementAlg);
    s_reservedSize[NV_OWNER_POLICY] = sizeof(gp.ownerPolicy);
    s_reservedSize[NV_ENDORSEMENT_POLICY] = sizeof(gp.endorsementPolicy);
    s_reservedSize[NV_OWNER_AUTH] = sizeof(gp.ownerAuth);
```
```c
void

s_reservedSize[NV_ENDORSEMENT_AUTH] = sizeof(gp.endorsementAuth);
s_reservedSize[NV_LOCKOUT_AUTH] = sizeof(gp.lockoutAuth);
s_reservedSize[NV_EP_SEED] = sizeof(gp.EPSeed);
s_reservedSize[NV_SP_SEED] = sizeof(gp.SPSeed);
s_reservedSize[NV_PP_SEED] = sizeof(gp.PPSeed);
s_reservedSize[NV_PH_PROOF] = sizeof(gp.phProof);
s_reservedSize[NV_SH_PROOF] = sizeof(gp.shProof);
s_reservedSize[NV_PP_SEED] = sizeof(gp.ppList);
s_reservedSize[NV_TOTAL_RESET_COUNT] = sizeof(gp.totalResetCount);
s_reservedSize[NV_RESET_COUNT] = sizeof(gp.resetCount);
s_reservedSize[NV_PCR_POLICIES] = sizeof(gp.pcrPolicies);
s_reservedSize[NV_PCR_ALLOCATED] = sizeof(gp.pcrAllocated);
s_reservedSize[NV_PP_LIST] = sizeof(gp.ppList);
s_reservedSize[NV_FAILED_TRIES] = sizeof(gp.failedTries);
s_reservedSize[NV_MAX_TRIES] = sizeof(gp.maxTries);
s_reservedSize[NV_RECOVERY_TIME] = sizeof(gp.recoveryTime);
s_reservedSize[NV_LOCKOUT_RECOVERY] = sizeof(gp.lockoutRecovery);
s_reservedSize[NV_LOCKOUT_AUTH_ENABLED] = sizeof(gp.lockOutAuthEnabled);
s_reservedSize[NV_ORDERLY] = sizeof(gp.orderlyState);
s_reservedSize[NV_AUDIT_COMMANDS] = sizeof(gp.auditComands);
s_reservedSize[NV_AUDIT_HASH_ALG] = sizeof(gp.auditHashAlg);
s_reservedSize[NV_AUDIT_COUNTER] = sizeof(gp.auditCounter);
s_reservedSize[NV_ALGORITHM_SET] = sizeof(gp.algorithmSet);
s_reservedSize[NV_FIRMWARE_V1] = sizeof(gp.firmwareV1);
s_reservedSize[NV_FIRMWARE_V2] = sizeof(gp.firmwareV2);
s_reservedSize[NV_CLOCK] = sizeof(go.clock);
s_reservedSize[NV_STATE_CLEAR] = sizeof(gc);
s_reservedSize[NV_STATE_RESET] = sizeof(gr);

// Initialize reserved data address. In this implementation, reserved data
// is stored at the start of NV memory
reservedAddr = 0;
for(i = 0; i < NV_RESERVE_LAST; i++)
{
    s_reservedAddr[i] = reservedAddr;
    reservedAddr += s_reservedSize[i];
}

// Initialize auxiliary variable space for index/evict implementation.
// Auxiliary variables are stored after reserved data area
// RAM index copy starts at the beginning
s_ramIndexSizeAddr = reservedAddr;
s_ramIndexAddr = s_ramIndexSizeAddr + sizeof(UINT32);

// Maximum counter value
s_maxCountAddr = s_ramIndexAddr + RAM_INDEX_SPACE;

// dynamic memory start
s_evictNvStart = s_maxCountAddr + sizeof(UINT64);

// dynamic memory ends that the end of NV memory
s_evictNvEnd = NV_MEMORY_SIZE;
return;
```

### 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.
NvInit(void)
{
    UINT32 nullPointer = 0;
    UINT64 zeroCounter = 0;

    // Initialize static variables
    NvInitStatic();

    // Initialize RAM index space as un-used
    _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &nullPointer);

    // Initialize max counter value to 0
    _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &zeroCounter);

    // Initialize the next offset of the first entry in evict/index list to 0
    _plat__NvMemoryWrite(s_evictNvStart, sizeof(TPM_HANDLE), &nullPointer);
    return;
}

8.4.6.3 NvReadReserved()

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

void NvReadReserved(
    NV_RESERVE type, // IN: type of reserved data
    void *buffer   // OUT: buffer receives the
                     // data.
)
{
    // Input type should be valid
    pAssert(type >= 0 && type < NV_RESERVE_LAST);
    _plat__NvMemoryRead(s_reservedAddr[type], s_reservedSize[type], buffer);
    return;
}

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.

void NvWriteReserved(
    NV_RESERVE type, // IN: type of reserved data
    void *buffer   // IN: data buffer
)
{
    // Input type should be valid
    pAssert(type >= 0 && type < NV_RESERVE_LAST);
    _plat__NvMemoryWrite(s_reservedAddr[type], s_reservedSize[type], buffer);
    g_updateNV = TRUE;
    return;
}
8.4.6.5  NvReadPersistent()

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

```c
void NvReadPersistent(void)
{
    // Hierarchy persistent data
    NvReadReserved(NV_DISABLE_CLEAR, &gp.disableClear);
    NvReadReserved(NV_OWNER_ALG, &gp.ownerAlg);
    NvReadReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
    NvReadReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
    NvReadReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
    NvReadReserved(NV_OWNER_AUTH, &gp.ownerAuth);
    NvReadReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
    NvReadReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
    NvReadReserved(NV_EP_SEED, &gp.EPSeed);
    NvReadReserved(NV_SP_SEED, &gp.SPSeed);
    NvReadReserved(NV_PP_SEED, &gp.PPSeed);
    NvReadReserved(NV_PH_PROOF, &gp.phProof);
    NvReadReserved(NV_SH_PROOF, &gp.shProof);
    NvReadReserved(NV_EH_PROOF, &gp.ehProof);

    // Time persistent data
    NvReadReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
    NvReadReserved(NV_RESET_COUNT, &gp.resetCount);

    // PCR persistent data
    NvReadReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
    NvReadReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);

    // Physical Presence persistent data
    NvReadReserved(NV_PP_LIST, &gp.ppList);

    // Dictionary attack values persistent data
    NvReadReserved(NV_FAILED_TRIES, &gp.failedTries);
    NvReadReserved(NV_MAX_TRIES, &gp.maxTries);
    NvReadReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
    NvReadReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
    NvReadReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);

    // Orderly State persistent data
    NvReadReserved(NV_ORDERLY, &gp.orderlyState);

    // Command audit values persistent data
    NvReadReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
    NvReadReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
    NvReadReserved(NV_AUDIT_COUNTER, &gp.auditCounter);

    // Algorithm selection persistent data
    NvReadReserved(NV_ALGORITHM_SET, &gp.algorithmSet);

    // Firmware version persistent data
    NvReadReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
    NvReadReserved(NV_FIRMWARE_V2, &gp.firmwareV2);

    return;
}
```

8.4.6.6  NvIsPlatformPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the platform.
### 8.4.6.7 NvIsOwnerPersistentHandle()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>handle is owner persistent handle</td>
</tr>
<tr>
<td>FALSE</td>
<td>handle is not owner persistent handle and may not be a persistent handle at all</td>
</tr>
</tbody>
</table>

```c
bool
NvIsOwnerPersistentHandle(
    TPM_HANDLE handle,  // IN: handle
)
{
    return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
}
```

### 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.

```c
static UINT32
NvNextIndex(
    NV_ITER *iter
)
{
    UINT32 addr;
    TPM_HANDLE handle;
    while((addr = NvNext(iter)) != 0)
    {
        // Read handle
        _plat_NvMemoryRead(addr, sizeof(TPM_HANDLE), &handle);
        if(HandleGetType(handle) == TPM_HT_NV_INDEX)
            return addr;
    }
    pAssert(addr == 0);
    return addr;
}
```
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.

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

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.

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

8.4.6.11 NvPowerOn()

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

```c
579 void
580 NvPowerOn(void)
581 {
582     NvInitStatic();
583     return;
584 }
```
8.4.6.12 NvStateSave()

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

```c
void NvStateSave(void)
{
    // Write RAM backed NV Index info to NV
    // No need to save s_ramIndexSize because we save it to NV whenever it is
    // updated.
    _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);

    // Set the flag so that an NV write happens before the command completes.
    g_updateNV = TRUE;

    return;
}
```

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.

```c
void NvEntityStartup(STARTUP_TYPE type) // IN: start up type
{
    NV_ITER             iter = NV_ITER_INIT;

    if(type == SU_RESUME)
        return;

    NV_INDEX    nvIndex;

    while((currentAddr = NvNextIndex(&iter)) != 0)
        {
        nvIndex.publicArea.attributes.TPMA_NV_READLOCKED = CLEAR;
        nvIndex.publicArea.attributes.TPMA_NV_READDEFINE = CLEAR;
        } // Clear read/write lock
```
// Reset NV data for TPMA_NV_CLEAR_STCLEAR
if(nvIndex.publicArea.attributes.TPMA_NV_CLEAR_STCLEAR == SET)
    nvIndex.publicArea.attributes.TPMA_NV_WRITTEN = CLEAR;

// Reset NV data for orderly values that are not counters
// NOTE: The function has already exited on a TPM Resume, so the only
// things being processed are TPM Restart and TPM Reset
if(type == SU_RESET &&
   nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET &&
   nvIndex.publicArea.attributes.TPMA_NV_COUNTER == CLEAR)
    nvIndex.publicArea.attributes.TPMA_NV_WRITTEN = CLEAR;

// Write NV Index info back
_plat__NvMemoryWrite(indexAddr, sizeof(NV_INDEX), &nvIndex);

// Set the flag that a NV write happens
if_updateNV = TRUE;

// Set the lower bits in an orderly counter to 1 for a non-orderly startup
if(g_prevOrderlyState == SHUTDOWN_NONE &&
   nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
{
    if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET &&
       nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET)
    {
        TPMI_RH_NV_INDEX    nvHandle;
        UINT64              counter;

        // Read NV handle
        __plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);

        // Read the counter value saved to NV upon the last roll over.
        // Do not use RAM backed storage for this one.
        nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = CLEAR;
        NvGetIntIndexData(nvHandle, &nvIndex, &counter);
        nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = SET;

        // Set the lower bits of counter to 1
        counter |= MAX_ORDERLY_COUNT;

        // Write back to RAM
        NvWriteIndexData(nvHandle, &nvIndex, 0, 8, &counter);

        // No write to NV because an orderly shutdown will update the
        // counters.
        return;
    }
}

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_DEFINED</td>
<td>the handle points to an existing NV Index</td>
</tr>
<tr>
<td>TPM_RC_HIERARCHY</td>
<td>the handle points to an existing NV Index that is created by a disabled hierarchy</td>
</tr>
</tbody>
</table>

```
TPM_RC
NvIsUndefinedIndex(
    TPMI_RH_NV_INDEX handle    // IN: handle
)
{
    UINT32          entityAddr;       // offset points to the entity
    NV_INDEX        nvIndex;
    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
    entityAddr = NvFindHandle(handle);
    // If handle is not found, return TPM_RC_SUCCESS
    if(entityAddr == 0) return TPM_RC_SUCCESS;
    // Read NV Index info structure
    _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
                         &nvIndex);
    // if SHEnable is disabled, an ownerCreate NV Index should not be
    // indicated as present
    if(gc.shEnable == FALSE &&
        nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
        return TPM_RC_HIERARCHY;
    // if PHEnable is disabled, a platformCreate NV Index should not be
    // indicated as present
    if(g_phEnable == FALSE &&
        nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
        return TPM_RC_HIERARCHY;
    // NV Index is defined
    return TPM_RC_NV_DEFINED;
}
```

8.4.7.3 NvIndexIsAccessible()

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RCHANDLE</td>
<td>the handle points to an undefined NV Index</td>
</tr>
<tr>
<td>TPM_RC_HIERARCHY</td>
<td>the handle points to an existing NV Index that is created by a disabled hierarchy</td>
</tr>
</tbody>
</table>

```
TPM_RC
NvIndexIsAccessible(
    TPMI_RH_NV_INDEX handle    // IN: handle
)
```


```c
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>handle does not reference an existing persistent object</td>
</tr>
<tr>
<td>FALSE</td>
<td>handle does reference an existing persistent object</td>
</tr>
</tbody>
</table>

```c
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_REFERENCE_H0</td>
<td>the handle does not point to an existing persistent object</td>
</tr>
<tr>
<td>TPM_RC_HIERARCHY</td>
<td>the handle points to an existing persistent object belongs to a disabled hierarchy</td>
</tr>
</tbody>
</table>

```c
TPM_RC
NvGetEvictObject(
    TPM_HANDLE handle, // IN: handle
    OBJECT *object  // OUT: object data
)
{
    UINT32 entityAddr; // offset points to the entity
    pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
    // Find the address of evict object
    entityAddr = NvFindHandle(handle);
    // If handle is not found, return TPM_RC_REFERENCE_H0
    if(entityAddr == 0) return TPM_RC_REFERENCE_H0;
    // Read evict object
    _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(OBJECT), object);
    if(HierarchyIsEnabled(ObjectDataGetHierarchy(object)) == FALSE)
        return TPM_RC_HIERARCHY;
    return TPM_RC_SUCCESS;
}
```

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.

```c
void
NvGetIndexInfo(
    TPMI_RH_NV_INDEX handle, // IN: handle
    NV_INDEX *nvIndex // OUT: NV index structure
)
{
    UINT32 entityAddr; // offset points to the entity
    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
    // Find the address of evict object
    entityAddr = NvFindHandle(handle);
    pAssert(entityAddr != 0);
    // This implementation uses the default format so just
    // read the data in
    _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
        nvIndex);
```
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.

```c
UINT64
NvInitialCounter(void)
{
    UINT64          maxCount;
    NV_ITER         iter = NV_ITER_INIT;
    UINT32          currentAddr;

    // Read the maxCount value
    maxCount = NvReadMaxCount();

    // Iterate all existing counters
    while((currentAddr = NvNextIndex(&iter)) != 0)
    {
        TPMI_RH_NV_INDEX    nvHandle;
        NV_INDEX            nvIndex;

        // Read NV handle
        _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);

        // Get NV Index
        NvGetIndexInfo(nvHandle, &nvIndex);
        if(    nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
             & nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
        {
            UINT64      countValue;
            // Read counter value
            NvGetIntIndexData(nvHandle, &nvIndex, &countValue);
            if(countValue > maxCount)
                maxCount = countValue;
        }
    }

    // Initialize the new counter value to be maxCount + 1
    // A counter is only initialized the first time it is written. The
    // way to write a counter is with TPM2_NV_INCREMENT(). Since the
    // "initial" value of a defined counter is the largest count value that
    // may have existed in this index previously, then the first use would
    // add one to that value.
    return maxCount;
}
```

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.

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

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     // bit and counter value is store in native format for TPM CPU. So we directly
912     // copy the contents of NV to output data buffer
913     if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
914     {
915         UINT32      ramAddr;
918  // Get data from RAM buffer
919  ramAddr = NvGetRAMIndexOffset(handle);
920  MemoryCopy(data, s_ramIndex + ramAddr, sizeof(*data));
921 }
922 else
923 {
924   UINT32 entityAddr;
925   entityAddr = NvFindHandle(handle);
926
927   // Get data from NV
928   // Skip NV Index info, read data buffer
929   _plat__NvMemoryRead(
930     entityAddr + sizeof(TPM_HANDLE) + sizeof(NV_INDEX),
931     sizeof(UINT64), data);
932 }
933 return;
934 }
935
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting so retry</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting so retry</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available</td>
</tr>
</tbody>
</table>

```c
TPM_RC
NvWriteIndexData(
    TPMI_RH_NV_INDEX handle,  // IN: handle
    NV_INDEX *nvIndex,       // IN: RAM copy of NV Index
    UINT32 offset,          // IN: offset of NV data
    UINT32 size,            // IN: size of NV data
    void *data              // OUT: data buffer
  ) {
    TPM_RC result;
    pAssert(nvIndex->publicArea.dataSize >= offset + size);
    if(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
      nvIndex->publicArea.attributes.TPMA_NV_WRITTEN = SET;
    result = NvWriteIndexInfo(handle, nvIndex);
    if(result != TPM_RC_SUCCESS)
      return result;

    // Check to see if process for an orderly index is required.
    if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
      UINT32 ramAddr;
      ramAddr = NvGetRAMIndexOffset(handle);
      MemoryCopy(s_ramIndex + ramAddr + offset, data, size);
      if(g_updateNV != FALSE)
        g_clearOrderly = TRUE;
    }
    // Need to process this part if the Index isn't orderly or if it is
    // an orderly counter that just rolled over.
    if(g_updateNV || nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == CLEAR)
      {
        // Processing for an index with TPMA_NV_ORDERLY CLEAR
        UINT32 entryAddr = NvFindHandle(handle);
        pAssert(entryAddr != 0);
        // Offset into the index to the first byte of the data to be written
        entryAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
        // If the data is actually changed, then a write to NV is required
        if(_plat__NvIsDifferent(entryAddr, size, data))
```
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.

```
UINT16
NvGetName(
  TPMI_RH_NV_INDEX    handle,        // IN: handle of the index
  BYTE                *name          // OUT: name of the index

) {
  UINT16               dataSize, digestSize;
  NV_INDEX             nvIndex;
  BYTE                 marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
  BYTE                *buffer;
  HASH_STATE           hashState;

  // Get NV public info
  NvGetIndexInfo(handle, &nvIndex);

  // Marshal public area
  buffer = marshalBuffer;
  dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex.publicArea, &buffer, NULL);

  // hash public area
  digestSize = CryptStartHash(nvIndex.publicArea.nameAlg, &hashState);
  CryptUpdateDigest(&hashState, dataSize, marshalBuffer);

  // Complete digest leaving room for the nameAlg
  CryptCompleteHash(&hashState, digestSize, &name[2]);

  // Include the nameAlg
  UINT16_TO_BYTE_ARRAY(nvIndex.publicArea.nameAlg, name);
  return digestSize + 2;
}
```

8.4.7.13 NvDefineIndex()

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_SPACE</td>
<td>insufficient NV space</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_HANDLE</td>
<td>the requested handle is already in use</td>
</tr>
<tr>
<td>TPM_RC_NV_SPACE</td>
<td>insufficient NV space</td>
</tr>
</tbody>
</table>

1105 TPM_RC
1106 NvAddEvictObject(
1107     TPMI_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)];
OBJECT *nvObject;  // a pointer to the OBJECT data in
UINT16 entrySize;  // size of entry

// evict handle type should match the object hierarchy
pAssert(  (  NvIsPlatformPersistentHandle(evictHandle)
&& object->attributes.ppsHierarchy == SET)
|| (  NvIsOwnerPersistentHandle(evictHandle)
&& (  object->attributes.spsHierarchy == SET
|| object->attributes.epsHierarchy == SET)));

// An evict needs 4 bytes of handle + sizeof OBJECT
entrySize = sizeof(TPM_HANDLE) + sizeof(OBJECT);

// Check if we have enough space to add the evict object
// An evict object needs 8 bytes in index table + sizeof OBJECT
// In this implementation, the only resource limitation is the available NV
// space. Other implementation may have other limitation on evict object
// handle space
if(!NvTestSpace(entrySize, FALSE)) return TPM_RC_NV_SPACE;

// Allocate a new evict handle
if(!NvIsUndefinedEvictHandle(evictHandle))
return TPM_RC_NV_DEFINED;

// Copy evict object to nvBuffer
// Copy handle
* (TPM_HANDLE *) nvBuffer = evictHandle;

// Copy OBJECT
nvObject = (OBJECT *) (nvBuffer + sizeof(TPM_HANDLE));
*nvObject = *object;

// Set evict attribute and handle
nvObject->attributes.evict = SET;
vObject->evictHandle = evictHandle;

// Add evict to NV memory
NvAdd(entrySize, entrySize, nvBuffer);
return TPM_RC_SUCCESS;
}

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.

void NvDeleteEntity(  
TPM_HANDLE handle  // IN: handle of entity to be deleted
)
{
  UINT32 entityAddr;  // pointer to entity
  entityAddr = NvFindHandle(handle);
pAssert(entityAddr != 0);
  if(HandleGetType(handle) == TPM_HT_NV_INDEX)
  {
    NV_INDEX nvIndex;
    // Read the NV Index info
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.

```c
void NvFlushHierarchy(TPMI_RH_HIERARCHY hierarchy) // IN: hierarchy to be flushed.
{
    NV_ITER iter = NV_ITER_INIT;
    UINT32 currentAddr;
    while((currentAddr = NvNext(&iter)) != 0)
    {
        TPM_HANDLE entityHandle;
        _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
        if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
        {
            // Handle NV Index
            NV_INDEX nvIndex;

            // If flush endorsement or platform hierarchy, no NV Index would be flushed
            if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
                continue;
            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX), &nvIndex);

            // For storage hierarchy, flush OwnerCreated index
            if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
                // Delete the NV Index
                NvDelete(entityAddr);
        }
    }
}
```
iter = NV_ITER_INIT;

// If the NV Index is RAM back, delete the RAM data as well
if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
    NvDeleteRAM(entityHandle);

else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
{
    OBJECT object;

    // Get evict object
    NvGetEvictObject(entityHandle, &object);

    // If the evict object belongs to the hierarchy to be flushed
    if(( hierarchy == TPM_RH_PLATFORM
        && object.attributes.ppsHierarchy == SET)
        || ( hierarchy == TPM_RH_OWNER
        && object.attributes.spsHierarchy == SET)
        || ( hierarchy == TPM_RH_ENDORSEMENT
        && object.attributes.epsHierarchy == SET))
    {
        // Delete the evict object
        NvDelete(currentAddr);

        // Re-iterate from beginning after a delete
        iter = NV_ITER_INIT;
    }
}
else
{
    pAssert(FALSE);
}

return;

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().

void NvSetGlobalLock(void)
{
    NV_ITER iter = NV_ITER_INIT;
    UINT32 currentAddr;

    // Check all indices
    while((currentAddr = NvNextIndex(&iter)) != 0)
    {
        NV_INDEX nvIndex;

        // Read the index data
        _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
            sizeof(NV_INDEX), &nvIndex);

        // See if it should be locked
        if(nvIndex.publicArea.attributes.TPMA_NV_GLOBALLOCK == SET)
        {
            // if so, lock it
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.

```c
static void InsertSort(
    TPML_HANDLE         *handleList,  // IN/OUT: sorted handle list
    UINT32              count,      // IN: maximum count in the handle list
    TPM_HANDLE          entityHandle // IN: handle to be inserted
)
{
    UINT32          i, j;
    UINT32          originalCount;

    // For a corner case that the maximum count is 0, do nothing
    if(count == 0) return;

    // For empty list, add the handle at the beginning and return
    if(handleList->count == 0)
    {
        handleList->handle[0] = entityHandle;
        handleList->count++;
        return;
    }

    // Check if the maximum of the list has been reached
    originalCount = handleList->count;
    if(originalCount < count)
    {
        handleList->count++;
    }

    // Insert the handle to the list
    for(i = 0; i < originalCount; i++)
    {
        if(handleList->handle[i] > entityHandle)
        {
            for(j = handleList->count - 1; j > i; j--)
            {
                handleList->handle[j] = handleList->handle[j-1];
            }
            break;
        }
    }

    // If a slot was found, insert the handle in this position
    if(i < originalCount || handleList->count > originalCount)
    {
        handleList->handle[i] = entityHandle;
    }
    return;
}
```
8.4.7.19 NvCapGetPersistent()

This function is used to get a list of handles of the persistent objects, starting at handle. Handle must be in valid persistent object handle range, but does not have to reference an existing persistent object.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO NvCapGetPersistent(
    TPMU_DH_OBJECT handle, // IN: start handle
    UINT32 count, // IN: maximum number of returned handles
    TPML_HANDLE *handleList // OUT: list of handle
)
{
    TPMI_YES_NO more = NO;
    NV_ITER iter = NV_ITER_INIT;
    UINT32 currentAddr;

    _pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if (count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    while((currentAddr = NvNextEvict(&iter)) != 0)
    {
        TPM_HANDLE entityHandle;

        // Read handle information.
        _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);

        // Ignore persistent handles that have values less than the input handle
        if(entityHandle < handle) continue;

        // if the handles in the list have reached the requested count, and there
        // are still handles need to be inserted, indicate that there are more.
        if(handleList->count == count)
        {
            more = YES;
        }

        // A handle with a value larger than start handle is a candidate
        // for return. Insert sort it to the return list. Insert sort algorithm
        // is chosen here for simplicity based on the assumption that the total
        // number of NV Indices is small. For an implementation that may allow
        // large number of NV Indices, a more efficient sorting algorithm may be
        // used here.
        InsertSort(handleList, count, entityHandle);
    }

    return more;
}
```

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.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles to report</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been reported</td>
</tr>
</tbody>
</table>

```c
TPM_IYES_NO

NvCapGetIndex(
    TPM_DH_OBJECT handle, // IN: start handle
    UINT32            count, // IN: maximum number of returned handles
    TPML_HANDLE      *handleList // OUT: list of handle
)
{
    TPMI_YES_NO more = NO;
    NV_ITER        iter = NV_ITER_INIT;
    UINT32         currentAddr;

    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if (count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    while ((currentAddr = NvNextIndex(&iter)) != 0)
    {
        TPM_HANDLE entityHandle;
        _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);

        // Ignore index handles that have values less than the 'handle'
        if (entityHandle < handle)
            continue;

        // if the count of handles in the list has reached the requested count,
        // and there are still handles to report, set more.
        if (handleList->count == count)
            more = YES;

        // A handle with a value larger than start handle is a candidate
        // for return. Insert sort it to the return list. Insert sort algorithm
        // is chosen here for simplicity based on the assumption that the total
        // number of NV Indices is small. For an implementation that may allow
        // large number of NV Indices, a more efficient sorting algorithm may be
        // used here.
        InsertSort(handleList, count, entityHandle);
    }

    return more;
}
```

8.4.7.21 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```c
UINT32
NvCapGetIndexNumber(void)
{
    UINT32 num = 0;
    NV_ITER iter = NV_ITER_INIT;

    while (NvNextIndex(&iter) != 0) num++;
```
8.4.7.22 NvCapGetPersistentNumber()

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

```c
UINT32 NvCapGetPersistentNumber(void)
{
    UINT32 num = 0;
    NV_ITER iter = NV_ITER_INIT;
    while(NvNextEvict(&iter) != 0) num++;
    return num;
}
```

8.4.7.23 NvCapGetPersistentAvail()

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

```c
UINT32 NvCapGetPersistentAvail(void)
{
    UINT32 availSpace;
    UINT32 objectSpace;
    availSpace = NvGetFreeByte();
    objectSpace = NvGetEvictObjectSize();
    return availSpace / objectSpace;
}
```

8.4.7.24 NvCapGetCounterNumber()

Get the number of defined NV Indexes that have NV TPMA_NV_COUNTER attribute SET.

```c
UINT32 NvCapGetCounterNumber(void)
{
    NV_ITER iter = NV_ITER_INIT;
    UINT32 currentAddr;
    UINT32 num = 0;
    while((currentAddr = NvNextIndex(&iter)) != 0)
    {
        NV_INDEX nvIndex;
        _plat_NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
                           sizeof(NV_INDEX), &nvIndex);
        if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET) num++;
    }
    return num;
}
```
8.4.7.25  NvCapGetCounterAvail()

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

```c
UINT32
NvCapGetCounterAvail(void)
{
    UINT32    availNVSpace;
    UINT32    availRAMSpace;
    UINT32    counterNVSpace;
    UINT32    counterRAMSpace;
    UINT32    persistentNum = NvCapGetPersistentNumber();

    // Get the available space in NV storage
    availNVSpace = NvGetFreeByte();

    if (persistentNum < MIN_EVICT_OBJECTS)
    {
        // Some space have to be reserved for evict object. Adjust availNVSpace.
        UINT32    reserved = (MIN_EVICT_OBJECTS - persistentNum) * NvGetEvictObjectSize();
        if (reserved > availNVSpace)
            availNVSpace = 0;
        else
            availNVSpace -= reserved;
    }

    // Get the space needed to add a counter index to NV storage
    counterNVSpace = NvGetCounterSize();

    // Compute the available space in RAM
    availRAMSpace = RAM_INDEX_SPACE - s_ramIndexSize;

    // Compute the space needed to add a counter index to RAM storage
    // It takes an size field, a handle and sizeof(UINT64) for counter data
    counterRAMSpace = sizeof(UINT32) + sizeof(TPM_HANDLE) + sizeof(UINT64);

    // Return the min of counter number in NV and in RAM
    if(availNVSpace / counterNVSpace > availRAMSpace / counterRAMSpace)
        return availRAMSpace / counterRAMSpace;
    else
        return availNVSpace / counterNVSpace;
}
```

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

```c
#define OBJECT_C
#include "InternalRoutines.h"
#include <Platform.h>
```
8.5.3 Functions

8.5.3.1 ObjectStartup()

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

```c
void ObjectStartup(void)
{
    UINT32 i;

    // object slots initialization
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        // Set the slot to not occupied
        s_objects[i].occupied = FALSE;
    }

    return;
}
```

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().

```c
void ObjectCleanupEvict(void)
{
    UINT32 i;

    // This has to be iterated because a command may have two handles
    // and they may both be persistent.
    // This could be made to be more efficient so that a search is not needed.
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        // If an object is a temporary evict object, flush it from slot
        if(s_objects[i].object.entity.attributes.evict == SET)
            s_objects[i].occupied = FALSE;
    }

    return;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if the handle references a loaded object</td>
</tr>
<tr>
<td>FALSE</td>
<td>if the handle is not an object handle, or it does not reference to a loaded object</td>
</tr>
</tbody>
</table>

```c
BOOL ObjectIsPresent(
    TPMI_DH_OBJECT  handle     // IN: handle to be checked
)
{
    UINT32 slotIndex;        // index of object slot
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>object is an HMAC, hash, or event sequence object</td>
</tr>
<tr>
<td>FALSE</td>
<td>object is not an HMAC, hash, or event sequence object</td>
</tr>
</tbody>
</table>

```c
BOOL ObjectIsSequence(
    OBJECT *object,  // IN: handle to be checked
)
{
    pAssert (object != NULL);
    if (    object->attributes.hmacSeq == SET
            || object->attributes.hashSeq == SET
            || object->attributes.eventSeq == SET)
        return TRUE;
    else
        return FALSE;
}
```

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.

```c
OBJECT * ObjectGet(
    TPMI_DH_OBJECT handle,  // IN: handle of the object
)
{
    pAssert( handle >= TRANSIENT_FIRST
                && handle - TRANSIENT_FIRST < MAX_LOADED_OBJECTS);
    pAssert(s_objects[handle - TRANSIENT_FIRST].occupied == TRUE);
    // In this implementation, the handle is determined by the slot occupied by the
    // object.
    return &s_objects[handle - TRANSIENT_FIRST].object.entity;
}
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.

```c
UINT16 ObjectGetName(
    TPMI_DH_OBJECT handle,  // IN: handle of the object
    BYTE *name           // OUT: name of the object
) {
    OBJECT *object = ObjectGet(handle);
    if (object->publicArea.nameAlg == TPM_ALG_NULL)
        return 0;
    // Copy the Name data to the output
    MemoryCopy(name, object->name.t.name, object->name.t.size);
    return object->name.t.size;
}
```

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.

```c
TPMI_ALG_HASH ObjectGetNameAlg(
    TPMI_DH_OBJECT handle  // IN: handle of the object
) {
    OBJECT *object = ObjectGet(handle);
    return object->publicArea.nameAlg;
}
```

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.

```c
void ObjectGetQualifiedName(
    TPMI_DH_OBJECT handle,  // IN: handle of the object
    TPM2B_NAME *qualifiedName // OUT: qualified name of the object
) {
    OBJECT *object = ObjectGet(handle);
    if (object->publicArea.nameAlg == TPM_ALG_NULL)
        qualifiedName->t.size = 0;
    else
        // Copy the name
        *qualifiedName = object->qualifiedName;
    return;
}
```
8.5.3.9 ObjectDataGetHierarchy()

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

```c
TPMI_RH_HIERARCHY
ObjectDataGetHierarchy(
    OBJECT *object   // IN : object
)
{
    if(object->attributes.spsHierarchy)
    {
        return TPM_RH_OWNER;
    }
    else if(object->attributes.epsHierarchy)
    {
        return TPM_RH_ENDORSEMENT;
    }
    else if(object->attributes.ppsHierarchy)
    {
        return TPM_RH_PLATFORM;
    }
    else
    {
        return TPM_RH_NULL;
    }
}
```

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.

```c
TPMI_RH_HIERARCHY
ObjectGetHierarchy(
    TPMI_DH_OBJECT       handle       // IN : object handle
)
{
    OBJECT          *object = ObjectGet(handle);
    return ObjectDataGetHierarchy(object);
}
```

8.5.3.11 ObjectAllocateSlot()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>allocate success</td>
</tr>
<tr>
<td>FALSE</td>
<td>do not have free slot</td>
</tr>
</tbody>
</table>

```c
static BOOL
ObjectAllocateSlot(
    TPMI_DH_OBJECT   *handle,       // OUT: handle of allocated object
)
```
OBJECT **object // OUT: points to the allocated object
{
  UINT32 i;
  // find an unoccupied handle slot
  for(i = 0; i < MAX_LOADED_OBJECTS; i++)
  {
    if(!s_objects[i].occupied) // If found a free slot
    {
      // Mark the slot as occupied
      s_objects[i].occupied = TRUE;
      break;
    }
  }
  // If we reach the end of object slot without finding a free one, return
  // error.
  if(i == MAX_LOADED_OBJECTS) return FALSE;

  *handle = i + TRANSIENT_FIRST;
  *object = &s_objects[i].object.entity;

  // Initialize the object attributes
  MemorySet((&(*object)->attributes), 0, sizeof(OBJECT_ATTRIBUTES));
  return TRUE;
}

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_BINDING</td>
<td>if the public and sensitive parts of the object are not matched</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>if the parameters in the public area of the object are not consistent</td>
</tr>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>if there is no free slot for an object</td>
</tr>
<tr>
<td>TPM_RC_TYPE</td>
<td>the public and private parts are not the same type</td>
</tr>
</tbody>
</table>

TPM_RC

ObjectLoad(
  TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy to which the object
  TPMT_PUBLIC *publicArea,      // IN: public area
  TPMT_SENSITIVE *sensitive,   // IN: sensitive area (may be null)
  TPM2B_NAME *name,            // IN: object's name (may be null)
  TPM_HANDLE parentHandle,     // IN: handle of parent
  BOOL skipChecks,             // IN: flag to indicate if it is OK to
  TPMI_DH_OBJECT *handle       // OUT: object handle
)
{
  OBJECT *object = NULL;
  OBJECT *parent = NULL;
  TPM_RC result = TPM_RC_SUCCESS;
  TPM2B_NAME parentQN; // Parent qualified name

  // Try to allocate a slot for new object
  if(!ObjectAllocateSlot(handle, &object))
  {
    return TPM_RC_OBJECT_MEMORY;
  }
// Initialize public
object->publicArea = *publicArea;

if (sensitive != NULL)
    object->sensitive = *sensitive;

// Are the consistency checks needed
if (!skipChecks)
{
    // Check if key size matches
    if (!CryptObjectIsPublicConsistent(&object->publicArea))
    {
        result = TPM_RC_KEY;
        goto ErrorExit;
    }

    if (sensitive != NULL)
    {
        // Check if public type matches sensitive type
        result = CryptObjectPublicPrivateMatch(object);
        if (result != TPM_RC_SUCCESS)
            goto ErrorExit;
    }

    object->attributes.publicOnly = (sensitive == NULL);
}

// If 'name' is NULL, then there is nothing left to do for this
// object as it has no qualified name and it is not a member of any
// hierarchy and it is temporary
if (name == NULL || name->t.size == 0)
{
    object->qualifiedName.t.size = 0;
    object->name.t.size = 0;
    object->attributes.temporary = SET;
    return TPM_RC_SUCCESS;
}

// If parent handle is a permanent handle, it is a primary or temporary
// object
if (HandleGetType(parentHandle) == TPM_HT_PERMANENT)
{
    // initialize QN
    parentQN.t.size = 4;

    // for a primary key, parent qualified name is the handle of hierarchy
    UINT32_TO_BYTE_ARRAY(parentHandle, parentQN.t.name);
}
else
{
    // Get hierarchy and qualified name of parent
    ObjectGetQualifiedName(parentHandle, &parentQN);

    // Check for stClear object
    parent = ObjectGet(parentHandle);
    if (publicArea->objectAttributes.stClear == SET
        || parent->attributes.stClear == SET)
        object->attributes.stClear = SET;

    object->name = *name;

    // Compute object qualified name
    ObjectComputeQualifiedName(&parentQN, publicArea->nameAlg,
        name, &object->qualifiedName);

    // Any object in TPM_RH_NULL hierarchy is temporary
    if (hierarchy == TPM_RH_NULL)
    {
        object->attributes.temporary = SET;
    }

else if (parentQN.t.size == sizeof(TPM_HANDLE))
{
    // Otherwise, if the size of parent's qualified name is the size of a
    // handle, this object is a primary object
    object->attributes.primary = SET;
}

switch(hierarchy)
{
    case TPM_RH_PLATFORM:
        object->attributes.ppsHierarchy = SET;
        break;
    case TPM_RH_OWNER:
        object->attributes.spsHierarchy = SET;
        break;
    case TPM_RH_ENDORSEMENT:
        object->attributes.epsHierarchy = SET;
        break;
    case TPM_RH_NULL:
        break;
    default:
        pAssert(FALSE);
        break;
}
return TPM_RC_SUCCESS;

ErrorExit:
ObjectFlush(*handle);
return result;

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.

static BOOL
AllocateSequenceSlot(
    TPM_HANDLE *newHandle,          // OUT: receives the allocated handle
    HASH_OBJECT **object,            // OUT: receives pointer to allocated
    // object
    TPM2B_AUTH *auth                // IN: the authValue for the slot
)
{
    OBJECT *objectHash;              // the hash as an object

    if (!ObjectAllocateSlot(newHandle, &objectHash))
        return FALSE;

    *object = (HASH_OBJECT *)objectHash;

    // Validate that the proper location of the hash state data relative to the
    // object state data.
    pAssert(&(*object)->auth == &objectHash->publicArea.authPolicy);

    // Set the common values that a sequence object shares with an ordinary object
    // The type is TPM_ALG_NULL
    (*object)->type = TPM_ALG_NULL;

    // This has no name algorithm and the name is the Empty Buffer
    (*object)->nameAlg = TPM_ALG_NULL;

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

8.5.3.14 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>if there is no free slot for an object</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>if there is no free slot for an object</td>
</tr>
</tbody>
</table>

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

This function creates an event sequence object.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>if there is no free slot for an object</td>
</tr>
</tbody>
</table>

```c
TPMI_DH_OBJECT *newHandle // OUT: sequence object handle
}
HASH_OBJECT *hashObject;
}
// Try to allocate a slot for new object
if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
    return TPM_RC_OBJECT_MEMORY;
// Set hash sequence bit
hashObject->attributes.hashSeq = SET;
// Start hash for hash sequence
CryptStartHashSequence(hashAlg, &hashObject->state.hashState[0]);
return TPM_RC_SUCCESS;
```

8.5.3.17 ObjectTerminateEvent()

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

```c
void ObjectTerminateEvent(void)
```
```c
{  HASH_OBJECT      *hashObject;
  int              count;
  BYTE             buffer[MAX_DIGEST_SIZE];
  hashObject = (HASH_OBJECT *)ObjectGet(g_DRTMHandle);
  
  // Don't assume that this is a proper sequence object
  if(hashObject->attributes.eventSeq) {
    // If it is, close any open hash contexts. This is done in case
    // the crypto implementation has some context values that need to be
    // cleaned up (hygiene).
    //
    for(count = 0; CryptGetHashAlgByIndex(count) != TPM_ALG_NULL; count++)
      { CryptCompleteHash(&hashObject->state.hashState[count], 0, buffer); }
    // Flush sequence object
    ObjectFlush(g_DRTMHandle);
  }

  g_DRTMHandle = TPM_RH_UNASSIGNED;
}

8.5.3.18 ObjectContextLoad()

This function loads an object from a saved object context.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>if there is no free slot for an object</td>
</tr>
</tbody>
</table>

```}

```c
TPM_RC
ObjectContextLoad(
    OBJECT              *object, // IN: object structure from saved
    TPMI_DH_OBJECT      *handle // OUT: object handle
)
{
    OBJECT      *newObject;
    // Try to allocate a slot for new object
    if(!ObjectAllocateSlot(handle, &newObject))
      return TPM_RC_OBJECT_MEMORY;
    // Copy input object data to internal structure
    *newObject = *object;
    return TPM_RC_SUCCESS;
}

8.5.3.19 ObjectFlush()

This function frees an object slot.
This function requires that the object is loaded.

```
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            switch(hierarchy)
487            {
488                case TPM_RH_PLATFORM:
489                    if(s_objects[i].object.entity.attributes.ppsHierarchy == SET)
490                        s_objects[i].occupied = FALSE;
491                    break;
492                case TPM_RH_OWNER:
493                    if(s_objects[i].object.entity.attributes.spsHierarchy == SET)
494                        s_objects[i].occupied = FALSE;
495                    break;
496                case TPM_RH_ENDORSEMENT:
497                    if(s_objects[i].object.entity.attributes.epsHierarchy == SET)
498                        s_objects[i].occupied = FALSE;
499                    break;
500                default:
501                    pAssert(FALSE);
502                    break;
503            }
504        }
505    }
506    return;
507 }

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

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_REFERENCE_H0</td>
<td>The persistent object does not exist</td>
</tr>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>No object slot</td>
</tr>
<tr>
<td>TPM_RC_HIERARCHY</td>
<td>The handle points to an existing persistent object belongs to a disabled hierarchy</td>
</tr>
</tbody>
</table>

```c
TPM_RC

ObjectLoadEvict(
  TPM_HANDLE *handle   // IN:OUT: evict object handle. If
                       // success, it will be replace by
                       // the loaded object handle
)
{
  TPM_RC          result;
  TPM_HANDLE      evictHandle = *handle; // Save the evict handle
  OBJECT          *object;

  // Try to allocate a slot for new object
  if(!ObjectAllocateSlot(handle, &object))
    return TPM_RC_OBJECT_MEMORY;

  // Copy persistent object to transient object slot. A TPM_RC_REFERENCE_H0
  // or TPM_RC_HIERARCHY error may be returned at this point
  result = NvGetEvictObject(evictHandle, object);

  // Free object slot if fails.
  if(result != TPM_RC_SUCCESS)
    ObjectFlush(*handle);

  return result;
}
```

### 8.5.3.22 ObjectComputeName()

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

```c
void
ObjectComputeName(
  TPM_PUBLIC         *publicArea,   // IN: public area of an object
  TPM2B_NAME          *name        // OUT: name of the object
)
{
  TPM2B_PUBLIC       marshalBuffer;
  BYTE                *buffer;   // auxiliary marshal buffer pointer
  HASH_STATE          hashState; // hash state

  // if the nameAlg is NULL then there is no name.
  if(publicArea->nameAlg == TPM_ALG_NULL)
    { name->t.size = 0;
      return;
    }

  // Start hash stack
  name->t.size = CryptStartHash(publicArea->nameAlg, &hashState);

  // Marshal the public area into its canonical form
  buffer = marshalBuffer.b.buffer;

  marshalBuffer.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);

  // 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.

void ObjectComputeQualifiedName(
  TPM2B_NAME *parentQN,  // IN: parent's qualified name
  TPM_ALG_ID nameAlg,    // IN: name hash
  TPM2B_NAME *name,      // IN: name of the object
  TPM2B_NAME *qualifiedName // OUT: qualified name of the object
) {
  HASH_STATE hashState;  // hash state
  // QN_A = hash_A (QN of parent || NAME_A)
  qualifiedName->t.size = CryptStartHash(nameAlg, &hashState);
  // Add parent's qualified name
  CryptUpdateDigest2B(&hashState, &parentQN->b);
  // Add self name
  CryptUpdateDigest2B(&hashState, &name->b);
  // Complete hash leaving room for the name algorithm
  CryptCompleteHash(&hashState, qualifiedName->t.size,
                    &qualifiedName->t.name[2]);
  UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
  qualifiedName->t.size += 2;
  return;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if the object is a storage key</td>
</tr>
<tr>
<td>FALSE</td>
<td>if the object is not a storage key</td>
</tr>
</tbody>
</table>

BOOL ObjectDataIsStorage(
  TPM_FUNCTION *publicArea       // IN: public area of the object
) {
  if( CryptIsAsymAlgorithm(publicArea->type) // must be asymmetric,
      && publicArea->objectAttributes.restricted == SET // restricted,
      && publicArea->objectAttributes.decrypt == SET // decryption key
  )
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if the object is a storage key</td>
</tr>
<tr>
<td>FALSE</td>
<td>if the object is not a storage key</td>
</tr>
</tbody>
</table>

```c
BOOL ObjectIsStorage(
    TPMI_DH_OBJECT handle       // IN: object handle
) {
    OBJECT *object = ObjectGet(handle);
    return ObjectDataIsStorage(&object->publicArea);
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO ObjectCapGetLoaded(
    TPMI_DH_OBJECT handle,       // IN: start handle
    UINT32 count,                // IN: count of returned handles
    TPML_HANDLE *handleList      // OUT: list of handle
) {
    TPMI_YES_NO more = NO;
    UINT32 i;
    pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    // Iterate object slots to get loaded object handles
    for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
        { if(s_objects[i].occupied == TRUE)
            { // A valid transient object can not be the copy of a persistent object
                pAssert(s_objects[i].object.entity.attributes.evict == CLEAR);
```

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

613
614
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if the object is a storage key</td>
</tr>
<tr>
<td>FALSE</td>
<td>if the object is not a storage key</td>
</tr>
</tbody>
</table>

```c
BOOL ObjectIsStorage(
    TPMI_DH_OBJECT handle       // IN: object handle
) {
    OBJECT *object = ObjectGet(handle);
    return ObjectDataIsStorage(&object->publicArea);
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO ObjectCapGetLoaded(
    TPMI_DH_OBJECT handle,       // IN: start handle
    UINT32 count,                // IN: count of returned handles
    TPML_HANDLE *handleList      // OUT: list of handle
) {
    TPMI_YES_NO more = NO;
    UINT32 i;
    pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    // Iterate object slots to get loaded object handles
    for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
        { if(s_objects[i].occupied == TRUE)
            { // A valid transient object can not be the copy of a persistent object
                pAssert(s_objects[i].object.entity.attributes.evict == CLEAR);
```
if(handleList->count < count)
{
    // If we have not filled up the return list, add this object
    // handle to it
    handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
    handleList->count++;
}
else
{
    // If the return list is full but we still have loaded object
    // available, report this and stop iterating
    more = YES;
    break;
}
}
}

8.5.3.27 ObjectCapGetTransientAvail()

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

UINT32

ObjectCapGetTransientAvail(void)
{
    UINT32      i;
    UINT32      num = 0;
    // Iterate object slot to get the number of unoccupied slots
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        if(s_objects[i].occupied == FALSE) num++;
    }
    return num;
}

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

#define PCR_C
#include "InternalRoutines.h"
#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.

```c
static const PCR_Attributes s_initAttributes[] =
{
    // PCR 0 - 15, static RTM
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {0, 0x0F, 0x1F},    // PCR 16, Debug
    {0, 0x10, 0x1C},    // PCR 17, Locality 4
    {0, 0x10, 0x1C},    // PCR 18, Locality 3
    {0, 0x10, 0x0C},    // PCR 19, Locality 2
    {0, 0x14, 0x0E},    // PCR 20, Locality 1
    {0, 0x14, 0x04},    // PCR 21, Dynamic OS
    {0, 0x14, 0x04},    // PCR 22, Dynamic OS
    {0, 0x0F, 0x1F},    // PCR 23, App specific
    {0, 0x0F, 0x1F}     // PCR 24, testing policy
};
```

### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE:</td>
<td>PCR belongs an auth group</td>
</tr>
<tr>
<td>FALSE:</td>
<td>PCR does not belong an auth group</td>
</tr>
</tbody>
</table>

```c
BOOL PCRBelongsAuthGroup(
    TPMI_DH_PCR handle,       // IN: handle of PCR
    UINT32        *groupIndex  // OUT: group index if PCR belongs a
)
{
    #if NUM_AUTHVALUE_PCR_GROUP > 0
        // Platform specification determines to which auth group a PCR belongs (if
        // any). In this implementation, we assume there is only
        // one auth group which contains PCR[20-22]. If the platform specification
        // requires differently, the implementation should be changed accordingly
        if(handle >= 20 && handle <= 22)
            *groupIndex = 0;
            return TRUE;
    }
    
    #endif
    return FALSE;
}
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE:</td>
<td>PCR belongs a policy group</td>
</tr>
<tr>
<td>FALSE:</td>
<td>PCR does not belong a policy group</td>
</tr>
</tbody>
</table>

```c
BOOL PCRBelongsPolicyGroup(
    TPMI_DH_PCR handle, // IN: handle of PCR
    UINT32 *groupIndex   // OUT: group index if PCR belongs a
                         //      group that allows policy. If PCR
                         //      does not belong to a policy group,
                         //      the value in this parameter is
                         //      invalid
){
    #if NUM_POLICY_PCR_GROUP > 0
    // Platform specification decides if a PCR belongs to a policy group and
    // belongs to which group. In this implementation, we assume there is only
    // one policy group which contains PCR20-22. If the platform specification
    // requires differently, the implementation should be changed accordingly
    if(handle >= 20 && handle <= 22)
        *groupIndex = 0;
        return TRUE;
    #endif
    return FALSE;
}
```

8.6.3.3 **PCRBelongsTCBGroup()**

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE:</td>
<td>PCR belongs to TCB group</td>
</tr>
<tr>
<td>FALSE:</td>
<td>PCR does not belong to TCB group</td>
</tr>
</tbody>
</table>

```c
static BOOL PCRBelongsTCBGroup(
    TPMI_DH_PCR handle       // IN: handle of PCR
){
    #if ENABLE_PCR_NO_INCREMENT == YES
    // Platform specification decides if a PCR belongs to a TCB group. In this
    // implementation, we assume PCR[20-22] belong to TCB group. If the platform
    // specification requires differently, the implementation should be
    // changed accordingly
    if(handle >= 20 && handle <= 22)
        return TRUE;
    #endif
    return FALSE;
}
```
8.6.3.4 **PCRPolicyIsAvailable()**

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>the PCR should be authorized by policy</td>
</tr>
<tr>
<td>FALSE</td>
<td>the PCR does not allow policy</td>
</tr>
</tbody>
</table>

```c
BOOL PCRPolicyIsAvailable(TPMI_DH_PCR handle) // IN: PCR handle
{
    UINT32 groupIndex;
    return PCRBelongsPolicyGroup(handle, &groupIndex);
}
```

8.6.3.5 **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.

```c
void PCRGetAuthValue(TPMI_DH_PCR handle, TPM2B_AUTH *auth) // IN: PCR handle // OUT: authValue of PCR
{
    UINT32 groupIndex;
    if (PCRBelongsAuthGroup(handle, &groupIndex))
    {
        *auth = gc.pcrAuthValues.auth[groupIndex];
    }
    else
    {
        auth->t.size = 0;
    }
    return;
}
```

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.

```c
TPMI_ALG_HASH PCRGetAuthPolicy(TPMI_DH_PCR handle, TPM2B_DIGEST *policy) // IN: PCR handle // OUT: policy of PCR
{
    UINT32 groupIndex;
    if (PCRBelongsPolicyGroup(handle, &groupIndex))
    {
        *policy = gp.pcrPolicies.policy[groupIndex];
        return gp.pcrPolicies.hashAlg[groupIndex];
    }
}```
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.

```c
void PCRSimStart(void)
{
    UINT32 i;
    for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
    {
        gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
        gp.pcrPolicies.policy[i].t.size = 0;
    }
    for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
    {
        gc.pcrAuthValues.auth[i].t.size = 0;
    }
    // We need to give an initial configuration on allocated PCR before
    // receiving any TPM2_PCR_Allocate command to change this configuration
    // When the simulation environment starts, we allocate all the PCRs
    for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
        gp.pcrAllocated.count++)
    {
        gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
        = CryptGetHashAlgByIndex(gp.pcrAllocated.count);
        gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
        = PCR_SELECT_MAX;
        for(i = 0; i < PCR_SELECT_MAX; i++)
            gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
            = 0xFF;
    }
    // Store the initial configuration to NV
    NvWriteReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
    NvWriteReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
    return;
}
```

8.6.3.8 GetSavedPcrPointer()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>no such algorithm</td>
</tr>
<tr>
<td>not NULL</td>
<td>pointer to the 0th byte of the 0th PCR</td>
</tr>
</tbody>
</table>

```c
static BYTE * GetSavedPcrPointer()
```
8.6.3.9 IsPcrAllocated()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>PCR is not allocated</td>
</tr>
<tr>
<td>TRUE</td>
<td>PCR is allocated</td>
</tr>
</tbody>
</table>

```
static BOOL IsPcrAllocated (  
    UINT32              pcr,  // IN: The number of the PCR  
    TPMI_ALG_HASH       hashAlg  // IN: The PCR algorithm  
)  
{  
    UINT32          i;  
    if(pcr >= IMPLEMENTATION_PCR)  
        return FALSE;  
    for(i = 0; i < gp.pcrAllocated.count; i++)  
    {  
        if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)  
            return TRUE;  
        break;  
    }  
    return FALSE;  
}
```
8.6.3.10 GetPcrPointer()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>no such algorithm</td>
</tr>
<tr>
<td>not NULL</td>
<td>pointer to the 0th byte of the 0th PCR</td>
</tr>
</tbody>
</table>

```c
static BYTE *
GetPcrPointer (TPM_ALG_ID alg, UINT32 pcrNumber)
{
    // PCR must be allocated
    pAssert(IsPcrAllocated(pcrNumber, alg) == TRUE);
    switch(alg)
    {
        #ifdef TPM_ALG_SHA1
        case TPM_ALG_SHA1:
            return s_pcrs[pcrNumber].sha1Pcr;
            break;
        #endif
        #ifdef TPM_ALG_SHA256
        case TPM_ALG_SHA256:
            return s_pcrs[pcrNumber].sha256Pcr;
            break;
        #endif
        #ifdef TPM_ALG_SHA384
        case TPM_ALG_SHA384:
            return s_pcrs[pcrNumber].sha384Pcr;
            break;
        #endif
        #ifdef TPM_ALG_SHA512
        case TPM_ALG_SHA512:
            return s_pcrs[pcrNumber].sha512Pcr;
            break;
        #endif
        #ifdef TPM_ALG_SM3_256
        case TPM_ALG_SM3_256:
            return s_pcrs[pcrNumber].sm3_256Pcr;
            break;
        #endif
        default:
        pAssert(FALSE);
            break;
    }
    return NULL;
```
8.6.3.11 IsPcrSelected()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>PCR is not selected</td>
</tr>
<tr>
<td>TRUE</td>
<td>PCR is selected</td>
</tr>
</tbody>
</table>

```c
static BOOL IsPcrSelected ( 
    UINT32 pcr,  // IN: The number of the PCR
    TPMS_PCR_SELECTION *selection  // IN: The selection structure
) {
    if (pcr >= IMPLEMENTATION_PCR)
        return FALSE;
    if (((selection->pcrSelect[pcr/8]) & (1 << (pcr % 8))) != 0)
        return TRUE;
    else
        return FALSE;
}
```

8.6.3.12 FilterPcr()

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

```c
static void FilterPcr( 
    TPMS_PCR_SELECTION *selection  // IN: input PCR selection
) {
    UINT32 i;
    TPMS_PCR_SELECTION *allocated = NULL;

    // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
    for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
        selection->pcrSelect[i] = 0;

    // Find the internal configuration for the bank
    for(i = 0; i < gp.pcrAllocated.count; i++)
        if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
            allocated = &gp.pcrAllocated.pcrSelections[i];
            break;

    for (i = 0; i < selection->sizeofSelect; i++)
        if(allocated == NULL)
            // If the required bank does not exist, clear input selection
            selection->pcrSelect[i] = 0;
        else
            selection->pcrSelect[i] &= allocated->pcrSelect[i];
```
8.6.3.13 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```c
void PCRStartup(
    STARTUP_TYPE type // IN: startup type
)
{
    UINT32 pcr, j;
    UINT32 saveIndex = 0;
    g_pcrReConfig = FALSE;

    if(type != SU_RESUME)
    {
        // PCR generation counter is cleared at TPM_RESET and TPM_RESTART
        gr.pcrCounter = 0;
    }

    // Initialize/Restore PCR values
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        BOOL incrSaveIndex = FALSE;

        // If PCR[0] it was already initialized by H-CRTM, then don't re-initialize
        if(pcr == 0 && g_DrtmPreStartup)
            continue;

        // Iterate each hash algorithm bank
        for(j = 0; j < gp.pcrAllocated.count; j++)
        {
            BYTE *pcrData;
            UINT32 pcrSize;
            pcrSize =
                CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[j].hash);

            if(IsPcrAllocated(pcr, gp.pcrAllocated.pcrSelections[j].hash))
            {
                pcrData =
                    GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);

                if(type == SU_RESUME && s_initAttributes[pcr].stateSave == SET)
                {
                    // Restore saved PCR value
                    BYTE *pcrSavedData;
                    pcrSavedData = GetSavedPcrPointer(
                        gp.pcrAllocated.pcrSelections[j].hash,
                        saveIndex);
                    MemoryCopy(pcrData, pcrSavedData, pcrSize);
                    incrSaveIndex = TRUE;
                }
                else
                {
                    // If the reset locality of the PCR is 4, then
                    // the reset value is all one's, otherwise it is
                    // all zero.
                    if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
                        MemorySet(pcrData, 0xFF, pcrSize);
                }
            }
        }
    }
}
```
8.6.3.14 PCRStateSave()

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

```c
void PCRStateSave(
    TPM_SU type              // IN: startup type
) {
    UINT32 pcr, j;
    UINT32 saveIndex = 0;

    // if state save CLEAR, nothing to be done.  Return here
    if(type == TPM_SU_CLEAR) return;

    // Copy PCR values to the structure that should be saved to NV
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
        {
            BOOL incrSaveIndex = FALSE;

            // Iterate each hash algorithm bank
            for(j = 0; j < gp.pcrAllocated.count; j++)
                {
                    BYTE  *pcrData;
                    UINT32 pcrSize;
                    pcrSize = CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[j].hash);

                    if(IsPcrAllocated(pcr, gp.pcrAllocated.pcrSelections[j].hash))
                        {
                            pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
                            if(s_initAttributes[pcr].stateSave == SET)
                                {
                                    // Restore saved PCR value
                                    BYTE  *pcrSavedData;
                                    pcrSavedData = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, saveIndex);
                                    MemoryCopy(pcrSavedData, pcrData, pcrSize);
                                    incrSaveIndex = TRUE;
                                };
                    }
                }

    // Don't reset PCR[0] if H-CRTM was done
    if(pcr != 0 || !g_DrtmPreStartup)
        MemorySet(pcrData, 0, pcrSize);
}
```

if (incrSaveIndex == TRUE) {
    saveIndex++;
}

return;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>PCR is state saved</td>
</tr>
<tr>
<td>FALSE</td>
<td>PCR is not state saved</td>
</tr>
</tbody>
</table>

BOOL PCRIsStateSaved(
TPMI_DH_PCR handle     // IN: PCR handle to be extended
) {
    UINT32 pcr = handle - PCR_FIRST;
    if (s_initAttributes[pcr].stateSave == SET)
        return TRUE;
    else
        return FALSE;
}

8.6.3.16 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>extend is allowed</td>
</tr>
<tr>
<td>FALSE</td>
<td>extend is not allowed</td>
</tr>
</tbody>
</table>

BOOL PCRIsResetAllowed(
TPMI_DH_PCR handle     // IN: PCR handle to be extended
) {
    UINT8 commandLocality;
    UINT8 localityBits = 1;
    UINT32 pcr = handle - PCR_FIRST;
    // Check for the locality
    commandLocality = _plat__LocalityGet();
    localityBits = localityBits << commandLocality;
    if ((localityBits & s_initAttributes[pcr].resetLocality) == 0)
        return FALSE;
    else
        return TRUE;
}
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.

```c
void
PCRChanged(
    TPM_HANDLE      pcrHandle      // IN: the handle of the PCR that changed.
)

// For the reference implementation, the only change that does not cause
// increment is a change to a PCR in the TCB group.
if(!PCRBelongsTCBGroup(pcrHandle))
    gr.pcrCounter++;
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>extend is allowed</td>
</tr>
<tr>
<td>FALSE</td>
<td>extend is not allowed</td>
</tr>
</tbody>
</table>

```c
BOOL
PCRIsExtendAllowed(
    TPMI_DH_PCR         handle        // IN: PCR handle to be extended
)

// Check for the locality
commandLocality = _plat__LocalityGet();
localityBits = localityBits << commandLocality;
if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
    return FALSE;
else
    return TRUE;
```

8.6.3.19 PCRExpend()

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

```c
void
PCRExend(
    TPMI_DH_PCR          handle,    // IN: PCR handle to be extended
    TPMI_ALG_HASH        hash,      // IN: hash algorithm of PCR
    UINT32               size,      // IN: size of data to be extended
    BYTE                *data       // IN: data to be extended
)
```

```c
UINT32               pcr = handle - PCR_FIRST;
BYTE                *pcrData;
HASH_STATE           hashState;
UINT16               pcrSize;
```
// Extend PCR if it is allocated
if(IsPcrAllocated(pcr, hash))
{
    pcrSize = CryptGetHashDigestSize(hash);
    pcrData = GetPcrPointer(hash, pcr);
    CryptStartHash(hash, &hashState);
    CryptUpdateDigest(&hashState, pcrSize, pcrData);
    CryptUpdateDigest(&hashState, size, data);
    CryptCompleteHash(&hashState, pcrSize, pcrData);

    // If PCR does not belong to TCB group, increment PCR counter
    if(!PCRBelongsTCBGroup(handle))
        gr.pcrCounter++;
}

return;

8.6.3.20 PCRCOMPUTE CURRENT DIGEST()

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.

void
PCRCOMPUTE CURRENT DIGEST(
    TPMI_ALG_HASH     hashAlg,    // IN: hash algorithm to compute digest
    TPML_PCR_SELECTION *selection, // IN/OUT: PCR selection (filtered on output)
    TPM2B_DIGEST      *digest      // OUT: digest
)
{
    HASH_STATE     hashState;
    TPMS_PCR_SELECTION *select;
    BYTE           *pcrData;     // will point to a digest
    UINT32         pcrSize;
    UINT32         pcr;
    UINT32         i;

    // Initialize the hash
digest->t.size = CryptStartHash(hashAlg, &hashState);
pAssert(digest->t.size > 0 & & digest->t.size < UINT16_MAX);

    // Iterate through the list of PCR selection structures
    for(i = 0; i < selection->count; i++)
    {
        // Point to the current selection
        select = &selection->pcrSelections[i]; // Point to the current selection
        FilterPcr(select); // Clear out the bits for unimplemented PCR

        // Need the size of each digest
        pcrSize = CryptGetHashDigestSize(selection->pcrSelections[i].hash);

        // Iterate through the selection
        for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
        {
            if(IsPcrSelected(pcr, select)) // Is this PCR selected
            {
                // Get pointer to the digest data for the bank
                pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
                CryptUpdateDigest(&hashState, pcrSize, pcrData); // add to digest
            }
        }
    }
}
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.

```c
void PCRRead(
    TPML_PCR_SELECTION *selection,  // IN/OUT: PCR selection (filtered on output)
    TPML_DIGEST *digest,            // OUT: digest
    UINT32 *pcrCounter              // OUT: the current value of PCR generation number
)
{
    TPMS_PCR_SELECTION *select;
    BYTE pcrData;                // will point to a digest
    UINT32 pcr;
    UINT32 i;

    digest->count = 0;

    // Iterate through the list of PCR selection structures
    for(i = 0; i < selection->count; i++)
    {
        // Point to the current selection
        select = &selection->pcrSelections[i];  // Point to the current selection
        FilterPcr(select);                      // Clear out the bits for unimplemented PCR

        // Iterate through the selection
        for (pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
        {
            if(IsPcrSelected(pcr, select))  // Is this PCR selected
            {
                // Check if number of digest exceed upper bound
                if(digest->count > 7)
                {
                    // Clear rest of the current select bitmap
                    while(  pcr < IMPLEMENTATION_PCR
                        // do not round up!
                        && (pcr / 8) < selection->sizeofSelect)
                    {
                        // do not round up!
                        select->pcrSelect[pcr/8] &= (BYTE) ~(1 << (pcr % 8));
                        pcr++;
                    }
                    // Exit inner loop
                    break;;
                }
            }
            // Need the size of each digest
            digest->digests[digest->count].t.size =
                CryptGetHashDigestSize(selection->pcrSelections[i].hash);

            // Get pointer to the digest data for the bank
            pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
            // Add to the data to digest
            MemoryCopy(digest->digests[digest->count].t.buffer, pcrData,
                digest->digests[digest->count].t.size);
        }
    }
    // Complete hash stack
    CryptCompleteHash2B(&hashState, &digest->b);

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

This function is used to change the PCR allocation.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>allocate success</td>
</tr>
<tr>
<td>NO</td>
<td>allocate fail</td>
</tr>
</tbody>
</table>

TPMI_YES_NO

PCRAlocate()

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

void
733 PCRSetValue(
734     TPM_HANDLE          handle,       // IN: the handle of the PCR to set
735     INT8                 initialValue  // IN: the value to set
736 )
737 {
738     int       i;
739     UInt32    pcr = handle - PCR_FIRST;
740     TPM_ALG_HASH hash;
741     UInt16    digestSize;
742     BYTE      *pcrData;
743
744     // Iterate supported PCR bank algorithms to reset
745     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.

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


8.6.3.25 PCRCapGetAllocation()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES:</td>
<td>if the return count is 0</td>
</tr>
<tr>
<td>NO:</td>
<td>if the return count is not 0</td>
</tr>
</tbody>
</table>

```c
TPML_PCR_SELECTION GetAllocation(  
    UINT32 count,  // IN: count of return  
    TPML_PCR_SELECTION *pcrSelection  // OUT: PCR allocation list
)
```

8.6.3.26 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```c
static void PCRSetSelectBit(  
    UINT32 pcr,  // IN: PCR number  
    BYTE bitmap  // OUT: bit map to be set
)
```

8.6.3.27 PCRGetProperty()

This function returns the selected PCR property.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>the property type is implemented</td>
</tr>
<tr>
<td>FALSE</td>
<td>the property type os not implemented</td>
</tr>
</tbody>
</table>

```c
static BOOL PCRGetProperty(  
    TPM_PT_PCR property,  
    TPMS_TAGGED_PCR_SELECT *select
)
```
// Always set the bitmap to be the size of all PCR
select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;

// Initialize bitmap
MemorySet(select->pcrSelect, 0, select->sizeofSelect);

// Collecting properties
for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
{
    switch(property)
    {
    case TPM_PT_PCR_SAVE:
        if(s_initAttributes[pcr].stateSave == SET)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_EXTEND_L0:
        if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_RESET_L0:
        if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_EXTEND_L1:
        if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_RESET_L1:
        if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_EXTEND_L2:
        if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_RESET_L2:
        if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_EXTEND_L3:
        if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_RESET_L3:
        if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_EXTEND_L4:
        if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_RESET_L4:
        if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    case TPM_PT_PCR_DRTM_RESET:
        // DRTM reset PCRs are the PCR reset by locality 4
        if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    #if NUM_POLICY_PCR_GROUP > 0
    case TPM_PT_PCR_POLICY:
        if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
    #endif
    }
}
#if NUM_AUTHVALUE_PCR_GROUP > 0
    case TPM_PT_PCR_AUTH:
        if (PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
#endif
#if ENABLE_PCR_NO_INCREMENT == YES
    case TPM_PT_PCR_NO_INCREMENT:
        if (PCRBelongsTCBGroup(pcr + PCR_FIRST))
            PCRSetSelectBit(pcr, select->pcrSelect);
        break;
#endif
    default:
        // If property is not supported, stop scanning PCR attributes
        // and return.
        return FALSE;
        break;
    }
} return TRUE;
}

8.6.3.28 PCRCapGetProperties()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES:</td>
<td>if no more property is available</td>
</tr>
<tr>
<td>NO:</td>
<td>if there are more properties not reported</td>
</tr>
</tbody>
</table>

TPMI_YES_NO PCRCapGetProperties(
    TPM_PT_PCR property,    // IN: the starting PCR property
    UINT32 count,           // IN: count of returned
    // properties
    TPML_TAGGED_PCR_PROPERTY *select    // OUT: PCR select
)
{
    TPMI_YES_NO more = NO;
    UINT32    i;
    // Initialize output property list
    select->count = 0;
    // The maximum count of properties we may return is MAX_PCR_PROPERTIES
    if (count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
    // TPM_PT_PCR_FIRST is defined as 0 in spec. It ensures that property
    // value would never be less than TPM_PT_PCR_FIRST
    pAssert(TPM_PT_PCR_FIRST == 0);
    // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
    // implemented on the TPM.
    for (i = property; i <= TPM_PT_PCR_LAST; i++)
    {
        if (select->count < count)
        {
            // If we have not filled up the return list, add more properties to it
            if (PCRGetProperty(i, &select->pcrProperty[select->count]))
                // only increment if the property is implemented
                select->count++;
        }
    }
else
{
    // If the return list is full but we still have properties
    // available, report this and stop iterating.
    more = YES;
    break;
}
}
return more;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

TPMI_YES_NO
PCRCapGetHandles(
    TPMI_DH_PCR             handle,       // IN: start handle
    UINT32                  count,        // IN: count of returned handles
    TPML_HANDLE             *handleList   // OUT: list of handle
)
{
    TPMI_YES_NO     more = NO;
    UINT32          i;
    pAssert(HandleGetType(handle) == TPM_HT_PCR);
    // Initialize output handle list
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    // Iterate PCR handle range
    for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
    {
        if(handleList->count < count)
        {
            // If we have not filled up the return list, add this PCR
            // handle to it
            handleList->handle[handleList->count] = i + PCR_FIRST;
            handleList->count++;
        }
        else
        {
            // If the return list is full but we still have PCR handle
            // available, report this and stop iterating
            more = YES;
            break;
        }
    }
    return more;
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

```c
#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.

```c
void PhysicalPresencePreInstall_Init(void)
{
    // Clear all the PP commands
    MemorySet(&gp.ppList, 0,
              ((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7) / 8);

    // TPM_CC_PP_Commands always requires PP
    if(CommandIsImplemented(TPM_CC_PP_Commands))
        PhysicalPresenceCommandSet(TPM_CC_PP_Commands);

    // Write PP list to NV
    NvWriteReserved(NV_PP_LIST, &gp.ppList);

    return;
}
```

8.7.3.2 PhysicalPresenceCommandSet()

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

```c
void PhysicalPresenceCommandSet(TPM_CC commandCode)
{
    UINT32 bitPos;

    // Assume command is implemented. It should be checked before this
    // function is called
    pAssert(CommandIsImplemented(commandCode));

    // If the command is not a PP command, ignore it
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return;

    bitPos = commandCode - TPM_CC_PP_FIRST;

    // Set bit
    gp.ppList[bitPos/8] |= 1 << (bitPos % 8);
8.7.3.3 PhysicalPresenceCommandClear()

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

```c
void PhysicalPresenceCommandClear(
    TPM_CC    commandCode  // IN: command code
) {
    UINT32    bitPos;
    // Assume command is implemented. It should be checked before this
    // function is called
    pAssert(CommandIsImplemented(commandCode));
    // If the command is not a PP command, ignore it
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return;
    // if the input code is TPM_CC_PP_Commands, it can not be cleared
    if(commandCode == TPM_CC_PP_Commands)
        return;
    bitPos = commandCode - TPM_CC_PP_FIRST;
    // Set bit
    gp.ppList[bitPos/8] |= (1 << (bitPos % 8));
    // Flip it to off
    gp.ppList[bitPos/8] ^= (1 << (bitPos % 8));
    return;
}
```

8.7.3.4 PhysicalPresenceIsRequired()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if physical presence is required</td>
</tr>
<tr>
<td>FALSE</td>
<td>if physical presence is not required</td>
</tr>
</tbody>
</table>

```c
BOOL PhysicalPresenceIsRequired(
    TPM_CC    commandCode  // IN: command code
) {
    UINT32    bitPos;
    // if the input commandCode is not a PP command, return FALSE
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return FALSE;
    bitPos = commandCode - TPM_CC_PP_FIRST;
    // Check the bit map. If the bit is SET, PP authorization is required
    return ((gp.ppList[bitPos/8] & (1 << (bitPos % 8))) != 0);
}
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more command codes available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available command codes have been returned</td>
</tr>
</tbody>
</table>

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

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

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.

```c
static void ContextIdSetOldest(void)
{
    CONTEXT_SLOT lowBits;
    CONTEXT_SLOT entry;
    CONTEXT_SLOT smallest = ((CONTEXT_SLOT) -0);
    UINT32 i;

    // Set oldestSaveContext to a value indicating none assigned
    s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;

    lowBits = (CONTEXT_SLOT)gr.contextCounter;
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        entry = gr.contextArray[i];

        // only look at entries that are saved contexts
        if(entry > MAX_LOADED_SESSIONS)
        {
            // Use a less than or equal in case the oldest
            // is brand new (= lowBits-1) and equal to our initial
            // value for smallest.
            if(((CONTEXT_SLOT) (entry - lowBits)) <= smallest)
            {
                smallest = (entry - lowBits);
                s_oldestSavedSession = i;
            }
        }
    }
}
```

8.8.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```c
void SessionStartup(
    STARTUP_TYPE type
)
{
    UINT32 i;

    // Initialize session slots. At startup, all the in-memory session slots
    // are cleared and marked as not occupied
    for(i = 0; i < MAX_LOADED_SESSIONS; i++)
```
s_sessions[i].occupied = FALSE;    // session slot is not occupied

// The free session slots the number of maximum allowed loaded sessions
s_freeSessionSlots = MAX_LOADED_SESSIONS;

// Initialize context ID data. On a ST_SAVE or hibernate sequence, it will
// scan the saved array of session context counts, and clear any entry that
// references a session that was in memory during the state save since that
// memory was not preserved over the ST_SAVE.
if (type == SU_RESUME || type == SU_RESTART)
{
    // On ST_SAVE we preserve the contexts that were saved but not the ones
    // in memory
    for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        // If the array value is unused or references a loaded session then
        // that loaded session context is lost and the array entry is
        // reclaimed.
        if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
            gr.contextArray[i] = 0;
    }
    // Find the oldest session in context ID data and set it in
    // s_oldestSavedSession
    ContextIdSetOldest();

    else
    {
        // For STARTUP_CLEAR, clear out the contextArray
        for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
            gr.contextArray[i] = 0;

        // reset the context counter
        gr.contextCounter = MAX_LOADED_SESSIONS + 1;

        // Initialize oldest saved session
        s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;

        return;
    }
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if session is loaded</td>
</tr>
<tr>
<td>FALSE</td>
<td>if it is not loaded</td>
</tr>
</tbody>
</table>

BOOL SessionIsLoaded(
    TPM_HANDLE handle     // IN: session handle
)
{
    pAssert(  HandleGetType(handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
8.8.5.2 SessionIsSaved()

This function tests if a session handle references a saved session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE: An password authorization does not have a session.

This function requires that the handle be a valid session handle.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if session is saved</td>
</tr>
<tr>
<td>FALSE</td>
<td>if it is not saved</td>
</tr>
</tbody>
</table>

8.8.5.3 SessionPCRValuesIsCurrent()

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

This function requires the session is loaded.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if PCR value is current</td>
</tr>
<tr>
<td>FALSE</td>
<td>if PCR value is not current</td>
</tr>
</tbody>
</table>
SessionPCRValueIsCurrent(
TPMI_SH_POLICY handle // IN: session handle
) {
    SESSION *session;
    pAssert(SessionIsLoaded(handle));
    session = SessionGet(handle);
    if (session->pcrCounter != 0 && session->pcrCounter != gr.pcrCounter)
        return FALSE;
    else
        return TRUE;
}

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.

SESSION *
SessionGet(
TPM_HANDLE handle // IN: session handle
) {
    CONTEXT_SLOT sessionIndex;
    pAssert( HandleGetType(handle) == TPM_HT_POLICY_SESSION 
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION 
            );
    pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);
    // get the contents of the session array. Because session is loaded, we 
    // should always get a valid sessionIndex
    sessionIndex = gr.contextArray[handle & HR_HANDLE_MASK] - 1;
    pAssert(sessionIndex < MAX_LOADED_SESSIONS);
    return &s_sessions[sessionIndex].session;
}

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.
### 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

```c
TPM_RC
SessionCreate(

    TPM_SE               sessionType, // IN: the session type
    TPMI_ALG_HASH        authHash, // IN: the hash algorithm
    TPM2B_NONCE         *nonceCaller, // IN: initial nonceCaller
    TPM_SYM_DEF        *symmetric, // IN: the symmetric algorithm
    TPMI_DH_ENTITY       bind, // IN: the bind object
    TPM2B_DATA          *seed, // IN: seed data
    TPM_HANDLE          *sessionHandle // OUT: the session handle
)
```

```c
TPM_RC               result = TPM_RC_SUCCESS;
CONTEXT_SLOT        slotIndex;
SESSION             *session = NULL;
pAssert(   sessionType == TPM_SE_HMAC
                  || sessionType == TPM_SE_POLICY
                  || sessionType == TPM_SE_TRIAL);

// If there are no open spots in the session array, then no point in searching
if(s_freeSessionSlots == 0)
    return TPM_RC_SESSION_MEMORY;

// Find a space for loading a session
for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
{
    // Is this available?
    if(s_sessions[slotIndex].occupied == FALSE)
    {
        session = &s_sessions[slotIndex].session;
        break;
    }
}

// if no spot found, then this is an internal error
pAssert (slotIndex < MAX_LOADED_SESSIONS);

// Call context ID function to get a handle. TPM_RC_SESSION_HANDLE may be
// returned from ContextIdHandelAssign()
result = ContextIdSessionCreate(sessionHandle, slotIndex);
if(result != TPM_RC_SUCCESS)
    return result;

//*** Only return from this point on is TPM_RC_SUCCESS
```

```c
// Can now indicate that the session array entry is occupied.
s_freeSessionSlots--;
s_sessions[slotIndex].occupied = TRUE;

// Initialize the session data
MemorySet(session, 0, sizeof(SESSION));

// Initialize internal session data
session->authHashAlg = authHash;

// Initialize session type
if(sessionType == TPM_SE_HMAC)
{
    *sessionHandle += HMAC_SESSION_FIRST;
```
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  
280     // Create initial session nonce  
281     session->nonceTPM.t.size = nonceCaller->t.size;  
282     CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);  
283  
284     // Set up session parameter encryption algorithm  
285     session->symmetric = *symmetric;  
286  
287     // If there is a bind object or a session secret, then need to compute  
288     // a sessionKey.  
289     if(bind != TPM_RH_NULL || seed->t.size != 0)  
290     {  
291         // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,  
292                     nonceCaller, bits)  
293         // The HMAC key for generating the sessionSecret can be the concatenation  
294         // of an authorization value and a seed value  
295         TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));  
296         TPM2B_KEY            key;  
297  
298         UINT16               hashSize;  // The size of the hash used by the  
299                     // session created by this command  
300         TPM2B_AUTH  entityAuth;  // The authValue of the entity  
301                     // associated with HMAC session  
302  
303         // Get hash size, which is also the length of sessionKey  
304         hashSize = CryptGetHashDigestSize(session->authHashAlg);  
305  
306         // Get authValue of associated entity  
307         entityAuth.t.size = EntityGetAuthValue(bind, &entityAuth.t.buffer[0]);  
308  
309         // Concatenate authValue and seed  
310         MemoryCopy2B(&key.b, &entityAuth.b);  
311         MemoryConcat2B(&key.b, &seed->b);  
312  
313         // Compute the session key  
314         KDFa(session->authHashAlg, &key.b, "ATH", &session->nonceTPM.b,  
315                     &nonceCaller->b, hashSize * 8, session->sessionKey.t.buffer, NULL);  
316     }  
317  
318     // Copy the name of the entity that the HMAC session is bound to  
319     // Policy session is not bound to an entity  
320     if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)  
321     {  
322         session->attributes.isBound = SET;  
323         SessionComputeBoundEntity(bind, &session->ul.boundEntity);  
324     }  
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
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CONTEXT_GAP</td>
<td>a contextID could not be assigned.</td>
</tr>
<tr>
<td>TPM_RC_TOO_MANY_CONTEXTS</td>
<td>the counter maxed out</td>
</tr>
</tbody>
</table>

```c
TPM_RC SessionContextSave (  
  TPM_HANDLE          handle,  // IN: session handle
  CONTEXT_COUNTER     *contextID       // OUT: assigned contextID
)
{
  UINT32                      contextIndex;
  CONTEXT_SLOT                slotIndex;
  
  pAssert(SessionIsLoaded(handle));
  
  // check to see if the gap is already maxed out
  // Need to have a saved session
  if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
          // if the oldest saved session has the same value as the low bits
          // of the contextCounter, then the GAP is maxed out.
          && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
        return TPM_RC_CONTEXT_GAP;
  
  // if the caller wants the context counter, set it
  if(contextID != NULL)
    *contextID = gr.contextCounter;
  
  pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);

  contextIndex = handle & HR_HANDLE_MASK;
  
  // Extract the session slot number referenced by the contextArray
  // because we are going to overwrite this with the low order
  // contextID value.
  slotIndex = gr.contextArray[contextIndex] - 1;
  
  // Set the contextID for the contextArray
  gr.contextArray[contextIndex] = (CONTEXT SLOT)gr.contextCounter;
  
  // Increment the counter
  gr.contextCounter++;
  
  // In the unlikely event that the 64-bit context counter rolls over...
  if(gr.contextCounter == 0)
    {  
      // back it up
      gr.contextCounter--;  
```
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SESSION_MEMORY</td>
<td>no free session slots</td>
</tr>
<tr>
<td>TPM_RC_CONTEXT_GAP</td>
<td>the gap count is maximum and this is not the oldest saved context</td>
</tr>
</tbody>
</table>

```c
TPM_RC SessionContextLoad(
    SESSION                *session,       // IN: session structure from saved
    // context
    TPM_HANDLE             *handle        // IN/OUT: session handle
)
{
    UINT32              contextIndex;
    CONTEXT_SLOT        slotIndex;

    pAssert(   HandleGetType(*handle) == TPM_HT_POLICY_SESSION
                 || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);

    // Don't bother looking if no openings
    if(s_freeSessionSlots == 0)
    {
        return TPM_RC_SESSION_MEMORY;
    }

    // Find a free session slot to load the session
    for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
    {
        if(s_sessions[slotIndex].occupied == FALSE) break;
    }

    // if no spot found, then this is an internal error
    pAssert (slotIndex < MAX_LOADED_SESSIONS);

    contextIndex = *handle & HR_HANDLE_MASK;  // extract the index

    // If there is only one slot left, and the gap is at maximum, the only session
    // context that we can safely load is the oldest one.
```
if (s_oldestSavedSession < MAX_ACTIVE_SESSIONS
    && s_freeSessionSlots == 1
    && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
    && contextIndex != s_oldestSavedSession
) 
    return TPM_RC_CONTEXT_GAP;

pAssert(contextIndex < MAX_ACTIVE_SESSIONS);

// set the contextArray value to point to the session slot where
// the context is loaded
gr.contextArray[contextIndex] = slotIndex + 1;

if (contextIndex == s_oldestSavedSession)
    ContextIdSetOldest();

// Copy session data to session slot
s_sessions[slotIndex].session = *session;

// Set session slot as occupied
s_sessions[slotIndex].occupied = TRUE;

// Reduce the number of open spots
s_freeSessionSlots--;

return TPM_RC_SUCCESS;

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.

void
SessionFlush(
    TPM_HANDLE           handle     // IN: loaded or saved session handle
)
{
    CONTEXT_SLOT         slotIndex;
    UINT32               contextIndex; // Index into contextArray

    pAssert(   (   HandleGetType(handle) == TPM_HT_POLICY_SESSION
                   || HandleGetType(handle) == TPM_HT_HMAC_SESSION
                   )
            && (SessionIsLoaded(handle)  || SessionIsSaved(handle))
        );

    // Flush context ID of this session
    // Convert handle to an index into contextArray
    contextIndex = handle & HR_HANDLE_MASK;

    // Get the current contents of the array
    slotIndex = gr.contextArray[contextIndex];

    // Mark context array entry as available
    gr.contextArray[contextIndex] = 0;

    // Is this a saved session being flushed
    if (slotIndex > MAX_LOADED_SESSIONS)
        { // Flushing the oldest session?
if(contextIndex == s_oldestSavedSession)
    // If so, find a new value for oldest.
    ContextIdSetOldest();
else
    {
        // Adjust slot index to point to session array index
        slotIndex -= 1;
        // Free session array index
        s_sessions[slotIndex].occupied = FALSE;
        s_freeSessionSlots++;
    }
return;

8.8.6.6 SessionComputeBoundEntity()

This function computes the binding value for a session. The binding value for a reserved handle is the handle itself. For all the other entities, the authValue at the time of binding is included to prevent squatting. For those values, the Name and the authValue are concatenated into the bind buffer. If they will not both fit, the will be overlapped by XORing() bytes. If XOR is required, the bind value will be full.

void SessionComputeBoundEntity(
  TPMI_DH_ENTITY entityHandle, // IN: handle of entity
  TPM2B_NAME *bind // OUT: binding value
)
{
    TPM2B_AUTH auth;
    INT16 overlap;
    // Get name
    bind->t.size = EntityGetName(entityHandle, bind->t.name);
    // The bound value of a reserved handle is the handle itself
    if(bind->t.size == sizeof(TPM_HANDLE)) return;
    // For all the other entities, concatenate the auth value to the name.
    // Get a local copy of the auth value because some overlapping
    // may be necessary.
    auth.t.size = EntityGetAuthValue(entityHandle, auth.t.buffer);
    pAssert(auth.t.size <= <K>sizeof(TPMU_HA));
    // Figure out if there will be any overlap
    overlap = bind->t.size + auth.t.size - sizeof(bind->t.name);
    // There is overlap if the combined sizes are greater than will fit
    if(overlap > 0)
    {
        // The overlap area is at the end of the Name
        BYTE *result = &bind->t.name[bind->t.size - overlap];
        int i;
        // XOR the auth value into the Name for the overlap area
        for(i = 0; i < overlap; i++)
            result[i] ^= auth.t.buffer[i];
    }
    else
    {
        // There is no overlap
        overlap = 0;
    }
//copy the remainder of the authData to the end of the name
MemoryCopy(&bind->t.name[bind->t.size], &auth.t.buffer[overlap],
        auth.t.size - overlap);

// Increase the size of the bind data by the size of the auth - the overlap
bind->t.size += auth.t.size-overlap;
return;
}

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.

void SessionInitPolicyData(
    SESSION         *session  // IN: session handle
)
{
    // Initialize start time
    session->startTime = go.clock;

    // Initialize policyDigest. policyDigest is initialized with a string of 0 of
    // session algorithm digest size. Since the policy already contains all zeros
    // it is only necessary to set the size
    session->u2.policyDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
    return;
}

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.

void SessionResetPolicyData(
    SESSION         *session  // IN: the session to reset
)
{
    session->commandCode = 0;  // No command

    // No locality selected
    MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));

    // The cpHash size to zero
    session->u1.cpHash.b.size = 0;

    // Reset the pcrCounter
    session->pcrCounter = 0;

    // Reset the policy hash
    MemorySet(&session->u2.policyDigest.t.buffer, 0, session->u2.policyDigest.t.size);

    // Reset the session attributes
    MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));

    // set the policy attribute
    session->attributes.isPolicy = SET;
}
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO SessionCapGetLoaded(
    TPMI_SH_POLICY handle, // IN: start handle
    UINT32 count, // IN: count of returned handles
    TPML_HANDLE *handleList // OUT: list of handle
)
{
    TPMI_YES_NO more = NO;
    UINT32 i;
    pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
    // Initialize output handle list
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    // Iterate session context ID slots to get loaded session handles
    for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
    {
        // If session is active
        if(gr.contextArray[i] != 0)
        {
            // If session is loaded
            if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
            {
                if(handleList->count < count)
                {
                    SESSION *session;
                    // If we have not filled up the return list, add this
                    // session handle to it
                    // assume that this is going to be an HMAC session
                    handle = i + HMAC_SESSION_FIRST;
                    session = SessionGet(handle);
                    if(session->attributes.isPolicy)
                        handle = i + POLICY_SESSION_FIRST;
                    handleList->handle[handleList->count] = handle;
                    handleList->count++;
                }
            }
        }
    } // If the return list is full but we still have loaded object
    else
    {
        // available, report this and stop iterating
        more = YES;
        break;
    }
    return more;
```
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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```
TPM_YES_NO SessionCapGetSaved(
    TPMI_SH_HMAC handle, // IN: start handle
    UINT32 count, // IN: count of returned handles
    TPMI_HANDLE *handleList // OUT: list of handle
) {

    TPMI_YES_NO more = NO;
    UINT32 i;

    pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if (count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    // Iterate session context ID slots to get loaded session handles
    for (i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSION; i++)
        {
            // If session is active
            if (gr.contextArray[i] != 0)
                {
                    // If session is saved
                    if (gr.contextArray[i] > MAX_LOADED_SESSIONS)
                        {
                            if (handleList->count < count)
                                {
                                    // If we have not filled up the return list, add this
                                    // session handle to it
                                    handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
                                    handleList->count++; // Increment count
                                }
                            else
                                {
                                    // If the return list is full but we still have loaded object
                                    // available, report this and stop iterating
                                    more = YES;
                                    break;
                                }
                        }
                    }
            }
        }

    return more;
```
8.8.6.11  SessionCapGetLoadedNumber()

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

```c
UINT32 SessionCapGetLoadedNumber(void)
{
    return MAX_LOADED_SESSIONS - s_freeSessionSlots;
}
```

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.

```c
UINT32 SessionCapGetLoadedAvail(void)
{
    return s_freeSessionSlots;
}
```

8.8.6.13  SessionCapGetActiveNumber()

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

```c
UINT32 SessionCapGetActiveNumber(void)
{
    UINT32 i;
    UINT32 num = 0;
    // Iterate the context array to find the number of non-zero slots
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        if(gr.contextArray[i] != 0) num++;
    }
    return num;
}
```

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.

```c
UINT32 SessionCapGetActiveAvail(void)
{
    UINT32 i;
    UINT32 num = 0;
    // Iterate the context array to find the number of zero slots
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        if(gr.contextArray[i] == 0) num++;
    }
    return num;
}
```
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

```c
#include "InternalRoutines.h"
#include "Platform.h"
```

8.9.3 Functions

8.9.3.1 TimePowerOn()

This function initialize time info at _TPM_Init().

```c
void
TimePowerOn (void)
{
    TPM_SU orderlyShutDown;
    // Read time info from NV memory
    NvReadReserved(NV_CLOCK, &go.clock);
    // Read orderly shut down state
    NvReadReserved(NV_ORDERLY, &orderlyShutDown);
    // If the previous cycle is orderly shut down, the value of the safe bit
    // the same as previously saved. Otherwise, it is not safe.
    if(orderlyShutDown == SHUTDOWN_NONE)
        go.clockSafe = NO;
    else
        go.clockSafe = YES;
    // Clear time
    g_time = 0;
    return;
}
```

8.9.3.2 TimeStartup()

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

```c
void
TimeStartup (STARTUP_TYPE type)  // IN: start up type
{
    if(type == SU_RESUME)
    {
```
// Resume sequence
    gr.restartCount++;
}  
else
{
    if(type == SU_RESTART)
    {
        // Hibernate sequence
        gr.clearCount++;
        gr.restartCount++;
    }
    else
    {
        // Reset sequence
        // Increase resetCount
        gp.resetCount++;

        // Write resetCount to NV
        NvWriteReserved(NV_RESET_COUNT, &gp.resetCount);
        gp.totalResetCount++;

        // We do not expect the total reset counter overflow during the life
        // time of TPM.  if it ever happens, TPM will be put to failure mode
        // and there is no way to recover it.
        // The reason that there is no recovery is that we don't increment
        // the NV totalResetCount when incrementing would make it 0. When the
        // TPM starts up again, the old value of totalResetCount will be read
        // and we will get right back here with the increment failing.
        if(gp.totalResetCount == 0)
            FAIL(FATAL_ERROR_INTERNAL);

        // Write total reset counter to NV
        NvWriteReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);

        // Reset restartCount
        gr.restartCount = 0;
    }

    return;
}

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.

void
TimeUpdateToCurrent(void)
{
    UINT64 oldClock;
    UINT64 elapsed;
#define CLOCK_UPDATE_MASK ((1ULL << NV_CLOCK_UPDATE_INTERVAL) - 1)

// Can't update time during the dark interval or when rate limiting.
if (NvIsAvailable() != TPM_RC_SUCCESS)
    return;

// Save the old clock value
oldClock = go.clock;

// Update the time info to current
elapsed = _plat__ClockTimeElapsed();
go.clock += elapsed;
g_time += elapsed;

// Check to see if the update has caused a need for an nvClock update
// CLOCK_UPDATE_MASK is measured by second, while the value in go.clock is
// recorded by millisecond.  Align the clock value to second before the bit
// operations
if( ((go.clock/1000) | CLOCK_UPDATE_MASK) > ((oldClock/1000) | CLOCK_UPDATE_MASK))
{
    NvWriteReserved(NV_CLOCK,&go.clock);
    // Now the time state is updated
    go.clockSafe = YES;
}

// Call self healing logic for dictionary attack parameters
DASelfHeal();
return;

8.9.3.4 TimeSetAdjustRate()

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

void TimeSetAdjustRate(
    TPM_CLOCK_ADJUST adjust, // IN: adjust constant
)
{
    switch (adjust)
    {
    case TPM_CLOCK_COARSE_SLOWER:
        _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
        break;
    case TPM_CLOCK_COARSE_FASTER:
        _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
        break;
    case TPM_CLOCK_MEDIUM_SLOWER:
        _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
        break;
    case TPM_CLOCK_MEDIUM_FASTER:
        _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
        break;
    case TPM_CLOCK_FINE_SLOWER:
        _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
        break;
    case TPM_CLOCK_FINE_FASTER:
        _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
        break;
    case TPM_CLOCK_NO_CHANGE:
        break;
    default:
    pAssert(FALSE);
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_RANGE</td>
<td>invalid data range</td>
</tr>
</tbody>
</table>

```
TPM_RC
TimeGetRange(
    UINT16 offset,   // IN: offset in TPMS_TIME_INFO
    UINT16 size,    // IN: size of data
    BYTE *dataBuffer // OUT: result buffer
)
```

8.9.3.6 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```
void
TimeFillInfo(
    TPMS_CLOCK_INFO *clockInfo
)
```

```
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

```c
#include "InternalRoutines.h"
typedef struct
{
    TPM_ALG_ID        algID;
    TPMA_ALGORITHM    attributes;
} ALGORITHM;
static const ALGORITHM s_algorithms[] = {
#ifdef TPM_ALG_RSA
{TPM_ALG_RSA,        {1, 0, 0, 1, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_DES
{TPM_ALG_DES,        {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_3DES
{TPM_ALG__3DES,       {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA1
{TPM_ALG_SHA1,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_HMAC
{TPM_ALG_HMAC,        {0, 0, 1, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_AES
{TPM_ALG_AES,        {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_MGF1
{TPM_ALG_MGF1,        {0, 0, 1, 0, 0, 0, 0, 0, 1}},
#endif
#ifdef TPM_ALG_XOR
{TPM_ALG_XOR,        {0, 1, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA256
{TPM_ALG_SHA256,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA384
{TPM_ALG_SHA384,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA512
{TPM_ALG_SHA512,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_WHIRLPOOL512
{TPM_ALG_WHIRLPOOL512, {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SM3_256
{TPM_ALG_SM3_256,     {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif

#ifdef TPM_ALG_KEYEDHASH
{TPM_ALG_KEYEDHASH,   {0, 0, 1, 1, 0, 1, 1, 0, 0}},
#endif
#ifdef TPM_ALG_SHA256
{TPM_ALG_SHA256,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA384
{TPM_ALG_SHA384,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA512
{TPM_ALG_SHA512,      {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_WHIRLPOOL512
{TPM_ALG_WHIRLPOOL512, {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SM3_256
{TPM_ALG_SM3_256,     {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
```
#ifdef TPM_ALG_SM4
{TPM_ALG_SM4, {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif

#ifdef TPM_ALG_RSASSA
{TPM_ALG_RSASSA, {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif

#ifdef TPM_ALG_RSAES
{TPM_ALG_RSAES, {1, 0, 0, 0, 0, 0, 0, 1, 0}},
#endif

#ifdef TPM_ALG_RSAPSS
{TPM_ALG_RSAPSS, {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif

#ifdef TPM_ALG_OAEP
{TPM_ALG_OAEP, {1, 0, 0, 0, 0, 0, 1, 0, 0}},
#endif

#ifdef TPM_ALG_ECDSA
{TPM_ALG_ECDSA, {1, 0, 0, 0, 0, 1, 0, 1, 0}},
#endif

#ifdef TPM_ALG_ECDH
{TPM_ALG_ECDH, {1, 0, 0, 0, 0, 0, 0, 1, 0}},
#endif

#ifdef TPM_ALG_ECDAA
{TPM_ALG_ECDAA, {1, 0, 0, 0, 0, 0, 1, 0, 0}},
#endif

#ifdef TPM_ALG_ECSCHNORR
{TPM_ALG_ECSCHNORR, {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif

#ifdef TPM_ALG_KDF1_SP800_56a
{TPM_ALG_KDF1_SP800_56a, {0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif

#ifdef TPM_ALG_KDF2
{TPM_ALG_KDF2, {0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif

#ifdef TPM_ALG_KDF1_SP800_108
{TPM_ALG_KDF1_SP800_108, {0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif

#ifdef TPM_ALG_ECC
{TPM_ALG_ECC, {1, 0, 0, 1, 0, 0, 0, 0, 0}},
#endif

#ifdef TPM_ALG_SYMCIPHER
{TPM_ALG_SYMCIPHER, {0, 0, 0, 1, 0, 0, 0, 0, 0}},
#endif

#ifdef TPM_ALG_CTR
{TPM_ALG_CTR, {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif

#ifdef TPM_ALG_OFB
{TPM_ALG_OFB, {0, 1, 0, 0, 0, 0, 0, 1, 0}},
#endif

#ifdef TPM_ALG_CBC
{TPM_ALG_CBC, {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif

#ifdef TPM_ALG_CFB
{TPM_ALG_CFB, {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif

#ifdef TPM_ALG_ECB
{TPM_ALG_ECB, {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif

9.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2_GetCapability() to return a list of the implemented algorithms.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>more algorithms to report</td>
</tr>
<tr>
<td>NO</td>
<td>no more algorithms to report</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO AlgorithmCapGetImplemented(
    TPM_ALG_ID algID,           // IN: the starting algorithm ID
    UINT32 count,               // IN: count of returned algorithms
    TPML_ALG_PROPERTY *algList   // OUT: algorithm list
)
{
    TPMI_YES_NO more = NO;
    UINT32 i;
    UINT32 algNum;

    // initialize output algorithm list
    algList->count = 0;

    // The maximum count of algorithms we may return is MAX_CAP_ALGS.
    if (count > MAX_CAP_ALGS) count = MAX_CAP_ALGS;

    // Compute how many algorithms are defined in s_algorithms array.
    algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);

    // Scan the implemented algorithm list to see if there is a match to 'algID'.
    for (i = 0; i < algNum; i++)
    {
        // If algID is less than the starting algorithm ID, skip it
        if (s_algorithms[i].algID < algID)
            continue;

        if (algList->count < count)
        {
            // If we have not filled up the return list, add more algorithms
            // to it
            algList->algProperties[algList->count].alg = s_algorithms[i].algID;
            algList->algProperties[algList->count].algProperties =
                s_algorithms[i].attributes;
            algList->count++;
        }
        else
        {
            // If the return list is full but we still have algorithms
            // available, report this and stop scanning.
            more = YES;
            break;
        }
    }
    return more;
}
```

### 9.2 Bits.c

#### 9.2.1 Introduction

This file contains bit manipulation routines. They operate on bit arrays. The 0th bit in the array is the right-most bit in the 0th octet in the array.
NOTE: If pAssert() is defined, the functions will assert if the indicated bit number is outside of the range of bArray. How the assert is handled is implementation dependent.

### 9.2.2 Includes

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

### 9.2.3 BitIsSet()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>bit is set</td>
</tr>
<tr>
<td>FALSE</td>
<td>bit is not set</td>
</tr>
</tbody>
</table>

```c
BOOL BitIsSet(
    unsigned int bitNum,  // IN: number of the bit in 'bArray'
    BYTE *bArray,        // IN: array containing the bits
    unsigned int arraySize,   // IN: size in bytes of 'bArray'
)
{
    pAssert(arraySize > (bitNum >> 3));
    return ((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
}
```

### 9.2.4 BitSet()

This function will set the indicated bit in bArray.

```c
void BitSet(
    unsigned int bitNum,  // IN: number of the bit in 'bArray'
    BYTE *bArray,        // IN: array containing the bits
    unsigned int arraySize,   // IN: size in bytes of 'bArray'
)
{
    pAssert(arraySize > bitNum/8);
    bArray[bitNum >> 3] |= (1 << (bitNum & 7));
}
```

### 9.2.5 BitClear()

This function will clear the indicated bit in bArray.

```c
void BitClear(
    unsigned int bitNum,  // IN: number of the bit in 'bArray'
    BYTE *bArray,        // IN: array containing the bits
    unsigned int arraySize,   // IN: size in bytes of 'bArray'
)
{
    pAssert(arraySize > bitNum/8);
    bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
}
```
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*.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>more command attributes are available</td>
</tr>
<tr>
<td>NO</td>
<td>no more command attributes are available</td>
</tr>
</tbody>
</table>

TPMI_YES_NO

CommandCapGetCCList(

TPM_CC   commandCode, // IN: start command code
UINT32   count,    // IN: maximum count for number of
          // entries in 'commandList'
TPML_CCA *commandList // OUT: list of TPMA_CC
)
{
    TPMI_YES_NO more = NO;
    UINT32 i;

    // initialize output handle list count
    commandList->count = 0;

    // The maximum count of commands that may be return is MAX_CAP_CC.
    if(count > MAX_CAP_CC) count = MAX_CAP_CC;

    // If the command code is smaller than TPM_CC_FIRST, start from TPM_CC_FIRST
    if(commandCode < TPM_CC_FIRST) commandCode = TPM_CC_FIRST;

    // Collect command attributes
    for(i = commandCode; i <= TPM_CC_LAST; i++)
    {
        if(CommandIsImplemented(i))
        {
            if(commandList->count < count)
            {
                // If the list is not full, add the attributes for this command.
                commandList->commandAttributes[commandList->count] = CommandGetAttribute(i);
                commandList->count++;
            }
        }
        else
        {
            // If the list is full but there are more commands to report,
            // indicate this and return.
        }
    }
```c
38         more = YES;
39         break;
40     }        }
41 }          
42 }
43 return more;
44
45 }

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

```c
#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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_HANDLE</td>
<td>handle type does not match</td>
</tr>
<tr>
<td>TPM_RC_REFERENCE_H0</td>
<td>entity is not present</td>
</tr>
<tr>
<td>TPM_RC_HIERARCHY</td>
<td>entity belongs to a disabled hierarchy</td>
</tr>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>handle is an evict object but there is no space to load it to RAM</td>
</tr>
</tbody>
</table>

```c
TPM_RC
EntityGetLoadStatus(
    TPM_HANDLE      *handle   // IN/OUT: handle of the entity
){
    switch(HandleGetType(*handle))
    {
        // For handles associated with hierarchies, the entity is present
        // only if the associated enable is SET.
        case TPM_HT_PERMANENT:
            switch(*handle)
            {
                case TPM_RH_OWNER:
                    if(!gc.shEnable)
                        return TPM_RC_HIERARCHY;
                    else
                        return TPM_RC_SUCCESS;
                    break;
                case TPM_RH_ENDORSEMENT:
                    if(!gc.ehEnable)
                        return TPM_RC_HIERARCHY;
                    else
                        return TPM_RC_SUCCESS;
                    break;
                case TPM_RH_PLATFORM:
                    if(!g_phEnable)
                        return TPM_RC_HIERARCHY;
                    else
                        return TPM_RC_SUCCESS;
                    break;
            // null handle, PW session handle and lockout
            // handle are always available
```
34     case TPM_RH_NULL:
35         break;
36     case TPM_RS_PW:
37         return TPM_RC_SUCCESS;
38         break;
39     case TPM_RH_LOCKOUT:
40         return TPM_RC_SUCCESS;
41             default:
42             // should never see any other permanent handle here
43             pAssert(FALSE);
44             return TPM_RC_HANDLE;
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(ObjectsIsPresent(*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:
// Any PCR handle that is unmarshaled successfully referenced
// a PCR that is defined.
return TPM_RC_SUCCESS;
break;

default:
    // Any other handle type is a defect in the unmarshaling code.
pAssert(FALSE);
return TPM_RC_HANDLE;
break;
}

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.

UINT16 EntityGetAuthValue(
    TPMI_DH_ENTITY handle,          // IN: handle of entity
    BYTE *auth)                      // OUT: authValue of the entity
{
    TPM2B_AUTH authValue = {0};

    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            switch(handle)
            {
                case TPM_RH_OWNER:
                    // ownerAuth for TPM_RH_OWNER
                    authValue = gp.ownerAuth;
                    break;

                case TPM_RH_ENDORSEMENT:
                    // endorsementAuth for TPM_RH_ENDORSEMENT
                    authValue = gp.endorsementAuth;
                    break;

                case TPM_RH_PLATFORM:
                    // platformAuth for TPM_RH_PLATFORM
                    authValue = gc.platformAuth;
                    break;

                case TPM_RH_LOCKOUT:
                    // lockoutAuth for TPM_RH_LOCKOUT
                    authValue = gp.lockoutAuth;
                    break;

                case TPM_RH_NULL:
                    // nullAuth for TPM_RH_NULL. Return 0 directly here
                    return 0;

                default:
                    // If any other permanent handle is present it is
                    // a code defect.
pAssert(FALSE);
                    break;

            }

            break;

        case TPM_HT_TRANSIENT:
152  // authValue for an object
153  // A persistent object would have been copied into RAM
154  // and would have a 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
174  case TPM_HT_NV_INDEX:
175      // authValue for an NV index
176      {
177          NV_INDEX nvIndex;
178          NvGetIndexInfo(handle, &nvIndex);
179          authValue = nvIndex.authValue;
180      }
181  break;
182
183  case TPM_HT_PCR:
184      // authValue for PCR
185      PCRGetAuthValue(handle, &authValue);
186  break;
187
188  default:
189      // If any other handle type is present here, then there is a defect
190      // in the unmarshaling code.
191      pAssert(FALSE);
192  break;
193
194      // Copy the authValue
195      pAssert(authValue.t.size <= <K>sizeof(authValue.t.buffer));
196      MemoryCopy(auth, authValue.t.buffer, authValue.t.size);
197  return authValue.t.size;
198
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 object. 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  {
TPM_ALG_HASH  hashAlg = TPM_ALG_NULL;

switch(HandleGetType(handle))
{
    case TPM_HT_PERMANENT:
        switch(handle)
        {
            case TPM_RH_OWNER:
                // ownerPolicy for TPM_RH_OWNER
                *authPolicy = gp.ownerPolicy;
                hashAlg = gp.ownerAlg;
                break;
            case TPM_RH_ENDORSEMENT:
                // endorsementPolicy for TPM_RH_ENDORSEMENT
                *authPolicy = gp.endorsementPolicy;
                hashAlg = gp.endorsementAlg;
                break;
            case TPM_RH_PLATFORM:
                // platformPolicy for TPM_RH_PLATFORM
                *authPolicy = gc.platformPolicy;
                hashAlg = gc.platformAlg;
                break;
            default:
                // If any other permanent handle is present it is
                // a code defect.
                pAssert(FALSE);
                break;
        }
        break;
    case TPM_HT_TRANSIENT:
        // authPolicy for an object
        {
            OBJECT *object = ObjectGet(handle);
            *authPolicy = object->publicArea.authPolicy;
            hashAlg = object->publicArea.nameAlg;
        }
        break;
    case TPM_HT_NV_INDEX:
        // authPolicy for a NV index
        {
            NV_INDEX        nvIndex;
            NvGetIndexInfo(handle, &nvIndex);
            *authPolicy = nvIndex.publicArea.authPolicy;
            hashAlg = nvIndex.publicArea.nameAlg;
        }
        break;
    case TPM_HT_PCR:
        // authPolicy for a PCR
        hashAlg = PCRGetAuthPolicy(handle, authPolicy);
        break;
    default:
        // If any other handle type is present it is a code defect.
        pAssert(FALSE);
        break;
}
return hashAlg;

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.
EntityGetName

```c
EntityGetName(  
    TPMI_DH_ENTITY handle, // IN: handle of entity  
    BYTE *name    // OUT: name of entity  
)
```

```c
switch(HandleGetType(handle))
{
    case TPM_HT_TRANSIENT:
        // Name for an object
        return ObjectGetName(handle, name);
    break;
    case TPM_HT_NV_INDEX:
        // Name for a NV index
        return NvGetName(handle, name);
    break;
    default:
        // For all other types, the handle is the Name
        return TPM_HANDLE_Marshal(&handle, &name, NULL);
    break;
}
```

**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.

```c
EntityGetHierarchy(  
    TPMI_DH_ENTITY handle // IN :handle of entity  
)
```

```c
switch(HandleGetType(handle))
{
    case TPM_HT_PERMANENT:
        // hierarchy for a permanent handle
        switch(handle)
        {
            case TPM_RH_PLATFORM:
                return handle;
            break;
            case TPM_RH_ENDORSEMENT:
                return handle;
            break;
            case TPM_RH_NULL:
                return handle;
            break;
            // all other permanent handles are associated with the owner
            default:
                return TPM_RH_OWNER;
            break;
        }
    break;
    case TPM_HT_NV_INDEX:
        // hierarchy for NV index
        {
            NV_INDEX nvIndex;
            NvGetIndexInfo(handle, &nvIndex);
            // If only the platform can delete the index, then it is
            // considered to be in the platform hierarchy, otherwise it
            // is in the owner hierarchy.
            ```c
```
if (nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
    return TPM_RH_PLATFORM;
else
    return TPM_RH_OWNER;

break;
case TPM_HT_TRANSIENT:
    // hierarchy for an object
    {
        OBJECT *object;
        object = ObjectGet(handle);
        if (object->attributes.ppsHierarchy)
            return TPM_RH_PLATFORM;
        else if (object->attributes.epsHierarchy)
            return TPM_RH_ENDORSEMENT;
        else if (object->attributes.spsHierarchy)
            return TPM_RH_OWNER;
        else
            return TPM_RH_NULL;
    }
break;
case TPM_HT_PCR:
    return TPM_RH_OWNER;
break;
default:
    pAssert(0);
    break;
}
// this is unreachable but it provides a return value for the default
// case which makes the compiler happy
return TPM_RH_NULL;

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

#define GLOBAL_C
#include "InternalRoutines.h"

9.7.3 Global Data Values

These values are visible across multiple modules.

BOOL           g_phEnable;
const UINT16    g_rcIndex[15] = {TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,
                                TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,
                                TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,
9.7.4 Private Values

9.7.4.1 SessionProcess.c

777777777777777

9.7.4.2 DA.c

9.7.4.3 NV.c

9.7.4.4 Object.c

9.7.4.5 PCR.c
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

    11  TPMI_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
family "02" level 00 revision 00.96

9.9 Locality.c

9.9.1 Includes

#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.

```c
#include "InternalRoutines.h"

// IN: locality value

TPMA_LOCALITY LocalityGetAttributes(
    UINT8 locality
)
{
    TPMA_LOCALITY locality_attributes;
    BYTE *localityAsByte = (BYTE *)&locality_attributes;
    MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
    switch(locality)
    {
        case 0:
            locality_attributes.TPM_LOC_ZERO = SET;
            break;
        case 1:
            locality_attributes.TPM_LOC_ONE = SET;
            break;
    }
    return more;
}
```
```c
    case 2:
        locality_attributes.TPM_LOC_TWO = SET;
        break;
    case 3:
        locality_attributes.TPM_LOC_THREE = SET;
        break;
    case 4:
        locality_attributes.TPM_LOC_FOUR = SET;
        break;
    default:
        pAssert(locality < 256 && locality > 31);
        *localityAsByte = locality;
        break;
    }
    return locality_attributes;
```

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

```c
#define MANUFACTURE_C
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>1</td>
<td>manufacturing process previously performed</td>
</tr>
</tbody>
</table>

```c
int TPM_Manufacture(void)
{
    TPM_SU orderlyShutdown;
    UINT64 totalResetCount = 0;
    // If TPM has been manufactured, return indication.
    if(s_manufactured)
        Return 1;
    // initialize crypto units
    CryptInitUnits();
    // initialize NV
    NvInit();
    // default configuration for PCR
```
PCRSimStart();

// initialize pre-installed hierarchy data
// This should happen after NV is initialized because hierarchy data is
// stored in NV.
HierarchyPreInstall_Init();

// initialize dictionary attack parameters
DAPreInstall_Init();

// initialize PP list
PhysicalPresencePreInstall_Init();

// initialize command audit list
CommandAuditPreInstall_Init();

// first start up is required to be Startup(CLEAR)
orderlyShutdown = TPM_SU_CLEAR;
NvWriteReserved(NV_ORDERLY, &orderlyShutdown);

// initialize the firmware version
#if defined FIRMWARE_V2
gp.firmwareV2 = FIRMWARE_V2;
#else
gp.firmwareV2 = 0;
#endif
NvWriteReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
NvWriteReserved(NV_FIRMWARE_V2, &gp.firmwareV2);

// initialize the total reset counter to 0
NvWriteReserved(NV_TOTAL_RESET_COUNT, &totalResetCount);

// Commit NV writes. Manufacture process is an artificial process existing
// only in simulator environment and it is not defined in the specification
// that what should be the expected behavior if the NV write fails at this
// point. Therefore, it is assumed the NV write here is always success and
// no return code of this function is checked.
NvCommit();
s_manufactured = TRUE;

return 0;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>1</td>
<td>TPM not previously manufactured</td>
</tr>
</tbody>
</table>

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

if(!s_manufactured)
    return 1;

// stop crypt units
CryptStopUnits();
s_manufactured = FALSE;
return 0;

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>
<thead>
<tr>
<th>Values</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>{TRANSIENT_FIRST:TRANSIENT_LAST}</td>
<td>allowed range for transient objects</td>
</tr>
<tr>
<td>{PERSISTENT_FIRST:PERSISTENT_LAST}</td>
<td>allowed range for persistent objects</td>
</tr>
<tr>
<td>+TPM_RH_NULL</td>
<td>the null handle</td>
</tr>
<tr>
<td>#TPM_RC_VALUE</td>
<td></td>
</tr>
</tbody>
</table>

This generates the following unmarshaling code:

```
TPM_RC
TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
        bool flag)
{
    TPM_RC    result;
    result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    if (*target == TPM_RH_NULL) {
        if(flag)
            return TPM_RC_SUCCESS;
        else
            return TPM_RC_VALUE;
    }
    if(*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
        return TPM_RC_VALUE;
```
and the following marshaling code:

```
NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```UINT16
TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
{
    return UINT32_Marshal((UINT32 *)source, buffer, size);
}
```

### 9.11.3 Unmarshal and Marshal a Union

In part 2, a TPMU_PUBLIC_PARMS union is defined by:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Selector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyedHash</td>
<td>TPMS_KEYEDHASH_PARMS</td>
<td>TPM_ALG_KEYEDHASH</td>
<td>sign</td>
</tr>
<tr>
<td>symDetail</td>
<td>TPMT_SYM_DEF_OBJECT</td>
<td>TPM_ALG_SYMCIPHER</td>
<td>a symmetric block cipher</td>
</tr>
<tr>
<td>rsaDetail</td>
<td>TPMS_RSA_PARMS</td>
<td>TPM_ALG_RSA</td>
<td>decrypt + sign</td>
</tr>
<tr>
<td>eccDetail</td>
<td>TPMS_ECC_PARMS</td>
<td>TPM_ALG_ECC</td>
<td>decrypt + sign</td>
</tr>
<tr>
<td>asymDetail</td>
<td>TPMSASYM_PARMS</td>
<td></td>
<td>common scheme structure</td>
</tr>
</tbody>
</table>

NOTE The Description column indicates which of TPMA_OBJECT.decrypt or TPMA_OBJECT.sign may be set. 
"+" indicates that both may be set but one shall be set. "|" indicates the optional settings.

From this table, the following unmarshaling code is generated.

```
TPM_RC
TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size, 
UINT32 selector)
{
    switch(selector) {
    #ifdef TPM_ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Unmarshal(
                (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
    #endif
    #ifdef TPM_ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Unmarshal( 
                (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
    #endif
    #ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Unmarshal(
                (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
    #endif
    #ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Unmarshal(
                (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
    #endif
    }
    return TPM_RC_SELECTOR;
}
```
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:

```c
UINT16
TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
UINT32 selector)
{
  switch(selector) {
  #ifdef TPM_ALG_KEYEDHASH
    case TPM_ALG_KEYEDHASH:
      return TPMS_KEYEDHASH_PARMS_Marshal(
        (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
  #endif
  #ifdef TPM_ALG_SYMCIPHER
    case TPM_ALG_SYMCIPHER:
      return TPMT_SYM_DEF_OBJECT_Marshal(
        (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
  #endif
  #ifdef TPM_ALG_RSA
    case TPM_ALG_RSA:
      return TPMS_RSA_PARMS_Marshal(
        (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
  #endif
  #ifdef TPM_ALG_ECC
    case TPM_ALG_ECC:
      return TPMS_ECC_PARMS_Marshal(
        (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
  #endif
  }
  assert(1);
  return 0;
}
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for selector. The 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>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>TPMI_ALG_PUBLIC</td>
<td>&quot;algorithm&quot; associated with this object</td>
</tr>
<tr>
<td>nameAlg</td>
<td>+TPMI_ALG_HASH</td>
<td>algorithm used for computing the Name of the object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOTE: The &quot;+&quot; indicates that the instance of a TPMT_PUBLIC may have a &quot;+&quot; to indicate that the nameAlg may be TPM_ALG_NULL.</td>
</tr>
<tr>
<td>objectAttributes</td>
<td>TPMA_OBJECT</td>
<td>attributes that, along with type, determine the manipulations of this object</td>
</tr>
<tr>
<td>authPolicy</td>
<td>TPM2B_DIGEST</td>
<td>optional policy for using this key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The policy is computed using the nameAlg of the object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOTE: shall be the Empty Buffer if no authorization policy is present</td>
</tr>
<tr>
<td>[type]parameters</td>
<td>TPMU_PUBLIC_PARMS</td>
<td>the algorithm or structure details</td>
</tr>
<tr>
<td>[type]unique</td>
<td>TPMU_PUBLIC_ID</td>
<td>the unique identifier of the structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For an asymmetric key, this would be the public key.</td>
</tr>
</tbody>
</table>

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```c
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      result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
22                                                   buffer, size, (UINT32)target->type);
23      if(result != TPM_RC_SUCCESS)
24          return result;
25      result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
26                                                   buffer, size, (UINT32)target->type);
27      if(result != TPM_RC_SUCCESS)
28          return result;
29     return TPM_RC_SUCCESS;
30  }
```
The marshaling code for the TPMT_PUBLIC structure is:

```c
UINT16
TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
{
    UINT16 result = 0;
    result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
        (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
    result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
        (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size));
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
        (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
    result = (UINT16)(result + TPM2B_DIGEST_Marshal(
        (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));
    result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
        (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
        (UINT32)source->type));
    result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
        (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
        (UINT32)source->type));

    return result;
}
```
9.11.5 Unmarshal and Marshal an Array

In part 2, the TPML_DIGEST is defined by:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count {2:}</td>
<td>UINT32</td>
<td>number of digests in the list, minimum is two</td>
</tr>
<tr>
<td>digests[count]{:8}</td>
<td>TPM2B_DIGEST</td>
<td>a list of digests</td>
</tr>
<tr>
<td>#TPM_RC_SIZE</td>
<td></td>
<td>response code when count is not at least two or is greater than 8</td>
</tr>
</tbody>
</table>

The digests parameter is an array of up to count structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```c
TPM_RC
TPM_DIGEST_Unmarshal(TPM_DIGEST *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    if( (target->count < 2)) // This check is triggered by the {2:} notation
        return TPM_RC_SIZE; // on 'count'

    if((target->count) > 8) // This check is triggered by the {:8} notation
        return TPM_RC_SIZE; // on 'digests'.

    result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
                                           buffer, size, (INT32)(target->count));
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```

The routine unmarshals a count value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```c
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}
```
Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
UINT16
TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
{
    UINT16 result = 0;
    result = (UINT16)(result + UINT32.Marshal((UINT32 *)&(source->count), buffer, size));
    result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
        (TPM2B_DIGEST *)&(source->digests), buffer, size,
        (INT32)(source->count)));
    return result;
}
```

The marshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
INT32 count)
{
    TPM_RC result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST.Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}
```
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>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>UINT16</td>
<td>Size of the operand</td>
</tr>
<tr>
<td>buffer [size]</td>
<td>BYTE</td>
<td>The operand</td>
</tr>
</tbody>
</table>

Table xxx — Definition of TPM2B_EVENT Structure

```c
1 TPM_RC
2 TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3 {
4     TPM_RC result;
5     result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6     if(result != TPM_RC_SUCCESS)
7         return result;
8     // if size equal to 0, the rest of the structure is a zero buffer. Stop processing
9     if((target->t.size == 0)
10        return TPM_RC_SUCCESS;
11     if((target->t.size) > 1024) // This check is triggered by the {:1024} notation on 'buffer'
12        return TPM_RC_SIZE;
13     result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
14         (INT32)(target->t.size));
15     if(result != TPM_RC_SUCCESS)
16         return result;
17     return TPM_RC_SUCCESS;
18 }
```

Which use these structure definitions:

```c
typedef struct {
    UINT16 size;
    BYTE buffer[1];
} TPM2B;

typedef struct {
    UINT16 size;
    BYTE buffer[1024];
} EVENT_2B;

typedef union {
    EVENT_2B t; // The type-specific union member
    TPM2B b;   // The generic union member
} TPM2B_EVENT;
```
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

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

These buffers are set aside to hold command and response values. In this implementation, it is not guaranteed that the code will stop accessing the s_actionInputBuffer before starting to put values in the s_actionOutputBuffer so different buffers are required. However, the s_actionInputBuffer and s_responseBuffer are not needed at the same time and they could be the same buffer.

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.

```c
static UINT32 s_actionInputBuffer[1024]; // action input buffer
static UINT32 s_actionOutputBuffer[1024]; // action output buffer
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.

```c
void MemoryCopy(
    void *destination, // OUT: copy destination
    void *source,     // IN: copy source
    UINT32 size       // IN: number of octets being copied
)
{
    BYTE *p = (BYTE *)source;
    BYTE *q = (BYTE *)destination;
    while (size--)
        *q++ = *p++;
    return;
}
```

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.

```c
void MemoryMove(
    void *destination, // OUT: move destination
    const void *source, // IN: move source
    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 source, then moving bytes in ascending order is safe.
28     if (p>q || (p+size <= q))
29     {
30         while (size--)
31             *q++ = *p++;
32     }
33     // If the destination buffer has a higher address than the source, then move bytes from the end to the beginning.
34     else if (p<q)
35     {
36         p += size;
37         q += size;
38         while (size--)
39             *--q = *--p;
40     }
41     // If the source and destination address are the same, nothing to move.
42     return;
43 }

9.12.3.3 MemoryEqual()

This function indicates if two buffers have the same values in the indicated number of bytes.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>all octets are the same</td>
</tr>
<tr>
<td>FALSE</td>
<td>all octets are not the same</td>
</tr>
</tbody>
</table>

46 BOOL MemoryEqual(
47     const void *buffer1, // IN: compare buffer1
48     const void *buffer2, // IN: compare buffer2
49     UINT32 size        // IN: size of bytes being compared
50 )
51 {
52     BOOL equal = TRUE;
53     const BYTE *b1, *b2;
54     b1 = (BYTE *)buffer1;
55     b2 = (BYTE *)buffer2;
56     // Compare all bytes so that there is no leakage of information
57     // due to timing differences.
58     for(; size > 0; size--)
59         equal = (*b1++ == *b2++) & equal;
60     return equal;
61 }

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.
9.12.3.5 MemoryConcat2B()

This function will concatenate the buffer contents of a TPM2B to another TPM2B and adjust the size accordingly ($a := (a | b)$).

```c
void MemoryConcat2B(
    TPM2B *aInOut, // IN/OUT: destination 2B
    TPM2B *bIn     // IN: second 2B
)
{
    MemoryCopy(&aInOut->buffer[aInOut->size], bIn->buffer, bIn->size);
    aInOut->size = aInOut->size + bIn->size;
    return;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>size and buffer contents are the same</td>
</tr>
<tr>
<td>FALSE</td>
<td>size or buffer contents are not the same</td>
</tr>
</tbody>
</table>

```c
BOOL Memory2BEqual(
    const TPM2B *aIn, // IN: compare value
    const TPM2B *bIn  // IN: compare value
)
{
    if(aIn->size != bIn->size)
        return FALSE;
    return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
}
```

9.12.3.7 MemorySet()

This function will set all the octets in the specified memory range to the specified octet value.

```c
void MemorySet(
    void   *destination, // OUT: memory destination
    char   value,        // IN: fill value
    UINT32 size          // IN: number of octets to fill
)
{
char *p = destination;
while (size--)
    *p++ = value;
return;
}

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.

BYTE *
MemoryGetActionInputBuffer(
    UINT32 size // Size, in bytes, required for the input unmarshaling
)
{
    // In this implementation, a static buffer is set aside for action output.
    // Other implementations may apply additional optimization based on command
    // code or other factors.
    UINT32 *p = s_actionInputBuffer;
    if (size == 0)
        return NULL;
    pAssert(size < <K>sizeof(s_actionInputBuffer));
    #define SZ sizeof(s_actionInputBuffer[0])
    for (size = (size + SZ - 1) / SZ; size > 0; size--)
        *p++ = 0;
    #undef SZ
    return (BYTE *)s_actionInputBuffer;
}

9.12.3.9 MemoryGetActionOutputBuffer()

This function returns the address of the buffer into which the command action code places its output values.

void *
MemoryGetActionOutputBuffer(
    TPM_CC command // Command that requires the buffer
)
{
    // In this implementation, a static buffer is set aside for action output.
    // Other implementations may apply additional optimization based on the command
    // code or other factors.
    command = 0; // Unreferenced parameter
    return s_actionOutputBuffer;
}

9.12.3.10 MemoryGetResponseBuffer()

This function returns the address into which the command response is marshaled from values in the action output buffer.

BYTE *
MemoryGetResponseBuffer(
    TPM_CC command // Command that requires the buffer
)
{
    // In this implementation, a static buffer is set aside for responses.
    // Other implementation may apply additional optimization based on the command

// code or other factors.
command = 0;       // Unreferenced parameter
return  s_responseBuffer;

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.

UINT16 MemoryRemoveTrailingZeros ( 
    TPM2B_AUTH  *auth       // IN/OUT: value to adjust
 )
{
    BYTE        *a = &auth->t.buffer[auth->t.size-1];
    for(; auth->t.size > 0; auth->t.size--)
    {
        if(*a--)
            break;
    }
    return auth->t.size;
}

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

#include "InternalRoutines.h"

9.13.3 Functions

9.13.3.1 TPMInit()

This function is used to process a power on event.

void
TPMInit(void)
{
    // Set state as not initialized
    s_initialized = FALSE;
    return;
}

9.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2_Startup() has completed
successfully).

void
9.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2_Startup() has completed successfully after a _TPM_Init()).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TPM has been initialized</td>
</tr>
<tr>
<td>FALSE</td>
<td>TPM has not been initialized</td>
</tr>
</tbody>
</table>

18
19
20
21
22

9.14 PropertyCap.c

9.14.1 Description

This file contains the functions that are used for accessing the TPM_CAP_TPMPROPERTY values.

9.14.2 Includes

#include "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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>referenced property exists and value set</td>
</tr>
<tr>
<td>FALSE</td>
<td>referenced property does not exist</td>
</tr>
</tbody>
</table>

2
3
4
5
6
7
8
9
10
11

static BOOL
TPMPropertyIsDefined(
    TPM_PT property,       // IN: property
    UINT32 *value          // OUT: property value
)

{ switch(property)
    { case TPM_PT_FAMILY_INDICATOR:
        // from the title page of the specification
        // only used to indicate the TPM is initialized.
// For this specification, the value is "2.0".
*value = TPM_SPEC_FAMILY;
break;

case TPM_PT_LEVEL:
  // from the title page of the specification
  *value = TPM_SPEC_LEVEL;
  break;

case TPM_PT_REVISION:
  // from the title page of the specification
  *value = TPM_SPEC_VERSION;
  break;

case TPM_PT_DAY_OF_YEAR:
  // computed from the date value on the title page of the specification
  *value = TPM_SPEC_DAY_OF_YEAR;
  break;

case TPM_PT_YEAR:
  // from the title page of the specification
  *value = TPM_SPEC_YEAR;
  break;

case TPM_PT_MANUFACTURER:
  // vendor ID unique to each TPM manufacturer
  *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
  break;

case TPM_PT_VENDOR_STRING_1:
  // first four characters of the vendor ID string
  *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
  break;

case TPM_PT_VENDOR_STRING_2:
  // second four characters of the vendor ID string
  ifdef VENDOR_STRING_2
    *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
  else
    *value = 0;
  endif
  break;

case TPM_PT_VENDOR_STRING_3:
  // third four characters of the vendor ID string
  ifdef VENDOR_STRING_3
    *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
  else
    *value = 0;
  endif
  break;

case TPM_PT_VENDOR_STRING_4:
  // fourth four characters of the vendor ID string
  ifdef VENDOR_STRING_4
    *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
  else
    *value = 0;
  endif
  break;

case TPM_PT_VENDOR_TPM_TYPE:
  // vendor-defined value indicating the TPM model
  *value = 1;
  break;

case TPM_PT_FIRMWARE_VERSION_1:
  // more significant 32-bits of a vendor-specific value
  *value = gp.firmwareV1;
  break;

case TPM_PT_FIRMWARE_VERSION_2:
  // less significant 32-bits of a vendor-specific value
  *value = gp.firmwareV2;
  break;

case TPM_PT_INPUT_BUFFER:
  // maximum size of TPM2B_MAX_BUFFER
  *value = MAX_DIGEST_BUFFER;
break;

case TPM_PT_HR_TRANSIENT_MIN:
    // minimum number of transient objects that can be held in TPM
    // RAM
    *value = MAX_LOADED_OBJECTS;
    break;

case TPM_PT_HR_PERSISTENT_MIN:
    // minimum number of persistent objects that can be held in
    // TPM NV memory
    // In this implementation, there is no minimum number of
    // persistent objects.
    *value = MIN_EVICT_OBJECTS;
    break;

case TPM_PT_HR_LOADED_MIN:
    // minimum number of authorization sessions that can be held in
    // TPM RAM
    *value = MAX_LOADED_SESSIONS;
    break;

case TPM_PT_ACTIVE_SESSIONS_MAX:
    // number of authorization sessions that may be active at a time
    *value = MAX_ACTIVE_SESSIONS;
    break;

case TPM_PT_PCR_COUNT:
    // number of PCR implemented
    *value = IMPLEMENTATION_PCR;
    break;

case TPM_PT_PCR_SELECT_MIN:
    // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
    *value = PCR_SELECT_MIN;
    break;

case TPM_PT_CONTEXT_GAP_MAX:
    // maximum allowed difference (unsigned) between the contextID
    // values of two saved session contexts
    *value = (1 << (K*sizeof(CONTEXT_SLOT) * 8)) - 1;
    break;

case TPM_PT_NV_COUNTERS_MAX:
    // maximum number of NV indexes that are allowed to have the
    // TPM_NV_COUNTER attribute SET
    // In this implementation, there is no limitation on the number
    // of counters, except for the size of the NV Index memory.
    *value = 0;
    break;

case TPM_PT_NV_INDEX_MAX:
    // maximum size of an NV index data area
    *value = MAX_NV_INDEX_SIZE;
    break;

case TPM_PT_MEMORY:
    // a TPMA_MEMORY indicating the memory management method for the TPM
    {
        TPMA_MEMORY attributes = {0};
        attributes.sharedNV = SET;
        attributes.objectCopiedToRam = SET;

        // Note: Different compilers may require a different method to cast
        // a bit field structure to a UINT32.
        *value = * (UINT32 *) &attributes;
        break;
    }

case TPM_PT CLOCK UPDATE:
    // interval, in seconds, between updates to the copy of
    // TPMS TIME_INFO .clock in NV
    *value = (1 << NV_CLOCK UPDATE INTERVAL);
    break;

case TPM_PT_CONTEXT_HASH:
    // algorithm used for the integrity hash on saved contexts and
    // for digesting the fUData of TPM2_FirmwareRead()
*value = CONTEXT_INTEGRITY_HASH_ALG;
break;

case TPM_PT_CONTEXT_SYM:
    // algorithm used for encryption of saved contexts
    *value = CONTEXT_ENCRYPT_ALG;
    break;

case TPM_PTCONTEXT_SYM_SIZE:
    // size of the key used for encryption of saved contexts
    *value = CONTEXT_ENCRYPT_KEY_BITS;
    break;

case TPM_PT_ORDERLY_COUNT:
    // maximum difference between the volatile and non-volatile
    // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY_SET
    *value = MAX_ORDERLY_COUNT;
    break;

case TPM_PT_MAX_COMMAND_SIZE:
    // maximum value for 'commandSize'
    *value = TPM_MAX_COMMAND_SIZE;
    break;

case TPM_PT_MAX_RESPONSE_SIZE:
    // maximum value for 'responseSize'
    *value = TPM_MAX_RESPONSE_SIZE;
    break;

case TPM_PT_MAX_DIGEST:
    // maximum size of a digest that can be produced by the TPM
    *value = sizeof(TPMU_HA);
    break;

case TPM_PT_MAX_OBJECT_CONTEXT:
    // maximum size of a TPMS_CONTEXT that will be returned by
    // TPM2_ContextSave for object context
    *value = 0;
    // adding sequence, saved handle and hierarchy
    *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
              sizeof(TPMI_RH_HIERARCHY);
    // add size field in TPM2B CONTEXT
    *value += sizeof(UINT16);
    // add integrity hash size
    *value += sizeof(UINT16) +
              CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
    // Add fingerprint size, which is the same as sequence size
    *value += sizeof(UINT64);
    // Add OBJECT structure size
    *value += sizeof(OBJECT);
    break;

case TPM_PT_MAX_SESSION_CONTEXT:
    // the maximum size of a TPMS_CONTEXT that will be returned by
    // TPM2_ContextSave for object context
    *value = 0;
    // adding sequence, saved handle and hierarchy
    *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
              sizeof(TPMI_RH_HIERARCHY);
    // Add size field in TPM2B CONTEXT
    *value += sizeof(UINT16);
    // Add integrity hash size
    *value += sizeof(UINT16) +
              CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
    // Add fingerprint size, which is the same as sequence size
    *value += sizeof(UINT64);
    // Add SESSION structure size
*value += sizeof(SESSION);

break;

case TPM_PT_PS_FAMILY_INDICATOR:
    // platform specific values for the TPM_PT_PS parameters from
    // the relevant platform-specific specification
    // In this reference implementation, all of these values are 0.
    *value = 0;
    break;

case TPM_PT_PS_LEVEL:
    // level of the platform-specific specification
    *value = 0;
    break;

case TPM_PT_PS_REVISION:
    // specification Revision times 100 for the platform-specific
    // specification
    *value = 0;
    break;

case TPM_PT_PS_DAY_OF_YEAR:
    // platform-specific specification day of year using TCG calendar
    *value = 0;
    break;

case TPM_PT_PS_YEAR:
    // platform-specific specification year using the CE
    *value = 0;
    break;

case TPM_PT_SPLIT_MAX:
    // number of split signing operations supported by the TPM
    *value = 0;
    break;

#ifdef TPM_ALG_ECDAA
    *value = sizeof(gr.commitArray) * 8;
#endif
    break;

case TPM_PT_TOTAL_COMMANDS:
    // total number of commands implemented in the TPM
    // Since the reference implementation does not have any
    // vendor-defined commands, this will be the same as the
    // number of library commands.
    {
        UINT32 i;
        *value = 0;

        // calculate implemented command numbers
        for (i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
        {
            if (CommandIsImplemented(i)) (*value)++;
        }
        break;
    }

case TPM_PT_LIBRARY_COMMANDS:
    // number of commands from the TPM library that are implemented
    {
        UINT32 i;
        *value = 0;

        // calculate implemented command numbers
        for (i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
        {
            if (CommandIsImplemented(i)) (*value)++;
        }
        break;
    }

case TPM_PT_VENDOR_COMMANDS:
    // number of vendor commands that are implemented
    *value = 0;
    break;

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
298        case TPM_PT_STARTUP_CLEAR:
299            // TPMA_STARTUP_CLEAR
300        {
301            TPMA_STARTUP_CLEAR  flags = {0};
302            if(g_phEnable)
303                flags.phEnable = SET;
304            if(gc.shEnable)
305                flags.shEnable = SET;
306            if(gc.ehEnable)
307                flags.ehEnable = SET;
308            if(g_prevOrderlyState != SHUTDOWN_NONE)
309                flags.orderly = SET;
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
316        case TPM_PT_HR_NV_INDEX:
317            // number of NV indexes currently defined
318            *value = NvCapGetIndexNumber();
319            break;
320        case TPM_PT_HR_LOADED:
321            // number of authorization sessions currently loaded into TPM
322            // RAM
323            *value = SessionCapGetLoadedNumber();
324            break;
325        case TPM_PT_HR_LOADED_AVAIL:
326            // number of additional authorization sessions, of any type,
327            // that could be loaded into TPM RAM
328            *value = SessionCapGetLoadedAvail();
329            break;
330        case TPM_PT_HR_ACTIVE:
331            // number of active authorization sessions currently being
332            // tracked by the TPM
333            *value = SessionCapGetActiveNumber();
334            break;
335        case TPM_PT_HR_ACTIVE_AVAIL:
336            // number of additional authorization sessions, of any type,
337            // that could be created
338            *value = SessionCapGetActiveAvail();
339            break;
340        case TPM_PT_HR_TRANSIENT_AVAIL:
341            // estimate of the number of additional transient objects that
342            // could be loaded into TPM RAM
*value = ObjectCapGetTransientAvail();
break;

case TPM_PT_HR_PERSISTENT:
  // number of persistent objects currently loaded into TPM
  // NV memory
  *value = NvCapGetPersistentNumber();
  break;

case TPM_PT_HR_PERSISTENT_AVAIL:
  // number of additional persistent objects that could be loaded
  // into NV memory
  *value = NvCapGetPersistentAvail();
  break;

case TPM_PT_NV_COUNTERS:
  // number of defined NV indexes that have NV TPMA_NV_COUNTER
  // attribute SET
  *value = NvCapGetCounterNumber();
  break;

case TPM_PT_NV_COUNTERS_AVAIL:
  // number of additional NV indexes that can be defined with their
  // TPMA_NV_COUNTER attribute SET
  *value = NvCapGetCounterAvail();
  break;

case TPM_PT_ALGORITHM_SET:
  // region code for the TPM
  *value = gp.algorithmSet;
  break;

case TPM_PT_LOADED_CURVES:
  #ifdef TPM_ALG_ECC
    // number of loaded ECC curves
    *value = CryptCapGetEccCurveNumber();
  #else // TPM_ALG_ECC
    *value = 0;
  #endif // TPM_ALG_ECC
  break;

case TPM_PT_LOCKOUT_COUNTER:
  // current value of the lockout counter
  *value = gp.failedTries;
  break;

case TPM_PT_MAX_AUTH_FAIL:
  // number of authorization failures before DA lockout is invoked
  *value = gp.maxTries;
  break;

case TPM_PT_LOCKOUT_INTERVAL:
  // number of seconds before the value reported by
  // TPM_PT_LOCKOUT_COUNTER is decremented
  *value = gp.recoveryTime;
  break;

case TPM_PT_LOCKOUT_RECOVERY:
  // number of seconds after a lockoutAuth failure before use of
  // lockoutAuth may be attempted again
  *value = gp.lockoutRecovery;
  break;

case TPM_PT_AUDIT_COUNTER_0:
  // high-order 32 bits of the command audit counter
  *value = (UINT32) (gp.auditCounter >> 32);
  break;

case TPM_PT_AUDIT_COUNTER_1:
  // low-order 32 bits of the command audit counter
  *value = (UINT32) (gp.auditCounter);
  break;
default:
  // property is not defined
  return FALSE;
  break;
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>more properties are available</td>
</tr>
<tr>
<td>NO</td>
<td>no more properties to be reported</td>
</tr>
</tbody>
</table>

TPM CapGetProperties(

TPM_PT property, // IN: the starting TPM property
UINT32 count, // IN: maximum number of returned properties
TPML_TAGGED_TPM_PROPERTY *propertyList // OUT: property list
)

TPMI_YES_NO more = NO;
UINT32 i;

// initialize output property list
propertyList->count = 0;

// maximum count of properties we may return is MAX_PCR_PROPERTIES
if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;

// If property is less than PT_FIXED, start from PT_FIXED.
if(property < PT_FIXED) property = PT_FIXED;

// Scan through the TPM properties of the requested group.
// The size of TPM property group is PT_GROUP * 2 for fix and variable groups.
for(i = property; i <= PT_FIXED + PT_GROUP * 2; i++)

  UINT32 value;
  if(TPMPropertyIsDefined((TPM_PT) i, &value))
    {
      if(propertyList->count < count)
        {
          // If the list is not full, add this property
          propertyList->tpmProperty[propertyList->count].property = (TPM_PT) i;
          propertyList->tpmProperty[propertyList->count].value = value;
          propertyList->count++;
        }
        else
          {
            // If the return list is full but there are more properties
            // available, set the indication and exit the loop.
            more = YES;
            break;
          }
    }
  }

return more;
})
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

```c
#include "TPM_Types.h" // types shared by CryptUtil and CryptoEngine.
#include "CryptPri.h" // Includes the function prototypes for the
                    // CryptoEngine functions.
#include "Global.h"
#include "CryptUtil_fp.h" // Declared here.
#include "InternalRoutines.h"
#include "MemoryLib_fp.h"
```

9.16.3 TranslateCryptErrors()

This function converts errors from the cryptographic library into TPM_RC_VALUES.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_VALUE</td>
<td>CRYPT_FAIL</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>CRYPT_NO_RESULT</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>CRYPT_SCHEME</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>CRYPT_PARAMETER</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>CRYPT_UNDERFLOW</td>
</tr>
<tr>
<td>TPM_RC_ECC_POINT</td>
<td>CRYPT_POINT</td>
</tr>
<tr>
<td>TPM_RC_CANCELLED</td>
<td>CRYPT_CANCEL</td>
</tr>
</tbody>
</table>

```c
static TPM_RC TranslateCryptErrors ( 
    CRYPT_RESULT         retVal // IN: crypt error to evaluate
) {
    switch (retVal) {
        case CRYPT_SUCCESS:      return TPM_RC_SUCCESS;
        case CRYPT_FAIL:         return TPM_RC_VALUE;
        case CRYPT_NO_RESULT:    return TPM_RC_NO_RESULT;
        case CRYPT_SCHEME:       return TPM_RC_SCHEME;
        case CRYPT_PARAMETER:    return TPM_RC_PARAMETER;
    }
}
```
Trusted Platform Module Library

Part 4: Supporting Routines

```c
return TPM_RC_VALUE;
case CRYPTO_UNDERFLOW:
    return TPM_RC_SIZE;
case CRYPTO_POINT:
    return TPM_RC_ECC_POINT;
case CRYPTO_CANCEL:
    return TPM_RC_CANCELED;
default: // Other unknown warnings
    return TPM_RC_FAILURE;
```

9.16.4 Random Number Generation Functions

9.16.4.1 CryptStirRandom()

Stir random entropy

```c
#ifdef TPM_ALG_NULL

void CryptStirRandom(
    UINT32 entropySize, // IN: size of entropy buffer
    BYTE buffer        // IN: entropy buffer
)
{
    // RNG self testing code may be inserted here
    // Call crypto engine random number stirring function
    _cpri__StirRandom(entropySize, buffer);
    return;
}
#endif
```

9.16.4.2 CryptGenerateRandom()

This is the interface to _cpri__GenerateRandom().

```c
UINT16 CryptGenerateRandom(
    UINT16 randomSize, // IN: size of random number
    BYTE buffer        // OUT: buffer of random number
)
{
    // Call crypto engine random number generation
    return _cpri__GenerateRandom(randomSize, buffer);
}
#endif
```

9.16.5 Hash/HMAC Functions

9.16.5.1 CryptGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```c
#ifdef TPM_ALG_KEYEDHASH

TPM_ALG_ID CryptGetContextAlg(
    void *state      // IN: the context to check
)
```

Family "02" Published Page 245
Level 00 Revision 00.96 Copyright © TCG 2006-2013 March 15, 2013
9.16.5.2 CryptStartHash()

This function starts a hash and return the size, in bytes, of the digest.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>the digest size of the algorithm</td>
</tr>
<tr>
<td>= 0</td>
<td>the hashAlg was TPM_ALG_NULL</td>
</tr>
</tbody>
</table>

```c
UINT16 CryptStartHash(
    TPM_ALG_HASH hashAlg,        // IN: hash algorithm  
    HASH_STATE *hashState       // OUT: the state of hash stack. It  
                      // will be used in hash update                         
                      // and completion                                         
) {
    CRYPT_RESULT   retVal;    
    pAssert(hashState != NULL);
    // Set the state type
    hashState->type = HASH_STATE_HASH;
    // Call crypto engine start hash function
    if((retVal = _cpri__StartHash(hashAlg, FALSE, &hashState->state)) == 0)
        hashState->type = HASH_STATE_EMPTY;
    return retVal;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>the digest size of the algorithm</td>
</tr>
<tr>
<td>= 0</td>
<td>the hashAlg was TPM_ALG_NULL</td>
</tr>
</tbody>
</table>

```c
UINT16 CryptStartHashSequence(
    TPM_ALG_HASH hashAlg,        // IN: hash algorithm  
    HASH_STATE *hashState       // OUT: the state of hash stack. It  
                      // will be used in hash update                         
                      // and completion                                         
) {
    CRYPT_RESULT   retVal;    
    pAssert(hashState != NULL);
    // Set the state type
    hashState->type = HASH_STATE_HASH;
    // 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 hashAlg 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 hashAlg 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
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>the digest size of the algorithm</td>
</tr>
<tr>
<td>= 0</td>
<td>the hashAlg was TPM_ALG_NULL</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>the digest size of the algorithm</td>
</tr>
<tr>
<td>= 0</td>
<td>the hashAlg was TPM_ALG_NULL</td>
</tr>
</tbody>
</table>
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      pAssert(digestState != NULL);
183      if(hashState->type == HASH_STATE_EMPTY)
184        return;
185      // If no data, nothing to do (this is not an error)
186      if(data == NULL || dataSize == 0)
187        return;
188      // Call crypto engine update hash function
189      _cpri__UpdateHash(&hashState->state, dataSize, data);
190      return;
191    }

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().
Part 4: Supporting Routines

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     #if BIG_ENDIAN_TPM == YES
217         CryptUpdateHash(state, inSize, (BYTE *)intValue);
218     #else
219         BYTE marshalBuffer[8];
220         // Point to the big end of an little-endian value
221         BYTE *p = &((BYTE *)intValue)[intSize - 1];
222         // Point to the big end of an big-endian value
223         BYTE *q = marshalBuffer;
224         pAssert(intSize <= 8 && intSize != 0 && intValue != NULL);
225         switch (intSize)
226         {
227             case 8:
228                 *q++ = *p--;
229                 *q++ = *p--;
230                 *q++ = *p--;
231                 *q++ = *p--;
232             case 4:
233                 *q++ = *p--;
234                 *q++ = *p--;
235             case 2:
236                 *q++ = *p--;
237             case 1:
238                 *q = *p;
239                 break;
240             default:
241                 pAssert(TRUE);
242                 return;
243         }
244     #endif
245     // Call update the hash
246     CryptUpdateDigest(state, intSize, marshalBuffer);
247     #endif
248     return;
249 }  

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>the number of bytes placed in digest</td>
</tr>
</tbody>
</table>

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 )
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>the number of bytes placed in ‘digest.buffer’</td>
</tr>
</tbody>
</table>

```c
UINT16 CryptCompleteHash2B(
    void *state,       // IN: the state of hash stack
    TPM2B *digest)     // IN: the size of the buffer
{
    if (digest == NULL)
        return 0;
    return CryptCompleteHash(state, digest->size, digest->buffer);
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>the number of bytes placed in ret</td>
</tr>
</tbody>
</table>

```c
UINT16 CryptHashBlock(
    TPM_ALG_ID algId,       // IN: the hash algorithm to use
    UINT16 blockSize,       // IN: size of the data block
    BYTE *block,            // IN: address of the block to hash
    UINT16 retSize,         // IN: size of the return buffer
    BYTE *ret               // OUT: address of the buffer
)
{
    return _cpri___HashBlock(algId, blockSize, block, retSize, ret);
}
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>the number of bytes placed in digest</td>
</tr>
</tbody>
</table>

```c
UINT16 CryptCompleteHMAC(HMAC_STATE *hmacState, // IN: the state of HMAC stack
                        UINT32 digestSize, // IN: size of digest buffer
                        BYTE *digest // OUT: HMAC digest)
{
    HASH_STATE *hashState;
    pAssert(hmacState != NULL);
    hashState = hmacState->hashState;
    if(hashState->type == HASH_STATE_EMPTY)
        return 0;
    pAssert(hashState->type == HASH_STATE_HMAC);
    hashState->type = HASH_STATE_EMPTY;
    return _cpri__CompleteHMAC(&hashState->state, &hmacState->hmacKey.b,
                              digestSize, digest);
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>the number of bytes placed in digest</td>
</tr>
</tbody>
</table>

```c
UINT16 CryptCompleteHMAC2B(HMAC_STATE *hmacState, // IN: the state of HMAC stack
                           TPM2B *digest // OUT: HMAC)
{
    if(digest == NULL)
        return 0;
    return CryptCompleteHMAC(hmacState, digest->size, digest->buffer);
}
```

9.16.5.16 CryptGetHashDigestSize()

This function returns the digest size in bytes for a hash algorithm.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>digest size for TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>digest size</td>
</tr>
</tbody>
</table>

```c
UINT16
CryptGetHashDigestSize(TPM_ALG_ID hashAlg) // IN: hash algorithm
{
    return _cpri__GetDigestSize(hashAlg);
}
```

### 9.16.5.17 CryptGetHashBlockSize()

Get the digest size in byte of a hash algorithm.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>block size for TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>block size</td>
</tr>
</tbody>
</table>

```c
UINT16
CryptGetHashBlockSize(TPM_ALG_ID hash) // IN: hash algorithm to look up
{
    return _cpri__GetHashBlockSize(hash);
}
```

### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_ALG_xxx()</td>
<td>a hash algorithm</td>
</tr>
<tr>
<td>TPM_ALG_NULL</td>
<td>this can be used as a stop value</td>
</tr>
</tbody>
</table>

```c
TPM_ALG_ID
CryptGetHashAlgByIndex(UINT32 index) // IN: the index
{
    return _cpri__GetHashAlgByIndex(index);
}
```

### 9.16.5.19 CryptSignHMAC()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>static TPM_RC</td>
<td></td>
</tr>
</tbody>
</table>
9.16.5.20 CryptHMACVerifySignature()

This function will verify a signature signed by a HMAC key.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>if invalid input or signature is not genuine</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
CryptHMACVerifySignature(
    OBJECT          *signKey,
    // IN: HMAC key signed the hash
    TPM2B_DIGEST    *hashData,
    // IN: digest being verified
    TPMT_SIGNATURE  *signature
    // IN: signature to be verified
)
{
    HMAC_STATE    hmacState;
    TPM2B_DIGEST digestToCompare;
    // HMAC algorithm self testing code may be inserted here
    digestToCompare.t.size = CryptStartHMAC2B(signature->signature.hmac.hashAlg,
        &signKey->sensitive.sensitive.bits.b, &hmacState);
    CryptUpdateDigest2B(&hmacState, &hashData->b);
    CryptCompleteHMAC2B(&hmacState, &digestToCompare.b);
    // Compare digest
    if(MemoryEqual(digestToCompare.t.buffer,
        (BYTE *) &signature->signature.hmac.digest,
        digestToCompare.t.size))
        return TPM_RC_SUCCESS;
    else
        return TPM_RC_SIGNATURE;
}
```
9.16.5.21 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIZE</td>
<td>sensitive data size is larger than allowed for the scheme</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
CryptGenerateKeyedHash(
    TPMT_PUBLIC *publicArea, // IN/OUT: the public area template
    // for the new key.
    TPMS SENSITIVE_CREATE *sensitiveCreate, // IN: sensitive creation data
    TPMT SENSITIVE *sensitive, // OUT: sensitive area
    TPM_ALG_ID kdfHashAlg, // IN: algorithm for the KDF
    TPM2B_SEED *seed, // IN: the seed
    TPM2B_NAME *name // IN: name of the object
)
{
    TPMT_KEYEDHASH_SCHEME *scheme;
    TPM_ALG_ID hashAlg;
    UINT16 hashBlockSize;

    scheme = &publicArea->parameters.keyedHashDetail.scheme;
    pAssert(publicArea->type == TPM_ALG_KEYEDHASH);

    // Pick the limiting hash algorithm
    if(scheme->scheme == TPM_ALG_NULL)
        hashAlg = publicArea->nameAlg;
    else if(scheme->scheme == TPM_ALG_XOR)
        hashAlg = scheme->details.xor.hashAlg;
    else
        hashAlg = scheme->details.hmac.hashAlg;

    hashBlockSize = CryptGetHashBlockSize(hashAlg);
    // if this is a signing or a decryption key, then the limit
    // for the data size is the block size of the hash. This limit
    // is set because larger values have lower entropy because of the
    // HMAC function.
    if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR
        && (publicArea->objectAttributes.decrypt
            || publicArea->objectAttributes.sign)
        && sensitiveCreate->data.t.size > hashBlockSize)
        return TPM_RC_SIZE;

    if(publicArea->objectAttributes.sensitiveDataOrigin == SET)
    {
        // Created block cannot be larger than the structure allows.
        if(hashBlockSize > MAX_SYM_DATA)
            hashBlockSize = MAX_SYM_DATA;

        // Create new keyedHash object
        sensitive->sensitive.bits.t.size = hashBlockSize;
        CryptKDFa(kdfHashAlg, &seed->b,
            /* sensitive", //This string is a vendor- */
            //specific information
            &name->b, // computed from the public template
```
463       // 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 //%        )
9.16.5.24 KDFa()

This function is used by functions outside of CryptUtil() to access _cpri_KDFa().

```c
void KDFa(
    TPM_ALG_ID hash,       // IN: hash algorithm used in HMAC
    TPM2B *key,            // IN: HMAC key
    const char *label,     // IN: a null-terminated label for KDF
    TPM2B *contextU,      // IN: context U
    TPM2B *contextV,      // IN: context V
    UINT32 sizeInBits,    // IN: size of generated key in bits
    BYTE *keyStream,      // OUT: key buffer
    UINT32 *counterInOut  // IN/OUT: caller may provide the
                          // iteration counter for incremental operations to avoid large intermediate
                          // buffers.
)

CryptKDFa(hash, key, label, contextU, contextV, sizeInBits,
         keyStream, counterInOut);
```

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().

```c
#ifdef TPM_ALG_KEYEDHASH
#define CryptKDFe(hashAlg, Z, label, partyUInfo, partyVInfo,
               sizeInBits, keyStream)
     _cpri__KDFe(((TPM_ALG_ID)hashAlg),
                 ((TPM2B *)Z),
                 ((const char *)label),
                 ((TPM2B *)partyUInfo),
                 ((TPM2B *)partyVInfo),
                 ((UINT32)sizeInBits),
                 ((BYTE *)keyStream))
#endif
```

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.

```c
#ifdef TPM_ALG_RSA
static void BuildRSA(
    OBJECT *rsaKey,
    RSA_KEY *key
)
#endif
```
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    if((rsaKey->attributes.publicOnly || rsaKey->privateExponent.t.size == 0)
556       key->privateKey = NULL;
557     else
558       key->privateKey = &(rsaKey->privateExponent.b);
559  }

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 \(p^2q\).

If only p is provided, q is computed by \(q = n/p\). If n mod p \(!= 0\), TPM_RC_BINDING is returned.

The key is validated by checking that a d can be found such that \(e \cdot d \mod ((p-1)(q-1)) = 1\). If d is found that satisfies this requirement, it will be placed in d.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_BINDING</td>
<td>the public and private portions of the key are not matched</td>
</tr>
</tbody>
</table>

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    pAssert(d != NULL && n != NULL && p != NULL);
572    // Set the exponent
573    if(e == 0)
574      e = RSA_DEFAULT_PUBLIC_EXPONENT;
575    // CRYPT_PARAMETER
576    retVal = _cpri__TestKeyRSA(d, e, n, p, q);
577    if(retVal == CRYPT_SUCCESS)
578      return TPM_RC_SUCCESS;
579    else
580      return TPM_RC_BINDING;       // convert CRYPT_PARAMETER
581  }

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CANCELLED</td>
<td>key generation has been cancelled</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>exponent is not prime or is less than 3; or could not find a prime using the provided parameters</td>
</tr>
</tbody>
</table>

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  { CRYPT_RESULT    retVal;
600         *counter = 0;
601         // _cpri_GenerateKeyRSA can return CRYPT_CANCEL or CRYPT_FAIL
602         retVal = _cpri__GenerateKeyRSA(&publicArea->unique.rsa.b,
603                        &sensitive->sensitive.rsa.b,
604                        publicArea->parameters.rsaDetail.keyBits,
605                        publicArea->parameters.rsaDetail.exponent,
606                        hashAlg,
607                        &seed->b,
608                        "RSA key by vendor",
609                        &name->b,
610                        counter);
611         // CRYPT_CANCEL -> TPM_RC_CANCELLED; CRYPT_FAIL -> TPM_RC_VALUE
612         return TranslateCryptErrors(retVal);
613  }
614 9.16.6.4  CryptLoadPrivateRSA()
9.16.6.4  CryptLoadPrivateRSA()
9.16.6.4  CryptLoadPrivateRSA()
9.16.6.4  CryptLoadPrivateRSA()
9.16.6.4  CryptLoadPrivateRSA()

This function is called to generate the private exponent of an RSA key. It uses CryptTestKeyRSA().

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_BINDING</td>
<td>public and private parts of rsaKey are not matched</td>
</tr>
</tbody>
</table>

615  TPM_RC
616  CryptLoadPrivateRSA(
617  OBJECT      *rsaKey    // IN: the RSA key object
618  )
619  { TPM_RC    result;
620      TPM_PUBLIC    *publicArea = &rsaKey->publicArea;
621      TPM_SENSITIVE *sensitive = &rsaKey->sensitive;
622      // Load key by computing the private exponent
623      // TPM_RC_BINDING
624      result = CryptTestKeyRSA(&(rsaKey->privateExponent.b),
625                          publicArea->parameters.rsaDetail.exponent,
626                          &{publicArea->unique.rsa.b},
627                          &{sensitive->sensitive.rsa.b},
628                          NULL);
629      if(result != TPM_RC_SUCCESS)
630          return result;
631      rsaKey->attributes.privateExp = SET;
632      return TPM_RC_SUCCESS;
633  )
634
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.

```c
TPMT_RSA_DECRYPT*
CryptSelectRSAScheme(
    TPMI_DH_OBJECT rsaHandle,   // IN: handle of sign key
    TPMT_RSA_DECRYPT *scheme    // IN: a sign or decrypt scheme
)
{
    OBJECT *rsaObject;
    TPMT_ASYM_SCHEME *keyScheme;

    // Get sign object pointer
    rsaObject = ObjectGet(rsaHandle);
    keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;

    // if the default scheme of the object is TPM_ALG_NULL, then select the
    // input scheme
    if(keyScheme->scheme == TPM_ALG_NULL)
        return scheme;

    // if the object scheme is not TPM_ALG_NULL and the input scheme is
    // TPM_ALG NULL, then select the default scheme of the object.
    else if(scheme->scheme == TPM_ALG_NULL)
        // if input scheme is NULL
        return
            (TPMT_RSA_DECRYPT *)keyScheme;

    // get here if both the object scheme and the input scheme are
    // not TPM_ALG NULL. Need to insure that they are the same.
    // IMPLEMENTATION NOTE: This could cause problems if future versions have
    // schemes that have more values than just a hash algorithm. A new function
    // (IsSchemeSame()) might be needed then.
    else if(   keyScheme->scheme == scheme->scheme
        && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
        return scheme;
    else
        // two different, incompatible schemes specified
        return NULL;
}
```

9.16.6.6 CryptDecryptRSA()

This function is the interface to __DecryptRSA(). It handles the return codes from that function and converts them from CRYPT_RESULT to TPM_RC values.
<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>The key is not a decryption key.</td>
</tr>
<tr>
<td>TPM_RC_BINDING</td>
<td>Public and private parts of the key are not cryptographically bound.</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>Size of data to decrypt is not the same as the key size.</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>Numeric value of the encrypted data is greater than the public exponent, or output buffer is too small for the decrypted message.</td>
</tr>
</tbody>
</table>

```c
TPM_RC

CryptDecryptRSA(
    UINT16 *dataOutSize, // OUT: size of plain text in byte
    BYTE *dataOut, // OUT: plain text
    OBJECT *rsaKey, // IN: internal RSA key
    TPM_NONCE *scheme, // IN: selects the padding scheme
    UINT16 cipherInSize, // IN: size of cipher text in byte
    BYTE *cipherIn, // IN: cipher text
    const char *label // IN: a label, when needed
) {

    RSA_KEY key;
    CRYPT_RESULT retVal = CRYPT_SUCCESS;
    UINT32 dSize; // Place to put temporary value for the returned data size

    TPMI_ALG_HASH hashAlg = TPM_ALG_NULL; // hash algorithm in the selected padding scheme

    // pointer checks
    pAssert(   (dataOutSize != NULL) && (dataOut != NULL)
        && (rsaKey != NULL) && (cipherIn != NULL));

    // The public type is a RSA object
    pAssert(rsaKey->publicArea.type == TPM_ALG_RSA);

    // Must have the private portion loaded. This check is made before this function is called.
    pAssert(rsaKey->attributes.publicOnly == CLEAR);

    if(rsaKey->publicArea.objectAttributes.decrypt != SET)
        return TPM_RC_ATTRIBUTES;

    // decryption requires that the private modulus be present
    if(rsaKey->attributes.privateExp == CLEAR)
        {
            TPM_RC result;
            // Load key by computing the private exponent
            // CryptLoadPrivateRSA may return TPM_RC_BINDING
            result = CryptLoadPrivateRSA(rsaKey);
            if(result != TPM_RC_SUCCESS)
                return result;
        }

    // the input buffer must be the size of the key
    if(cipherInSize != rsaKey->publicArea.unique.rsa.t.size)
        return TPM_RC_SIZE;

    BuildRSA(rsaKey, &key);

    // Initialize the dOutSize parameter
    dSize = *dataOutSize;

    // For OAEP scheme, initialize the hash algorithm for padding
    if(scheme->scheme == TPM_ALG_OAEP)
        hashAlg = scheme->details.oaep.hashAlg;
```
// _cpri__DecryptRSA may return CRYPT_PARAMETER CRYPT_FAIL CRYPT_SCHEME
retVal = _cpri__DecryptRSA(dSize, dataOut, &key, scheme->scheme,
cipherInSize, cipherIn, hashAlg, label);

// Scheme must have been validated when the key was loaded/imported
if(retVal == CRYPT_SCHEME) //++ this needs to be an assert
    return TPM_RC_FAILURE;

// Set the return size
pAssert(dSize <= UINT16_MAX);
*dataOutSize = (UINT16)dSize;

// CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_FAIL -> TPM_RC_VALUE
return TranslateCryptErrors(retVal);

9.16.6.7 CryptEncryptRSA()

This function provides the interface to _cpri__EncryptRSA().

<table>
<thead>
<tr>
<th>TPM_RC_ATTRIBUTES</th>
<th>rsaKey is not a valid decryption key</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEME</td>
<td>scheme is not supported</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>numeric value of dataIn is greater than the key modulus</td>
</tr>
</tbody>
</table>

TPM_RC

CryptEncryptRSA(
    UINT16 *cipherOutSize, // OUT: size of cipher text in byte
    BYTE *cipherOut, // OUT: cipher text
    OBJECT *rsaKey, // IN: internal RSA key
    TPMT_RSA_DECRYPT *scheme, // IN: selects the padding scheme
    UINT16 dataInSize, // IN: size of plain text in byte
    BYTE *dataIn, // IN: plain text
    const char *label // IN: an optional label
)

}
// For OAEP scheme, initialize the hash algorithm for padding
if (scheme->scheme == TPM_ALG_OAEP)
    hashAlg = scheme->details.oaep.hashAlg;

// Encrypt the data _cpri__EncryptRSA may return CRYPT_PARAMETER or CRYPT_SCHEME
retVal = _cpri__EncryptRSA(&cOutSize, cipherOut, &key, scheme->scheme,
dataInSize, dataIn, hashAlg, label);

pAssert (cOutSize <= UINT16_MAX);
*cipherOutSize = (UINT16)cOutSize;

// CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_SCHEME -> TPM_RC_SCHEME
return TranslateCryptErrors(retVal);

9.16.6.8  CryptSignRSA()

This function is used to sign a digest with an RSA signing key.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_BINDING</td>
<td>public and private part of signKey are not properly bound</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>scheme is not supported</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>hashData is larger than the modulus of signKey, or the size of hashData does not match hash algorithm in scheme</td>
</tr>
</tbody>
</table>

static TPM_RC
CryptSignRSA(OBJECT *signKey, // IN: RSA key signs the hash TPM2B_DIGEST *hashData, // IN: hash to be signed)
{
    UINT32  signSize;
    RSA_KEY  key;
    CRYPT_RESULT  retVal;

    pAssert(    (signKey != NULL) && (scheme != NULL)
    && (hashData != NULL) && (sig != NULL));

    // assume that the key has private part loaded and that it is a signing key.
    pAssert(    (signKey->attributes.publicOnly == CLEAR)
    && (signKey->publicArea.objectAttributes.sign == SET));

    // check if the private exponent has been computed
    if (signKey->attributes.privateExp == CLEAR)
    {
        // need to compute the private exponent
        TPM_RC  result;
        // May return TPM_RC_BINDING
        result = CryptLoadPrivateRSA(signKey);
        if (result != TPM_RC_SUCCESS)
            return result;
    }

    BuildRSA(signKey, &key);

    // initialize the common signature values
    sig->sigAlg = scheme->scheme;
    sig->signature.any.hashAlg = scheme->details.any.hashAlg;

    // _crypi__SignRSA can return CRYPT_SCHEME and CRYPT_PARAMETER
9.16.6.9 CryptRSAVerifySignature()

This function is used to verify signature signed by a RSA key.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>if signature is not genuine</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>signature scheme not supported</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
CryptRSAVerifySignature(
    OBJECT *signKey,       // IN: RSA key signed the hash
    TPM2B_DIGEST *hashData, // IN: hash being signed
    TPMT_SIGNATURE *sig    // IN: signature to be verified
) {
    RSA_KEY key;
    CRYPT_RESULT retVal;
    // Validate parameter assumptions
    pAssert((signKey != NULL) && (hashData != NULL) && (sig != NULL));
    // This is a public-key-only operation
    BuildRSA(signKey, &key);
    // Call crypto engine to verify signature
    // _cpri_ValidateSignatureRSA may return CRYPTO_FAIL or CRYPTO_SCHEME
    retVal = _cpri__ValidateSignatureRSA(&key, sig->sigAlg,
                                           sig->signature.any.hashAlg,
                                           hashData->t.size,
                                           hashData->t.buffer,
                                           sig->signature.rsassa.sig.t.size,
                                           sig->signature.rsassa.sig.t.buffer,
                                           0);
    // _cpri__ValidateSignatureRSA can return CRYPTO_SUCCESS, CRYPTO_FAIL, or CRYPTO_SCHEME.
    // Translate CRYPTO_FAIL to TPM_RC_SIGNATURE
    if(retVal == CRYPTO_FAIL)
        return TPM_RC_SIGNATURE;
    // CRYPTO_SCHEME -> TPM_RC_SCHEME
    return TranslateCryptErrors(retVal);
}
```
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.

```
#ifdef TPM_ALG_ECC // 3
static const ECC_CURVE * CryptEccGetCurveDataPointer(
    TPM_ECC_CURVE curveID // IN: id of the curve
) {
    return _cpri___EccGetParametersByCurveId(curveID);
}
```

9.16.7.2 CryptEccGetKeySizeInBits()

This function returns the size in bits of the key associated with a curve.

```
UINT16 CryptEccGetKeySizeInBits(
    TPM_ECC_CURVE curveID // IN: id of the curve
) {
    const ECC_CURVE *curve = CryptEccGetCurveDataPointer(curveID);
    if(curve == NULL)
        return 0;
    return curve->keySizeBits;
}
```

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 CyrptUtil_fp.h with the //% removed.

```
//% #define CryptEccGetKeySizeInBytes(curve)\((CryptEccGetKeySizeInBits(curve)+7)/8\)
```

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}.

```
const TPM2B * CryptEccGetParameter(
    char p, // IN: the parameter selector
    TPM_ECC_CURVE curve // IN: the curve id
)
{
    const ECC_CURVE *curveData = _cpri___EccGetParametersByCurveId(curve);
    if(curveData == NULL)
        return NULL;
    switch (p) {
    case 'p':
        return curveData->curveData->p;
```
9.16.7.5 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```c
const TPMT_ECC_SCHEME *
CryptGetCurveSignScheme(TPM_ECC_CURVE curveId) // IN: The curve selector
{
    const ECC_CURVE *curveData = _cpri__EccGetParametersByCurveId(curveId);
    if(curveData == NULL)
        return NULL;
    return &(curveData->sign);
}
```

9.16.7.6 CryptEccIsPointOnCurve()

This function will validate that an ECC point is on the curve of given curveID.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if the point is on curve</td>
</tr>
<tr>
<td>FALSE</td>
<td>if the point is not on curve</td>
</tr>
</tbody>
</table>

```c
BOOL
CryptEccIsPointOnCurve(TPM_ECC_CURVE curveID, // IN: ECC curve ID
TPMS_ECC_POINT *Q // IN: ECC point
)
{
    // ECC algorithm self testing code may be inserted here
    // Call crypto engine function to check if a ECC public point is on the
    // given curve
    if(_cpri__EccIsPointOnCurve(curveID, Q))
        return TRUE;
    else
        return FALSE;
}
```

9.16.7.7 CryptNewEccKey()

This function creates a random ECC key that is not derived from other parameters as is a Primary Key.
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ECC_POINT</td>
<td>invalid optional ECC point ( pIn )</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>multiplication resulted in a point at infinity</td>
</tr>
</tbody>
</table>

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.
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEME</td>
<td>unsupported scheme</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>invalid commit status</td>
</tr>
</tbody>
</table>

static TPM_RC
CryptSignECC()

OBJECT *signKey, // IN: ECC key to sign the hash
TPMT_SIG_SCHEME *scheme, // IN: sign scheme
TPM2B_DIGEST *hashData, // IN: hash to be signed
TPM2B_SIGNATURE *signature // OUT: signature

if(CryptIsSplitSign(scheme->scheme))
{
    // When this code was written, the only split scheme was ECDAA
    // (which can also be used for U-Prove).
    if(!CryptGenerateR(&r, &scheme->details.ecdaa.count, signKey->publicArea.parameters.eccDetail.curveID, &signKey->name))
        return TPM_RC_VALUE;

    pr = &r;

    // _cpri__GenerateKeyEcc only has one error return (CRYPT_PARAMETER) which means
    // that the hash algorithm is not supported. This should not be possible
    retVal = _cpri__GenerateKeyEcc(&publicArea->unique.ecc, &sensitive->sensitive.ecc, publicArea->parameters.eccDetail.curveID, hashAlg, &seed->b, "ECC key by vendor", &name->b, counter);
    // This will only be useful if _cpri__GenerateKeyEcc return CRYPTO_CANCEL
    return TranslateCryptErrors(retVal);
}
1044 }  // Call crypto engine function to sign
1045 // _cpri__SignEcc may return CRYPT_SCHEME
1046 retVal = _cpri__SignEcc(&signature->signature.ecdsa.signatureR,
1047   &signature->signature.ecdsa.signatureS,
1048   scheme->scheme,
1049   scheme->details.any.hashAlg,
1050   signKey->publicArea.parameters.eccDetail.curveID,
1051   &signKey->sensitive.sensitive.ecc,
1052   &hashData->b,
1053   pr
1054 );
1055 if(CryptIsSplitSign(scheme->scheme) && retVal == CRYPT_SUCCESS)
1056   CryptEndCommit(scheme->details.ecdaa.count);
1057 // CRYPT_SCHEME->TPM_RC_SCHEME
1058 return TranslateCryptErrors(retVal);
1059 }

9.16.7.11 CryptECCVerifySignature()

This function is used to verify a signature created with an ECC key.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>if signature is not valid</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>the signing scheme or hashAlg is not supported</td>
</tr>
</tbody>
</table>

static TPM_RC
1062 CryptECCVerifySignature(
1063 {
1064   OBJECT *signKey,     // IN: ECC key signed the hash
1065   TPM2B_DIGEST *hashData,  // IN: hash being signed
1066   TPMT_SIGNATURE *signature  // IN: signature to be verified
1067 )
1068 {
1069   CRYPT_RESULT retVal;
1070   // This implementation uses the fact that all the defined ECC signing
1071   // schemes have the hash as the first parameter.
1072   // _cpriValidateSignatureEcc may return CRYPT_FAIL or CRYPT_SCHEME
1073   retVal = _cpri__ValidateSignatureEcc(&signature->signature.ecdsa.signatureR,
1074     &signature->signature.ecdsa.signatureS,
1075     &signature->signAlg,
1076     signature->signature.any.hashAlg,
1077     signKey->publicArea.parameters.eccDetail.curveID,
1078     &signKey->publicArea.unique.ecc,
1079     &hashData->b);
1080   if(retVal == CRYPT_FAIL)
1081     return TPM_RC_SIGNATURE;
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.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>r value computed</td>
</tr>
<tr>
<td>FALSE</td>
<td>no r value computed</td>
</tr>
</tbody>
</table>

```c

BOOL CryptGenerateR(
    TPM2B_ECC_PARAMETER *r,  // OUT: the generated random value
    UINT16 *c,              // IN/OUT: count value.
    TPMI_ECC_CURVE curveID, // IN: the curve for the value
    TPM2B_NAME *name        // IN: optional name of a key to
                           //     associate with 'r'
) {

    TPM2B_TYPE(8B, 8);
    cntr = {8,{0}};
    iterations;
    const TPM2B *n;
    UINT64                   currentCount = gr.commitCounter;

    n = CryptEccGetParameter('n', curveID);
    pAssert(r != NULL && n != NULL);

    // If this is the commit phase, use the current value of the commit counter
    if(c != NULL) {
        UINT16 t1;
        // if the array bit is not set, can't use the value.
        if(!BitIsSet((*c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray)))
            return FALSE;

        // If it is the sign phase, figure out what the counter value was
        // when the commitment was made.
        // When gr.commitArray has less than 64K bits, the extra
        // bits of 'c' are used as a check to make sure that the
        // signing operation is not using an out of range count value
        t1 = (UINT16)currentCount;
        // If the lower bits of c are greater or equal to the lower bits of t1
        // then the upper bits of t1 must be one more than the upper bits
        // of c
        if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
            // Since the counter is behind, reduce the current count
            currentCount = currentCount - (COMMIT_INDEX_MASK + 1);

        t1 = (UINT16)currentCount;
        // If the array bit is not set, can't use the value.
        if(!BitIsSet((*c & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
            return FALSE;

        // set the counter to the value that was
        // present when the commitment was made
        currentCount = (currentCount & 0xffffffffffff0000) | *c;

        // Marshal the count value to a TPM2B buffer for the KDF
        cntr.t.size = sizeof(currentCount);
        UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);

        // Now can do the KDF to create the random value for the signing operation
        // During the creation process, we may generate an r that does not meet the
```
// requirements of the random value.
// want to generate a new r.
r->t.size = n->size;

// Arbitrary upper limit on the number of times that we can look for
// a suitable random value. The normally number of tries will be 1.
for( iterations = 1; iterations < 1000000; )
{
    BYTE *pr = &r->b.buffer[0];
    int i;
    CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, "ECDAA Commit", 
        name, &cntr.b, n->size * 8, r->t.buffer, &iterations);
    // random value must be less than the prime
    if(CryptCompare(r->b.size, r->b.buffer, n->size, n->buffer) >= 0)
    {
        continue;
    }
    // in this implementation it is required that at least bit
    // in the upper half of the number be set
    for(i = n->size/2; i > 0; i--)
        if(*pr++ != 0)
            return TRUE;
}
return FALSE;

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.

UINT16
CryptCommit(
    void
)
{
    UINT16 oldCount = (UINT16)gr.commitCounter;
    gr.commitCounter++;
    BitSet(oldCount & COMMIT_INDEX_MASK, gr.commitArray, sizeof(gr.commitArray));
    return oldCount;

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.

void
CryptEndCommit(
    UINT16 c // IN: the counter value of the commitment
)
{
    BitSet(c & COMMIT_INDEX_MASK, gr.commitArray, sizeof(gr.commitArray));

9.16.7.15 CryptCommitCompute()

This function performs the computations for the TPM2_Commit() command. This could be a macro.
## Error Returns

<table>
<thead>
<tr>
<th>Error Return</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>$K, L, \text{ or } E$ is the point at infinity</td>
</tr>
<tr>
<td>TPM_RC_CANCELLED</td>
<td>Command was cancelled</td>
</tr>
</tbody>
</table>

### 9.16.7.16 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>Get parameters success</td>
</tr>
<tr>
<td>FALSE</td>
<td>Unsupported ECC curve ID</td>
</tr>
</tbody>
</table>

```c
BOOL CryptEccGetParameters(
    TPM_ECC_CURVE                curveId,  // IN: ECC curve ID
    TPMS_ALGORITHM_DETAIL_ECC   *parameters // OUT: ECC parameters
) {
    const ECC_CURVE             *curve = _cpri__EccGetParametersByCurveId(curveId);
    const ECC_CURVE_DATA        *data;
    if(curve == NULL)
        return FALSE;
    data = curve->curveData;
    parameters->curveID = curve->curveId;
    // Key size in bit
    parameters->keySize = curve->keySizeBits;
    // KDF
    parameters->kdf = curve->kdf;
    // Sign
    parameters->sign = curve->sign;
    // Copy p value
    MemoryCopy2B(&parameters->p.b, data->p);
    // Copy a value
    MemoryCopy2B(&parameters->a.b, data->a);
}
```
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 ECDAA. ECDAA 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_ECDAA
1281    || scheme == TPM_ALG_ECDAA
1282    #endif
1283    )
1284 )
9.16.8 Symmetric Functions

9.16.8.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SYMMETRIC</td>
<td>unsupported symmetric algorithm</td>
</tr>
</tbody>
</table>

```c
static TPM_RC ParmDecryptSym(
    TPM_ALG_ID           symAlg,  // IN: the symmetric algorithm
    TPM_ALG_ID           hash,    // IN: hash algorithm for KDFa
    UINT16               keySizeInBits,  // IN: key key size in bits
    TPM2B *key,         // IN: KDF HMAC key
    TPM2B *nonceCaller,  // IN: nonce caller
    TPM2B *nonceTpm,    // IN: nonce TPM
    UINT32               dataSize,  // IN: size of parameter buffer
    BYTE                *data   // OUT: buffer to be decrypted
) {

    // KDF output buffer
    // It contains parameters for the CFB encryption
    // From MSB to LSB, they are the key and iv
    BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
    // Symmetric key size in byte
    UINT16           keySize = (keySizeInBits + 7) / 8;
    TPM2B_IV         iv;

    if(iv.t.size == 0)
        return TPM_RC_SYMMETRIC;

    // Generate key and iv
    CryptKDFa(hash, key, "CFB", nonceCaller, nonceTpm,
             keySizeInBits + (iv.t.size * 8), symParmString, NULL);
    MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
    CryptSymmetricDecrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB, symParmString,
                           &iv, dataSize, data);
    return TPM_RC_SUCCESS;
}
```

9.16.8.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SYMMETRIC</td>
<td>unsupported symmetric algorithm</td>
</tr>
</tbody>
</table>

```c
static TPM_RC ParmEncryptSym(
    TPM_ALG_ID           symAlg,  // IN: symmetric algorithm
    TPM_ALG_ID           hash,    // IN: hash algorithm for KDFa
    UINT16               keySizeInBits,  // IN: AES key size in bits
    TPM2B *key,         // IN: KDF HMAC key
    TPM2B *nonceCaller,  // IN: nonce caller
    TPM2B *nonceTpm,    // IN: nonce TPM
    UINT32               dataSize,  // IN: size of parameter buffer
    BYTE                *data   // OUT: buffer to be encrypted
) {
```
9.16.8.3 CryptGenerateKeySymmetric()

This function derives a symmetric cipher key from the provided seed.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_KEY_SIZE</td>
<td>key size in the public area does not match the size in the sensitive creation area</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
CryptGenerateKeySymmetric(
    TPMT_PUBLIC             *publicArea, // IN/OUT: The public area template
    // for the new key.
    TPMS_SENSITIVE_CREATE   *sensitiveCreate, // IN: sensitive creation data
    TPMT_SENSITIVE          *sensitive, // OUT: sensitive area
    TPM_ALG_ID              hashAlg, // IN: hash algorithm for the KDF
    TPM2B_SEED              *seed, // IN: seed used in creation
    TPM2B_NAME              *name // IN: name of the object
)
{
    // If this is not a new key, then the provided key data must be the right size
    if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR
        && (sensitiveCreate->data.t.size * 8) !=
        publicArea->parameters.symDetail.keyBits.sym)
        return TPM_RC_KEY_SIZE;

    // Make sure that the key size is OK.
    // This implementation only supports symmetric key sizes that are
    // multiples of 8
    if(publicArea->parameters.symDetail.keyBits.sym % 8 != 0)
        return TPM_RC_KEY_SIZE;

    if(publicArea->objectAttributes.sensitiveDataOrigin == SET)
    {
        // Create new symmetric key
        sensitive->sensitive.sym.t.size =
            (publicArea->parameters.symDetail.keyBits.sym + 7)/8;

        CryptKDFa(hashAlg, &seed->b, "sensitive", &name->b,
            NULL, publicArea->parameters.symDetail.keyBits.sym,
```
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.

```c
#define TPM_ALG_KEYEDHASH

static TPM_RC
CryptXORObfuscation(  
  TPM_ALG_ID           hash, // IN: hash algorithm for KDF  
  TPM2B               *key,  // IN: KDF key  
  TPM2B               *contextU, // IN: contextU  
  TPM2B               *contextV, // IN: contextV  
  UINT32               dataSize, // IN: size of data buffer  
  BYTE*               data  // IN/OUT: data to be XORed in place
)  
{  
  BYTE             mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer  
  BYTE            *pm;  
  UINT32           i;  
  UINT32           counter = 0;  
  UINT16           hLen = CryptGetHashDigestSize(hash);  
  UINT32           requestSize = dataSize * 8;  
  INT32            remainBytes = (INT32) dataSize;  
  
  pAssert((key != NULL) && (data != NULL) && (hLen != 0));  
  
  // Call KDFa to generate XOR mask  
  for(; remainBytes > 0; remainBytes -= hLen)  
  {  
    // Make a call to KDFa to get next iteration  
    CryptKDFaOnce(hash, key, "XOR", contextU, contextV,  
                  requestSize, mask, &counter);  
    
    // XOR next piece of the data  
    pm = mask;  
    for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)  
      *data++ ^= *pm++;  
  }  
  return TPM_RC_SUCCESS;  
}
#endif
```
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.

```c
void CryptInitUnits(void)
{
    // Call crypto engine unit initialization
    // We assume crypt engine initialization should always succeed. Otherwise,
    // TPM should go to failure mode.
    // This is used to make sure that the correct version of CryptoEngine
    // has been linked
    _cpri__InitCryptoUnits();
    return;
}
```

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.

```c
void CryptStopUnits(void)
{
    // Call crypto engine unit stopping
    _cpri__StopCryptoUnits();
    return;
}
```

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.

```c
BOOL CryptUtilStartup(
    STARTUP_TYPE type, // IN: the startup type
)
{
    // Make sure that the crypto library functions are ready
    if( !_cpri__Startup())
        return FALSE;
    if(type == SU_RESET)
    {
        #ifdef TPM_ALG_ECDAA
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if it is an asymmetric algorithm</td>
</tr>
<tr>
<td>FALSE</td>
<td>if it is not an asymmetric algorithm</td>
</tr>
</tbody>
</table>

```c
BOOL CryptIsAsymAlgorithm(  
    TPM_ALG_ID           algID  // IN: algorithm ID  
)  
{
    return ( 0  
    #ifdef TPM_ALG_RSA  // if it is RSA  
        || algID == TPM_ALG_RSA  
    #ifdef TPM_ALG_ECC  // if it is ECC  
        || algID == TPM_ALG_ECC  
    #endif  
    #endif  
);  
}
```

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.

```c
INT16 CryptGetSymmetricBlockSize(  
    TPM_ALG_SYM         algorithm,  // IN: symmetric algorithm  
    UINT16               keySize  // IN: key size in bit  
)  
{
```
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.

```c
void CryptSymmetricEncrypt(
    BYTE *encrypted, // OUT: the encrypted data
    TPM_ALG_ID algorithm, // IN: algorithm for encryption
    UINT16 keySizeInBits, // IN: key size in bits
    TPMI_ALG_SYM_MODE mode, // IN: symmetric encryption mode
    BYTE *key, // IN: encryption key
    TPM2B_IV *ivIn, // IN/OUT: Input IV and output chaining value for the next block
    UINT32 dataSize, // IN: data size in byte
    BYTE* data // IN/OUT: data buffer
) {
    BYTE *iv = NULL;
    BYTE defaultIV[sizeof(TPMT_HA)];
    pAssert(   ((mode == TPM_ALG_ECB) && (ivIn->t.size == 0))
        || (mode != TPM_ALG_ECB));

    // Both SM4 and AES have block size of 128 bits
    // If the iv is not provided, create a default of 0
    if(ivIn == NULL) {
        // Initialize the default IV
        iv = defaultIV;
        MemorySet(defaultIV, 0, 16);
    } else {
        // A provided IV has to be the right size
        pAssert(mode == TPM_ALG_ECB || ivIn->t.size == 16);
        iv = &(ivIn->t.buffer[0]);
    }

    switch(algorithm) {
    #ifdef TPM_ALG_AES
    case TPM_ALG_AES:
        switch (mode) {
            case TPM_ALG_CTR:
                _cpri__AESEncryptCTR(encrypted, keySizeInBits, key, iv,
                                     dataSize, data);
                break;
            case TPM_ALG_OFB:
                _cpri__AESEncryptOFB(encrypted, keySizeInBits, key, iv,
```
```c
    break;
  case TPM_ALG_CBC:
    _cpri__AESEncryptCBC(encrypted, keySizeInBits, key, iv,
    dataSize, data);
    break;
  case TPM_ALG_CFB:
    _cpri__AESEncryptCFB(encrypted, keySizeInBits, key, iv,
    dataSize, data);
    break;
  case TPM_ALG_ECB:
    _cpri__AESEncryptECB(encrypted, keySizeInBits, key,
    dataSize, data);
    break;
  default:
    pAssert(0);
  }
  } break;
#endif
#else
  default:
    pAssert(FALSE);
  break;
#endif
return;
}
```
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.

```c
void CryptSymmetricDecrypt(
    BYTE *decrypted,  // IN: algorithm for decryption
    TPM_ALG_ID algorithm,  // IN: algorithm for encryption
    UINT16 keySizeInBits,  // IN: key size in bits
    TPMI_ALG_SYM_MODE mode,  // IN: symmetric encryption mode
    BYTE *key,  // IN: encryption key
    TPM2B_IV *ivIn,  // IN/OUT: IV for next block
    UINT32 dataSize,  // IN: data size in byte
    BYTE* data  // IN/OUT: data buffer
)
```

If the IV is not provided, create a default of 0:

```c
BYTE defaultIV[sizeof(TPMT_HA)];
```

If the IV is both SM4 and AES have block size of 128 bits:

```c
if(ivIn == NULL)
```

Initialize the default IV:

```c
MemorySet(defaultIV, 0, 16);
```

A provided IV has to be the right size:

```c
pAssert(mode == TPM_ALG_ECB || ivIn->t.size == 16);
```

```c
BYTE* iv = &((ivIn->t.buffer[0]);
```

```
switch(algorithm)
```n
```

```c
# ifdef TPM_ALG_AES
```

```c
case TPM_ALG_AES:
```

```c
switch (mode)
```

```c
```

```c
# ifdef TPM_ALG_SM4
```

```c
case TPM_ALG_SM4
```

```c
```

```c
```

```c
```

```c
```

```
```

```c
```

```
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```n
case TPM_ALG_CFB:
  _cpri___AESDecryptCFB(decrypted, keySizeInBits, key, iv,
                   dataSize, data);
  break;

case TPM_ALG_ECB:
  _cpri___AESDecryptECB(decrypted, keySizeInBits, key,
                   dataSize, data);
  break;

default:
  pAssert(0);
}
break;
#endif //TPM_ALG_AES

#ifdef TPM_ALG_SM4

  case TPM_ALG_SM4 :
  switch (mode)
  {
  case TPM_ALG_CTR:
    _cpri____SM4DecryptCTR(decrypted, keySizeInBits, key, iv,
               dataSize, data);
    break;
  case TPM_ALG_OFB:
    _cpri____SM4DecryptOFB(decrypted, keySizeInBits, key, iv,
               dataSize, data);
    break;
  case TPM_ALG_CBC:
    _cpri____SM4DecryptCBC(decrypted, keySizeInBits, key, iv,
               dataSize, data);
    break;
  case TPM_ALG_CFB:
    _cpri____SM4DecryptCFB(decrypted, keySizeInBits, key, iv,
               dataSize, data);
    break;
  case TPM_ALG_ECB:
    _cpri____SM4DecryptECB(decrypted, keySizeInBits, key,
               dataSize, data);
    break;
  default:
    pAssert(0);
  }
  break;
#endif //TPM_ALG_SM4
} 
default:
  pAssert(FALSE);
break;
}
return;

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>keyHandle does not reference a valid decryption key</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>invalid ECC key (public point is not on the curve)</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>RSA key with an unsupported padding scheme</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>numeric value of the data to be decrypted is greater than the RSA key modulus</td>
</tr>
</tbody>
</table>

```c
TPM_RC
CryptSecretEncrypt(
    TPMI_DH_OBJECT  keyHandle,  // IN: encryption key handle
    const char *label,     // IN: a null-terminated string as L
    TPM2B_DATA *data,      // OUT: secret value
    TPM2B_ENCRYPTED_SECRET *secret // OUT: secret structure
)
```

```c
TPM_RC       result = TPM_RC_SUCCESS;
OBJECT      *encryptKey = ObjectGet(keyHandle); // TPM key used for encrypt
pAssert(data != NULL && secret != NULL);
// The output secret value has the size of the digest produced by the nameAlg.
data->t.size = CryptGetHashDigestSize(encryptKey->publicArea.nameAlg);
```

```c
pAssert(encryptKey->publicArea.objectAttributes.decrypt == SET);
switch(encryptKey->publicArea.type)
{
    #ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            {  // Use OAEP scheme
                TPMT_RSA_DECRYPT scheme;
                scheme.scheme = TPM_ALG_OAEP;
                scheme.details.oaep.hashAlg = encryptKey->publicArea.nameAlg;
                // Create secret data from RNG
                CryptGenerateRandom(data->t.size, data->t.buffer);
                // Encrypt the data by RSA OAEP into encrypted secret
                result = CryptEncryptRSA(&secret->t.size, secret->t.secret,
                                            encryptKey, &scheme,
                                            data->t.size, data->t.buffer, label);
                if(result != TPM_RC_SUCCESS)
                    return result;
            } break;
    #endif //TPM_ALG_RSA
    #ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            {  // Need to make sure that the public point of the key is on the curve defined by the key.
                BYTE *buffer = secret->t.secret;
                if(!_cpri__EccIsPointOnCurve(encryptKey->publicArea.parameters.eccDetail.curveID,
                                              &encryptKey->publicArea.unique.ecc))
```

```
```
return TPM_RC_KEY;

// Call crypto engine to create an auxiliary ECC key
// We assume crypto engine initialization should always succeed.
// Otherwise, TPM should go to failure mode.
CryptNewEccKey(encryptKey->publicArea.parameters.eccDetail.curveID,
    &eccPublic, &eccPrivate);

// Marshal ECC public to secret structure. This will be used by the
// recipient to decrypt the secret with their private key.
secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);

// Compute ECDH shared secret which is \( R = [d]Q \) where \( d \) is the private
// part of the ephemeral key and \( Q \) is the public part of a TPM key.
// TPM_RC_KEY error return from CryptComputeECDHSecret because the
// auxiliary ECC key is just created according to the parameters of
// input ECC encrypt key.
if( CryptEccPointMultiply(&eccSecret,
        encryptKey->publicArea.parameters.eccDetail.curveID,
        &eccPrivate,
        &encryptKey->publicArea.unique.ecc) != CRYPT_SUCCESS)
    return TPM_RC_KEY;

// The secret value is computed from \( Z \) using KDFe as:
// secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
// Where:
// HashID  the nameAlg of the decrypt key
// Z      the x coordinate (Px) of the product (P) of the point (Q) of
//        the secret and the private x coordinate (de,V) of the
//        decryption key
// Use a null-terminated string containing "SECRET"
// PartyUInfo  the x coordinate of the point in the secret (Qe,U )
// PartyVInfo  the x coordinate of the public key (Qs,V )
// bits    the number of bits in the digest of HashID
// Retrieve seed from KDFe
CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
    eccPublic.x.b, &encryptKey->publicArea.unique.ecc.x.b,
    data->t.size * 8, data->t.buffer);
}

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
<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>RSA key is not a decryption key</td>
</tr>
<tr>
<td>TPM_RC_BINDING</td>
<td>Invalid RSA key (public and private parts are not cryptographically bound.</td>
</tr>
<tr>
<td>TPM_RC_ECC_POINT</td>
<td>ECC point in the secret is not on the curve</td>
</tr>
<tr>
<td>TPM_RC_INSUFFICIENT</td>
<td>failed to retrieve ECC point from the secret</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>key of unsupported type</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>multiplication resulted in ECC point at infinity</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>data to decrypt is not of the same size as RSA key</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>For RSA key, numeric value of the encrypted data is greater than the modulus, or the recovered data is larger than the output buffer. For keyedHash or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.</td>
</tr>
</tbody>
</table>

```c
TPM_RC_FAILURE

TPM_RC

CryptSecretDecrypt(tpmKey, // IN: decrypt key
    TPM2B_NONCE *nonceCaller, // IN: nonceCaller. It is needed for
   // symmetric decryption. For
   // asymmetric decryption, this
   // parameter is NULL
   const char *label, // IN: a null-terminated string as L
   TPM2B_ENCRYPTED_SECRET *secret, // IN: input secret
   TPM2B_DATA *data // OUT: decrypted secret value
)

TPM_RC      result = TPM_RC_SUCCESS;
OBJECT      *decryptKey = ObjectGet(tpmKey); // TPM key used for decrypting

// Decryption for secret
switch(decryptKey->publicArea.type)
{
    #ifdef TPM_ALG_RSA
    case TPM_ALG_RSA:
    {
        TPM_rsa_decrypt scheme;
        // Use OAEP scheme
        scheme.scheme = TPM_ALG_OAEP;
        scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
        // Set the output buffer capacity
        data->t.size = sizeof(data->t.buffer);
        // Decrypt seed by RSA OAEP
        result = CryptDecryptRSA(&data->t.size, data->t.buffer, decryptKey,
            &scheme,
            secret->t.size, secret->t.secret,label);
        if( result == TPM_RC_SUCCESS
            && data->t.size > CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
            return TPM_RCVALUE;
    }
    #endif //TPM_ALG_RSA
    #ifdef TPM_ALG_ECC
```

---

**Error Returns**

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>RSA key is not a decryption key</td>
</tr>
<tr>
<td>TPM_RC_BINDING</td>
<td>Invalid RSA key (public and private parts are not cryptographically bound.)</td>
</tr>
<tr>
<td>TPM_RC_ECC_POINT</td>
<td>ECC point in the secret is not on the curve</td>
</tr>
<tr>
<td>TPM_RC_INSUFFICIENT</td>
<td>failed to retrieve ECC point from the secret</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>key of unsupported type</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>multiplication resulted in ECC point at infinity</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>data to decrypt is not of the same size as RSA key</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>For RSA key, numeric value of the encrypted data is greater than the modulus, or the recovered data is larger than the output buffer. For keyedHash or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.</td>
</tr>
</tbody>
</table>

**C Code Snippet**

```c
TPM_RC

CryptSecretDecrypt(tpmKey, // IN: decrypt key
    TPM2B_NONCE *nonceCaller, // IN: nonceCaller. It is needed for
   // symmetric decryption. For
   // asymmetric decryption, this
   // parameter is NULL
   const char *label, // IN: a null-terminated string as L
   TPM2B_ENCRYPTED_SECRET *secret, // IN: input secret
   TPM2B_DATA *data // OUT: decrypted secret value
)

TPM_RC      result = TPM_RC_SUCCESS;
OBJECT      *decryptKey = ObjectGet(tpmKey); // TPM key used for decrypting

// Decryption for secret
switch(decryptKey->publicArea.type)
{
    #ifdef TPM_ALG_RSA
    case TPM_ALG_RSA:
    {
        TPM_rsa_decrypt scheme;
        // Use OAEP scheme
        scheme.scheme = TPM_ALG_OAEP;
        scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
        // Set the output buffer capacity
        data->t.size = sizeof(data->t.buffer);
        // Decrypt seed by RSA OAEP
        result = CryptDecryptRSA(&data->t.size, data->t.buffer, decryptKey,
            &scheme,
            secret->t.size, secret->t.secret,label);
        if( result == TPM_RC_SUCCESS
            && data->t.size > CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
            return TPM_RCVALUE;
    }
    #endif //TPM_ALG_RSA
    #ifdef TPM_ALG_ECC
```
case TPM_ALG_ECC:
{
    TPM_S_ECC_POINT       eccPublic;
    TPM_S_ECC_POINT       eccSecret;
    BYTE                *buffer = secret->t.secret;
    INT32                size = secret->t.size;

    // Retrieve ECC point from secret buffer
    result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
    if(result != TPM_RC_SUCCESS)
      return result;

    result = CryptEccPointMultiply(&eccSecret,
                                   decryptKey->publicArea.parameters.eccDetail.curveID,
                                   &decryptKey->sensitive.sensitive.ecc,
                                   &eccPublic);
    if(result != TPM_RC_SUCCESS)
      return result;

    // Set the size of the "recovered" secret value to be the size of the digest
    // produced by the nameAlg.
    data->t.size = CryptGetHashDigestSize(decryptKey->publicArea.nameAlg);

    // The secret value is computed from Z using KDFe as:
    // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
    // Where:
    // HashID  the nameAlg of the decrypt key
    // Z      the x coordinate (Px) of the product (P) of the point (Q) of
    //        the secret and the private x coordinate (de,V) of the
    //        decryption key
    // Use a null-terminated string containing "SECRET"
    // PartyUInfo  the x coordinate of the point in the secret (Qe,U )
    // PartyVInfo  the x coordinate of the public key (Qs,V )
    // bits     the number of bits in the digest of HashID
    // Retrieve seed from KDFe
    CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
               &eccPublic.x.b,
               &decryptKey->publicArea.unique.ecc.x.b,
               data->t.size * 8, data->t.buffer);
    } break;
#endif
}
secret->t.size, secret->t.secret);

// Copy decrypted seed
MemoryCopy2B(&data->b, &secret->b);
break;

}
case TPM_ALG_SYMCIPHER:
{
TPM2B_IV        iv = {0};
TPMT_SYM_DEF_OBJECT *symDef;

// The seed size can not be bigger than the digest size of nameAlg
if(secret->t.size >
    CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
    return TPM_RC_VALUE;

symDef = &decryptKey->publicArea.parameters.symDetail;

iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
                                        symDef->keyBits.sym);
pAssert(iv.t.size != 0);

if(nonceCaller->t.size >= iv.t.size)
    MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
else
    MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
               nonceCaller->t.size);

// CFB decrypt in place, using nonceCaller as iv
CryptSymmetricDecrypt(secret->t.secret, symDef->algorithm,
                       symDef->keyBits.sym, TPM_ALG_CFB,
                       decryptKey->sensitive.sensitive.sym.t.buffer,
                       &iv, secret->t.size, secret->t.secret);

// Copy decrypted seed
MemoryCopy2B(&data->b, &secret->b);
break;
default:
    return TPM_RC_KEY;
break;
}

return TPM_RC_SUCCESS;

9.16.10.8 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

TPM_RC
CryptParameterEncryption(
    TPM_HANDLE           handle, // IN: encrypt session handle
    TPM2B               *nonceCaller, // IN: nonce caller
    UINT16               leadingSizeInByte, // IN: the size of the leading size
    TPM2B_AUTH          *extraKey, // IN: additional key material other
    BYTE                *buffer // IN/OUT: parameter buffer to be
                           // encrypted
)

    SESSION *session = SessionGet(handle); // encrypt session
    TPM2B_TYPE(SYM_KEY, (sizeof(extraKey->t.buffer) * 2));
TPM2B_SYM_KEY key;  // encryption key
UINT32 cipherSize = 0;  // size of cipher text

pAssert((session->sessionKey.t.size + extraKey->t.size) <= <K>sizeof(key.t.buffer));

// Retrieve encrypted data size.
if(leadingSizeInByte == 2)
{
    // Extract the first two bytes as the size field as the data size
    // encrypt
    cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
    // advance the buffer
    buffer = &buffer[2];
}
else if(leadingSizeInByte == 4)
{
    // use the first four bytes to indicate the number of bytes to encrypt
    cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
    // advance pointer
    buffer = &buffer[4];
}
else
{
    pAssert(FALSE);
}

// Compute encryption key by concatenating sessionAuth with extra key
MemoryCopy2B(&key.b, &session->sessionKey.b);
MemoryConcat2B(&key.b, &extraKey->b);

if (session->symmetric.algorithm == TPM_ALG_XOR)

    return CryptXORObfuscation(session->authHashAlg, &(key.b),
        &session->nonceTPM.b),
        nonceCaller, cipherSize, buffer);
else
    return ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
        session->symmetric.keyBits.aes, &(key.b),
        nonceCaller, &session->nonceTPM.b),
        cipherSize, buffer);

9.16.10.9  CryptParameterDecryption()

This function does in-place decryption of a command parameter.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmarshal errors</td>
<td>if input buffer is in wrong canonical format</td>
</tr>
</tbody>
</table>

TPM_RC

CryptParameterDecryption(
    TPM_HANDLE handle,  // IN: encrypted session handle
    TPM2B *nonceCaller,  // IN: nonce caller
    UINT32 bufferSize,  // IN: size of parameter buffer
    UINT16 leadingSizeInByte,  // IN: the size of the leading size field in byte
    TPM2B_AUTH *extraKey,  // IN: the authValue
    BYTE *buffer  // IN/OUT: parameter buffer to be decrypted
)

SESSION *session = SessionGet(handle); // encrypt session

// The hmac key is going to be the concatenation of the session key and any
// additional key material (like the authValue). The size of both of these
// is the size of the buffer which can contain a TPMT_HA.
TPM2B_TYPE(HMAC_KEY, sizeof(extraKey->t.buffer));
TPM2B_HMAC_KEY key; // decryption key

UINT32 cipherSize = 0; // size of cipher text

// Retrieve encrypted data size.
if(leadingSizeInByte == 2)
{
    // The first two bytes of the buffer are the size of the
data to be decrypted
    cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
    buffer = &buffer[2]; // advance the buffer
}
else if(leadingSizeInByte == 4)
{
    // the leading size is four bytes so get the four byte size field
    cipherSize = BYTE_ARRAY_TOUINT32(buffer);
    buffer = &buffer[4]; // advance pointer
}
else
{
    pAssert(FALSE);
}
if(cipherSize > bufferSize)
    return TPM_RC_SIZE;

// Compute decryption key by concatenating sessionAuth with extra input key
MemoryCopy2B(&key.b, &session->sessionKey.b);
MemoryConcat2B(&key.b, &extraKey->b);

if(session->symmetric.algorithm == TPM_ALG_XOR)
    // XOR parameter decryption formulation:
    // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
    // Call XOR obfuscation function
    return CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
    &session->nonceTPM.b, cipherSize, buffer);
else
    // Assume that it is one of the symmetric block ciphers.
    return ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
    session->symmetric.keyBits.sym, &key.b, nonceCaller, &session->nonceTPM.b,
    cipherSize, buffer);

9.16.10.10 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

void CryptComputeSymmetricUnique(
    TPMI_ALG_HASH nameAlg, // IN: object name algorithm
    TPM2B_SENSITIVE *sensitive, // IN: sensitive area
    TPM2B_DIGEST *unique // OUT: unique buffer
)
{
    HASH_STATE hashState;
    pAssert(sensitive != NULL || unique != NULL);
// Compute the public value as the hash of sensitive.symkey || unique.buffer
unique->t.size = CryptGetHashDigestSize(nameAlg);
CryptStartHash(nameAlg, &hashState);

// Add obfuscation value
CryptUpdateDigest2B(&hashState, &sensitive->seedValue.b);

// Add sensitive value
CryptUpdateDigest2B(&hashState, &sensitive->sensitive.any.b);
CryptCompleteHash2B(&hashState, &unique->b);
return;
}

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.

void CryptComputeSymValue(
    TPM_HANDLE parentHandle, // IN: parent handle of the
    // object to be created
    TPMT_PUBLIC *publicArea, // IN/OUT: the public area template
    TPMT_SENSITIVE *sensitive, // IN: sensitive area
    TPM2B_SEED *seed, // IN: the seed
    TPMI_ALG_HASH hashAlg, // IN: hash algorithm for KDFa
    TPM2B_NAME *name // IN: object name
) {
    TPM2B_AUTH *proof = NULL;
    if(CryptIsAsymAlgorithm(publicArea->type))
        { // Generate seedValue only when an asymmetric key is a storage key
            if(publicArea->objectAttributes.decrypt == SET
                && publicArea->objectAttributes.restricted == SET)
                {
                // If this is a primary object in the endorsement hierarchy, use
                // ehProof in the creation of the symmetric seed so that child
                // objects in the endorsement hierarchy are voided on TPM2_Clear()
                // or TPM2_ChangeEPS()
                if( parentHandle == TPM_RH_ENDORSEMENT
                    && publicArea->objectAttributes.fixedTPM == SET)
                proof = &gp.ehProof;
            }
            else
                { sensitive->seedValue.t.size = 0;
                return;
            }
        }
    else
        { sensitive->seedValue.t.size = 0;
            return;
        }

    // For all the object type, the size of seedValue is the digest size of nameAlg
    sensitive->seedValue.t.size = CryptGetHashDigestSize(publicArea->nameAlg);
    // Compute seedValue using KDFa
    CryptKDFa(hashAlg,
        &seed->b,
        "seedValue", // This string is a vendor-specific information
        &name->b, // computed from the public template
proof,
sensitive->seedValue.t.size * 8,
sensitive->seedValue.t.buffer, NULL);

return;
}

9.16.10.12 CryptCreateObject()

This function creates an object. It:

a) fills in the created key in public and sensitive area;
b) creates a random number in sensitive area for symmetric keys; and

c) compute the unique id in public area for symmetric keys.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_KEY_SIZE</td>
<td>key size in the public area does not match the size in the sensitive creation area for a symmetric key</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>sensitive data size is larger than allowed for the scheme for a keyed hash object</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>exponent is not prime or could not find a prime using the provided parameters for an RSA key; unsupported name algorithm for an ECC key</td>
</tr>
</tbody>
</table>

TPM_RC

CryptCreateObject(

TPM_HANDLE               parentHandle,         // IN/OUT: indication of the
// seed source
TPMT_PUBLIC             *publicArea,          // IN/OUT: public area
TPMS_SENSITIVE_CREATE   *sensitiveCreate,    // IN: sensitive creation
TPMT_SENSITIVE          *sensitive          // OUT: sensitive area
)

// Next value is a placeholder for a random seed that is used in
// key creation when the parent is not a primary seed. It has the same
// size as the primary seed.

TPM2B_SEED       localSeed;     // data to seed key creation if this is not a primary seed

TPM2B_SEED       *seed = NULL;

TPM2B_NAME       name;

TPM_ALG_ID       hashAlg = CONTEXT_INTEGRITY_HASH_ALG;

OBJECT          *parent;

UINT32           counter;

// Set the sensitive type for the object
sensitive->sensitiveType = publicArea->type;
ObjectComputeName(publicArea, &name);

// For all objects, copy the initial auth data
sensitive->authValue = sensitiveCreate->userAuth;

// If this is a permanent handle assume that it is a hierarchy
if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
{
    seed = HierarchyGetPrimarySeed(parentHandle);
}
else


```c
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  

```c
2281 BOOL CryptObjectIsPublicConsistent(
2282     TPMT_PUBLIC *publicArea    // IN: public area
2283 )
2284 {
2285     switch (publicArea->type)
2286     {
2287         #ifdef TPM_ALG_RSA
2288             case TPM_ALG_RSA:
2289                 // RSA key size validation is handled by unmarshal process. No further
2290                 // check is needed at this point.
2291                 break;
2292         #endif //TPM_ALG_RSA
2293         #ifdef TPM_ALG_ECC
2294             case TPM_ALG_ECC:
2295                 const ECC_CURVE *curveValue;
2296                 // Check that the public point is on the indicated curve.
2297                 if(!CryptEccIsPointOnCurve(publicArea->parameters.eccDetail.curveID,
2298                    &publicArea->unique.ecc))
2299                     return FALSE;
2300                 curveValue = CryptEccGetCurveDataPointer(
2301                    publicArea->parameters.eccDetail.curveID);
2302                 // The input ECC curve must be a supported curve
2303                 pAssert(curveValue != NULL);
2304                 if(curveValue->sign.scheme != TPM_ALG_NULL
2305                     & publicArea->parameters.eccDetail.scheme.scheme !=
2306                     curveValue->sign.scheme)
2307                     return FALSE;
2308             }
2309         #endif //TPM_ALG_ECC
2310         default:
2311             // Symmetric object common checks
2312             // There is noting to check with a symmetric key that is public only. Also
2313             // not sure that there is anything useful to be done with it either.
2314             return TRUE;
2315     }
2316     // Asymmetric stuff falls through and is checked for consistent key sizes in
2317     // the public area
2318     if(!CryptAreKeySizesConsistent(publicArea))
2319         return FALSE;
2320     return TRUE;
2321 }
2322 9.16.10.14 CryptObjectPublicPrivateMatch()
2323
2324 This function checks the cryptographic binding between the public and sensitive areas.
```
<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_TYPE</td>
<td>the type of the public and private areas are not the same</td>
</tr>
<tr>
<td>TPM_RC_FAILURE</td>
<td>crypto error</td>
</tr>
<tr>
<td>TPM_RC_BINDING</td>
<td>the public and private areas are not cryptographically matched.</td>
</tr>
</tbody>
</table>

```c
2328 TPM_RC
2329 CryptObjectPublicPrivateMatch(
2330     OBJECT *object      // IN: the object to check
2331 )
2332 {
2333     TPMT_PUBLIC    *publicArea;
2334     TPMT_SENSITIVE *sensitive;
2335     pAssert(object != NULL);
2336     publicArea = &object->publicArea;
2337     sensitive = &object->sensitive;
2338     if(publicArea->type != sensitive->sensitiveType)
2339         return TPM_RC_TYPE;
2340     switch(publicArea->type)
2341     {
2342         #ifdef TPM_ALG_RSA
2343         case TPM_ALG_RSA:
2344             // The public and private key sizes need to be consistent
2345             if(sensitive->sensitive.rsa.t.size != publicArea->unique.rsa.t.size/2)
2346                 return TPM_RC_BINDING;
2347             // Load key by computing the private exponent
2348             return CryptLoadPrivateRSA(object);
2349             break;
2350         #endif
2351         #ifdef TPM_ALG_ECC
2352         case TPM_ALG_ECC:
2353             // This function is called from ObjectLoad() which has already checked to
2354             // see that the public point is on the curve so no need to repeat that
2355             // check.
2356             if(publicArea->nameAlg != TPM_ALG_NULL)
2357                 {  
2358                     TPMS_ECC_POINT   publicToCompare;
2359                     // Compute ECC public key
2360                     CryptEccPointMultiply(&publicToCompare,
2361                                       publicArea->parameters.eccDetail.curveID,
2362                                       &sensitive->sensitive.ecc, NULL);
2363                     // Compare ECC public key
2364                     if( (!Memory2BEqual(&publicArea->unique.ecc.x.b,
2365                                        &publicToCompare.x.b))
2366                         || (!Memory2BEqual(&publicArea->unique.ecc.y.b,
2367                                        &publicToCompare.y.b)))
2368                         return TPM_RC_BINDING;
2369                     }
2370                     return TPM_RC_SUCCESS;
2371                     break;
2372         #endif
2373         case TPM_ALG_KEYEDHASH:
2374             break;
2375         case TPM_ALG_SYMCIPHER:
2376             if( (publicArea->parameters.symDetail.keyBits.sym + 7)/8
2377                   != sensitive->sensitive.sym.t.size)
2378                 return TPM_RC_BINDING;
2379             break;
```
default:
    // The choice here is an assert or a return of a bad type for the object
    return TPM_RC_TYPE;
    break;

// For asymmetric keys, the algorithm for validating the linkage between
// the public and private areas is algorithm dependent. For symmetric keys
// the linkage is based on hashing the symKey and obfuscation values.
if (publicArea->nameAlg != TPM_ALG_NULL)
{
    TPM2B_DIGEST uniqueToCompare;

    // Compute unique for symmetric key
    CryptComputeSymmetricUnique(publicArea->nameAlg, sensitive,
    &uniqueToCompare);

    // Compare unique
    if (!Memory2BEqual(&publicArea->unique.sym.b,
    &uniqueToCompare.b))
        return TPM_RC_BINDING;
}
return TPM_RC_SUCCESS;

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

```c
TPMI_ALG_HASH
CryptGetSignHashAlg(  
    TPMT_SIGNATURE      *auth, // IN: signature
)
{
    pAssert(auth->sigAlg != TPM_ALG_NULL);

    // Get authHash algorithm based on signing scheme
    switch(auth->sigAlg)
    {
        #ifdef TPM_ALG_RSA
            case TPM_ALG_RSASSA:
                return auth->signature.rsassa.hash;
            
            case TPM_ALG_RSAPSS:
                return auth->signature.rsapss.hash;
        #endif //TPM_ALG_RSA

        #ifdef TPM_ALG_ECC
            case TPM_ALG_ECDSA:
                return auth->signature.ecdsa.hash;
        #endif //TPM_ALG_ECC

        case TPM_ALG_HMAC:
            return auth->signature.hmac.hashAlg;

        default:
            return TPM_ALG_NULL;
    }
```
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().

```c
BOOL CryptIsSplitSign(
    TPM_ALG_ID scheme // IN: the algorithm selector
)
{
    if( scheme != scheme
        # ifdef TPM_ALG_ECDAA
            || scheme == TPM_ALG_ECDAA
        # endif // TPM_ALG_ECDAA
    )
        return TRUE;
    return FALSE;
}
```

9.16.10.17 CryptIsSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```c
BOOL CryptIsSignScheme(
    TPM_ALG_ASYM_SCHEME scheme
)
{
    switch(scheme)
    {
        #ifdef TPM_ALG_RSA
            // If RSA is implemented, then both signing schemes are required
            case TPM_ALG_RSASSA:
                return TRUE;
                break;
        #endif //TPM_ALG_RSA
        #ifdef TPM_ALG_ECC
            // If ECC is implemented ECDSA is required
            case TPM_ALG_ECDSA:
                // ECDA is optional
            case TPM_ALG_ECDAA:
                #endif
            #ifdef TPM_ALG_ECSCHNORR
                // Schnorr is also optional
            case TPM_ALG_ECSCHNORR:
                #endif
            #ifdef TPM_ALG_SM2
                return TRUE;
                break;
            #endif //TPM_ALG_ECC
            default:
                return FALSE;
                break;
        }
    }
```
9.16.10.18 CryptIsDecryptScheme()

This function indicates if a scheme algorithm is a decrypt algorithm.

```c
BOOL CryptIsDecryptScheme(
    TPMI_ALGASYM_SCHEME scheme
) {
    switch(scheme) {
        #ifdef TPM_ALG_RSA
            // If RSA is implemented, then both decrypt schemes are required
            case TPM_ALG_RSAES:
            case TPM_ALG_OAEP:
                return TRUE;
                break;
            #endif //TPM_ALG_RSA

        #ifdef TPM_ALG_ECC
            // If ECC is implemented ECDH is required
            case TPM_ALG_ECDH:
                #ifdef TPM_ALG_SM2
                    case TPM_ALG_SM2:
                    #endif
                    #ifdef TPM_ALG_ECMQV
                        case TPM_ALG_ECMQV:
                    #endif
                    return TRUE;
                    break;
                #endif //TPM_ALG_ECC

        default:
            return FALSE;
            break;
    }
}
```

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.

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

```c
TPM_RC
CryptSelectSignScheme(
    TPMI_DH_OBJECT signHandle, // IN: handle of signing key
    TPMT_SIG_SCHEME *scheme    // IN/OUT: signing scheme
) {
    OBJECT *signObject;
    TPMT_SIG_SCHEME *objectScheme;
```
TPMT_PUBLIC *publicArea;

// If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
// of the setting of scheme
if (signHandle == TPM_RH_NULL)
{
    scheme->scheme = TPM_ALG_NULL;
    scheme->details.any.hashAlg = TPM_ALG_NULL;
    return TPM_RC_SUCCESS;
}

// Get sign object pointer
signObject = ObjectGet(signHandle);
publicArea = &signObject->publicArea;

// is this a signing key?
if (!publicArea->objectAttributes.sign)
    return TPM_RC_KEY;

if (CryptIsAsymAlgorithm(publicArea->type))
    objectScheme = (TPMT_SIG_SCHEME *)&publicArea->parameters.asymDetail.scheme;
else
    objectScheme = (TPMT_SIG_SCHEME *)&publicArea->parameters.keyedHashDetail.scheme;

// If the object doesn't have a default scheme, then use the input scheme.
if (objectScheme->scheme == TPM_ALG_NULL)
{
    // Input and default can't both be NULL
    if (scheme->scheme == TPM_ALG_NULL)
        return TPM_RC_SCHEME;

    // Assume that the scheme is compatible with the key. If not,
    // we will generate an error in the signing operation.
    return TPM_RC_SUCCESS;
}
else if (scheme->scheme == TPM_ALG_NULL)
{
    // input scheme is NULL so use default
    // First, check to see if the default requires that the caller provide
    // scheme data
    if (CryptIsSplitSign(objectScheme->scheme))
        return TPM_RC_SCHEME;

    scheme->scheme = objectScheme->scheme;
    scheme->details.any.hashAlg = objectScheme->details.any.hashAlg;
    return TPM_RC_SUCCESS;
}

// Both input and object have scheme selectors
// If the scheme and the hash are not the same then...
if (objectScheme->scheme != scheme->scheme
    || objectScheme->details.any.hashAlg != scheme->details.any.hashAlg)
    return TPM_RC_SCHEME;

return TPM_RC_SUCCESS;
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>$signHandle$ references not a signing key</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>$signScheme$ is not compatible with the signing key type</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>digest value is greater than the modulus of $signHandle$ or size of hashData does not match hash algorithm $insignScheme$ (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)</td>
</tr>
</tbody>
</table>

```c
TPM_RC
CryptSign(
    TPMI_DH_OBJECT signHandle,    // IN: The handle of sign key
    TPM_SIG_SCHEME *signScheme,    // IN: sign scheme.
    TPM2B_DIGEST *digest,          // IN: The digest being signed
    TPM2_SIGNATURE *signature      // OUT: signature
) {
    OBJECT *signKey = ObjectGet(signHandle);
    // check if input handle is a sign key
    if (signKey->publicArea.objectAttributes.sign != SET)
        return TPM_RC_ATTRIBUTES;
    // Must have the private portion loaded. This check is made during
    // authorization.
    pAssert(signKey->attributes.publicOnly == CLEAR);
    // Initialize signature scheme
    signature->sigAlg = signScheme->scheme;
    // Initialize signature hash
    signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
    // perform sign operation based on different key type
    #ifdef TPM_ALG_RSA
        if (signKey->publicArea.type == TPM_ALG_RSA)
            // Sign it
            return CryptSignRSA(signKey, signScheme, digest, signature);
    #endif //TPM_ALG_RSA
    #ifdef TPM_ALG_ECC
        if (signKey->publicArea.type == TPM_ALG_ECC)
            // Perform the signature operation
            return CryptSignECC(signKey, signScheme, digest, signature);
    #endif //TPM_ALG_ECC
    if (signKey->publicArea.type == TPM_ALG_KEYEDHASH)
        { // Sign
            return CryptSignHMAC(signKey, signScheme, digest, signature);
        }
    pAssert(FALSE);
    return TPM_RC_ATTRIBUTES; // This is unreachable code but this makes the
```
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>the signature is not genuine</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>the scheme is not supported</td>
</tr>
</tbody>
</table>

```c
TPM_RC
CryptVerifySignature(
    TPMI_DH_OBJECT keyHandle,   // IN: The handle of sign key
    TPM2B_DIGEST *digest,       // IN: The digest being validated
    TPMT_SIGNATURE *signature   // IN: signature
) {
    OBJECT*authObject = ObjectGet(keyHandle);
    TPMT_PUBLIC*publicArea = &authObject->publicArea;

    switch (publicArea->type) {
    #ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            return CryptRSAVerifySignature(authObject, digest, signature);
            break;
    #endif // TPM_ALG_RSA
    #ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            return CryptECCVerifySignature(authObject, digest, signature);
            break;
    #endif // TPM_ALG_ECC
    case TPM_ALG_KEYEDHASH:
        return CryptHMACVerifySignature(authObject, digest, signature);
        break;
    default:
        pAssert(FALSE);
        return TPM_RC_SCHEME; // This is unreachable but it makes the compiler happy.
        break;
    }
} // compiler happy
```
9.16.11 Math functions

9.16.11.1 CryptDivide()

This function interfaces to the math library for large number divide.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIZE</td>
<td>quotient or remainder is too small to receive the result</td>
</tr>
</tbody>
</table>

```c
TPM_RC
CryptDivide(
    TPM2B *numerator, // IN: numerator
    TPM2B *denominator, // IN: denominator
    TPM2B *quotient, // OUT: quotient = numerator / denominator.
    TPM2B *remainder // OUT: numerator mod denominator.
)
```

9.16.11.2 CryptCompare()

This function interfaces to the math library for large number, unsigned compare.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#a&gt;b</td>
</tr>
<tr>
<td>0</td>
<td>#a=b</td>
</tr>
<tr>
<td>-1</td>
<td>#a&lt;b</td>
</tr>
</tbody>
</table>

```c
int
CryptCompare(
    const UINT32 a, // IN: a buffer
    const BYTE *a, // IN: size of a
    const UINT32 b, // IN: b buffer
    const BYTE *b // INPUT: size of b
)
```

```c
int borrow = 0;
int notZero = 0;
int i;
// If a has more digits than b, then a is greater than b if
// any of the more significant bytes is non zero
if((i = (int)aSize - (int)bSize) > 0)
    for(; i > 0; i--) // a++
        if(*a++) // means a > b
    return 1;
// If b has more digits than a, then b is greater if any of the
// more significant bytes is non zero
if(i < 0) <Q> // Means that b is longer than a
```
for (; i < 0; i++)
    if (*b++) // means that b > a
        return -1;

// Either the values are the same size or the upper bytes of a or b are
// all zero, so compare the rest
i = (aSize > bSize) ? bSize : aSize;
a = &a[i-1];
b = &b[i-1];
for (; i > 0; i--)
{
    borrow = *a-- - *b-- + borrow;
    notZero = notZero || borrow;
    borrow >>= 8;
}

// if there is a borrow, then b > a
if (borrow)
    return -1;

// either a > b or they are the same
return notZero;

9.16.11.3 CryptCompareSigned()

This function interfaces to the math library for large number, signed compare.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#a&gt;b</td>
</tr>
<tr>
<td>0</td>
<td>#a=b</td>
</tr>
<tr>
<td>-1</td>
<td>#a&lt;b</td>
</tr>
</tbody>
</table>

int
CryptCompareSigned(
UINT32 aSize,     // IN: size of a
BYTE *a,           // IN: a buffer
UINT32 bSize,     // IN: size of b
BYTE *b            // IN: b buffer
)
{
    int signA, signB;     // sign of a and b
    // For positive or 0, sign_a is 1
    // for negative, sign_a is 0
    signA = ((a[0] & 0x80) == 0) ? 1 : 0;
    // For positive or 0, sign_b is 1
    // for negative, sign_b is 0
    signB = ((b[0] & 0x80) == 0) ? 1 : 0;

    if(signA != signB)
    {
        return signA - signB;
    }
    if(signA == 1)
    // do unsigned compare function
    return CryptCompare(aSize, a, bSize, b);
    else
    // do unsigned compare the other way
    return 0 - CryptCompare(aSize, a, bSize, b);
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().

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_TESTING</td>
<td>if fullTest is YES</td>
</tr>
</tbody>
</table>

```
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_TESTING</td>
<td>if toTest list is not empty</td>
</tr>
</tbody>
</table>

```
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().

```
TPM_RC
CryptGetTestResult(
    TPM2B_MAX_BUFFER    *outData     // OUT: test result data
) {
    outData->t.size = 0;
    return TPM_RC_SUCCESS;
}

9.16.13 Capability Support

9.16.13.1 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if no more ECC curve is available</td>
</tr>
<tr>
<td>NO</td>
<td>if there are more ECC curves not reported</td>
</tr>
</tbody>
</table>

```
9.16.13.2 CryptCapGetEccCurveNumber()

This function returns the number of ECC curves supported by the TPM.

```c
UINT32 CryptCapGetEccCurveNumber(void)
{
    // There is an array that holds the curve data. Its size divided by the
    // size of an entry is the number of values in the table.
    return _cpri__EccGetCurveCount();
}
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>sizes are consistent</td>
</tr>
<tr>
<td>FALSE</td>
<td>sizes are not consistent</td>
</tr>
</tbody>
</table>

```c
BOOL CryptAreKeySizesConsistent(
    TPMT_PUBLIC *publicArea        // IN: the public area to check
)
{
    #ifdef TPM_ALG_RSA
    if(publicArea->type == TPM_ALG_RSA)
    {
        // The key size in bits is filtered by the unmarshaling
        return (   ((publicArea->parameters.rsaDetail.keyBits+7)/8)
                == publicArea->unique.rsa.t.size);
    }
    #endif //TPM_ALG_RSA

    #ifdef TPM_ALG_ECC
    if(publicArea->type == TPM_ALG_ECC)
    {
        UINT16    keySizeInBytes;
        TPM_ECC_Curve curveId = publicArea->parameters.eccDetail.curveID;
        keySizeInBytes = CryptEccGetKeySizeInBytes(curveId);
        return (    keySizeInBytes > 0
                    && publicArea->unique.ecc.x.t.size <= keySizeInBytes
                    && publicArea->unique.ecc.y.t.size <= keySizeInBytes);
    }
    #endif //TPM_ALG_ECC
```
9.17 Ticket.c

9.17.1 Introduction

This clause contains the functions used for ticket computations.

9.17.2 Includes

```c
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>It is safe to produce ticket</td>
</tr>
<tr>
<td>FALSE</td>
<td>It is not safe to produce ticket</td>
</tr>
</tbody>
</table>

```c
BOOL TicketIsSafe(
   TPM2B *buffer
)
{
    TPM_GENERATED valueToCompare = TPM_GENERATED_VALUE;
    BYTE bufferToCompare[sizeof(valueToCompare)];
    BYTE *marshalBuffer;
    // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume it is not safe to generate a ticket
    if(buffer->size < sizeof(valueToCompare))
        return FALSE;
    // marshalling Buffer
    marshalBuffer = bufferToCompare;
    TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);
    if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
        return FALSE;
    else
        return TRUE;
}
```

9.17.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```c
void TicketComputeVerified(
   TPMI_RH_HIERARCHY hierarchy,    // IN: hierarchy constant for ticket
   TPM2B_DIGEST *digest,           // IN: digest
   TPM2B_NAME *keyName,            // IN: name of key that signed the
```

Part 4: Supporting Routines

Trusted Platform Module Library

Page 306 Published Family "02"

March 15, 2013 Copyright © TCG 2006-2013 Level 00 Revision 00.96
9.17.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```c
void TicketComputeAuth(
    TPM_ST               type,        // IN: the type of ticket.
    TPM_RH_HIERARCHY    hierarchy,   // IN: hierarchy constant for ticket
    UINT64               timeout,     // IN: timeout
    TPM2B_DIGEST        *cpHashA,     // IN: input cpHashA
    TPM2B_NONCE         *policyRef,   // IN: input policyRef
    TPM2B_NAME          *entityName,  // IN: name of entity
    TPMT_TK_AUTH        *ticket       // OUT: Created ticket
) {

    TPM2B_AUTH          *proof;
    HMAC_STATE           hmacState;

    // Get proper proof
    proof = HierarchyGetProof(hierarchy);

    // Fill in ticket fields
    ticket->tag = type;
    ticket->hierarchy = hierarchy;

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

    // Adding TPM_ST_VERIFIED
    CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);

    // digest
    CryptUpdateDigest2B(&hmacState, &digest->b);

    // key name
    CryptUpdateDigest2B(&hmacState, &keyName->b);

    // complete HMAC
    CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);

    return;
}
```
9.17.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```c
void TicketComputeHashCheck(
    TPM_RH_HIERARCHY hierarchy,   // IN: hierarchy constant for ticket
    TPM2B_DIGEST *digest,         // IN: input digest
    TPMT_TK_HASHCHECK *ticket      // OUT: Created ticket
) {
    TPM2B_AUTH *proof;
    HMAC_STATE hmacState;
    // Get proper proof
    proof = HierarchyGetProof(hierarchy);
    // Fill in ticket fields
    ticket->tag = TPM_ST_HASHCHECK;
    ticket->hierarchy = hierarchy;
    ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
                                           &proof->b, &hmacState);
    // Add TPM_ST_HASHCHECK
    CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
    // Add digest
    CryptUpdateDigest2B(&hmacState, &digest->b);
    // Compute HMAC
    CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
    return;
}
```

9.17.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```c
void TicketComputeCreation(
    TPM_RH_HIERARCHY hierarchy,   // IN: hierarchy for ticket
    TPM2B_NAME *name,             // IN: object name
    TPM2B_DIGEST *digest,         // IN: input digest
    TPMT_TK_CREATION *ticket      // OUT: Created ticket
) {
    // Get proper proof
    TPM2B_AUTH *proof;
    HMAC_STATE hmacState;
    // Fill in ticket fields
    ticket->tag = TPM_ST_CREATION;
    ticket->hierarchy = hierarchy;
    ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
                                           &proof->b, &hmacState);
    // Add TPM_ST_CREATION
    CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
    // Add digest
    CryptUpdateDigest2B(&hmacState, &digest->b);
    // Compute HMAC
    CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
    return;
}
```
TPM2B_DIGEST *creation, // IN: creation hash
TPMT_TK_CREATION *ticket // OUT: created ticket

TPM2B_AUTH *proof;
HMAC_STATE hmacState;

// Get proper proof
proof = HierarchyGetProof(hierarchy);

// Fill in ticket fields
ticket->tag = TPM_ST_CREATION;
ticket->hierarchy = hierarchy;

// Get proper proof
proof = HierarchyGetProof(hierarchy);

// Fill in ticket fields
ticket->tag = TPM_ST_CREATION;
ticket->hierarchy = hierarchy;

ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
    &proof->b, &hmacState);

// Add TPM_ST_CREATION
CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);

// Add name
CryptUpdateDigest2B(&hmacState, &name->b);

// Add creation hash
CryptUpdateDigest2B(&hmacState, &creation->b);

// Compute HMAC
CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);

return;
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

```c
#ifndef _IMPLEMENTATION_H
#define _IMPLEMENTATION_H

#ifndef ALG_ALL
#define ALG_ALL NO
#endif

#include "BaseTypes.h"

#ifdef TRUE
#undef TRUE
#endif

#ifdef FALSE
#undef FALSE
#endif

#define SHA1_DIGEST_SIZE 20
#define SHA1_BLOCK_SIZE 64
#define SHA1_DER_SIZE 15
#define SHA1_DER {
  0x30,0x21,0x30,0x09,0x06,0x05,0x2B,0x0E,0x03,0x02,0x1A,0x05,0x00,0x04,0x14}

#define SHA256_DIGEST_SIZE 32
#define SHA256_BLOCK_SIZE 64
#define SHA256_DER_SIZE 19
#define SHA256_DER {
  0x30,0x31,0x30,0x0d,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x01,\n  0x05,0x00,0x04,0x20}

#define SHA384_DIGEST_SIZE 48
#define SHA384_BLOCK_SIZE 128
#define SHA384_DER_SIZE 19
#define SHA384_DER {
  0x30,0x41,0x30,0x0c,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x02,\n  0x05,0x00,0x04,0x30}

#define SHA512_DIGEST_SIZE 64
#define SHA512_BLOCK_SIZE 128
#define SHA512_DER_SIZE 19
#define SHA512_DER {
  0x30,0x51,0x30,0x0c,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x03,\n  0x05,0x00,0x04,0x40}

#define SM3_256_DIGEST_SIZE 32
#define SM3_256_BLOCK_SIZE 64
#define SM3_256_DER_SIZE 18
#define SM3_256_DER {
  0x30,0x30,0x30,0x0c,0x06,0x08,0x2a,0x81,0x1c,0x81,0x45,0x01,0x83,0x11,0x05,\n  0x00,0x04,0x20}
```

#### Table 205 -- SHA1 Hash Values

- Define `SHA1_DIGEST_SIZE` as 20
- Define `SHA1_BLOCK_SIZE` as 64
- Define `SHA1_DER_SIZE` as 15
- Define `SHA1_DER` as a byte array starting with `0x30,0x21,0x30,0x09,0x06,0x05,0x2B,0x0E,0x03,0x02,0x1A,0x05,0x00,0x04,0x14`

#### Table 206 -- SHA256 Hash Values

- Define `SHA256_DIGEST_SIZE` as 32
- Define `SHA256_BLOCK_SIZE` as 64
- Define `SHA256_DER_SIZE` as 19
- Define `SHA256_DER` as a byte array starting with `0x30,0x31,0x30,0x0d,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x01,0x05,0x00,0x04,0x20`

#### Table 207 -- SHA384 Hash Values

- Define `SHA384_DIGEST_SIZE` as 48
- Define `SHA384_BLOCK_SIZE` as 128
- Define `SHA384_DER_SIZE` as 19
- Define `SHA384_DER` as a byte array starting with `0x30,0x41,0x30,0x0c,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x02,0x05,0x00,0x04,0x30`

#### Table 208 -- SHA512 Hash Values

- Define `SHA512_DIGEST_SIZE` as 64
- Define `SHA512_BLOCK_SIZE` as 128
- Define `SHA512_DER_SIZE` as 19
- Define `SHA512_DER` as a byte array starting with `0x30,0x51,0x30,0x0c,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x03,0x05,0x00,0x04,0x40`

#### Table 210 -- SM3_256 Hash Values

- Define `SM3_256_DIGEST_SIZE` as 32
- Define `SM3_256_BLOCK_SIZE` as 64
- Define `SM3_256_DER_SIZE` as 18
- Define `SM3_256_DER` as a byte array starting with `0x30,0x30,0x30,0x0c,0x06,0x08,0x2a,0x81,0x1c,0x81,0x45,0x01,0x83,0x11,0x05,0x00,0x04,0x20`
Table 211 -- Architectural Limits Values

<table>
<thead>
<tr>
<th>Line</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td><code>#define MAX_SESSION_NUMBER</code></td>
<td>3</td>
</tr>
</tbody>
</table>

Table 213 -- Logic Values

<table>
<thead>
<tr>
<th>Line</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td><code>#define YES</code></td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td><code>#define NO</code></td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td><code>#define TRUE</code></td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td><code>#define FALSE</code></td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td><code>#define SET</code></td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td><code>#define CLEAR</code></td>
<td>0</td>
</tr>
</tbody>
</table>

Table 214 -- Processor Values

<table>
<thead>
<tr>
<th>Line</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td><code>#define BIG_ENDIAN_TPM</code></td>
<td>NO</td>
</tr>
<tr>
<td>50</td>
<td><code>#define LITTLE_ENDIAN_TPM</code></td>
<td>YES</td>
</tr>
<tr>
<td>51</td>
<td><code>#define NO_AUTOALIGN</code></td>
<td>NO</td>
</tr>
</tbody>
</table>

Table 215 -- Implemented Algorithms

<table>
<thead>
<tr>
<th>Line</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td><code>#define ALG_RSA</code></td>
<td>YES</td>
</tr>
<tr>
<td>53</td>
<td><code>#define ALG_SHA1</code></td>
<td>YES</td>
</tr>
<tr>
<td>54</td>
<td><code>#define ALG_HMAC</code></td>
<td>YES</td>
</tr>
<tr>
<td>55</td>
<td><code>#define ALG_AES</code></td>
<td>YES</td>
</tr>
<tr>
<td>56</td>
<td><code>#define ALG_MGF1</code></td>
<td>YES</td>
</tr>
<tr>
<td>57</td>
<td><code>#define ALG_XOR</code></td>
<td>YES</td>
</tr>
<tr>
<td>58</td>
<td><code>#define ALG_KEYEDHASH</code></td>
<td>YES</td>
</tr>
<tr>
<td>59</td>
<td><code>#define ALG_SHA256</code></td>
<td>YES</td>
</tr>
<tr>
<td>60</td>
<td><code>#define ALG_SHA384</code></td>
<td>NO</td>
</tr>
<tr>
<td>61</td>
<td><code>#define ALG_SHA512</code></td>
<td>NO</td>
</tr>
<tr>
<td>62</td>
<td><code>#define ALG_SM3_256</code></td>
<td>YES</td>
</tr>
<tr>
<td>63</td>
<td><code>#define ALG_SM4</code></td>
<td>YES</td>
</tr>
<tr>
<td>64</td>
<td><code>#define ALG_RSASSA</code></td>
<td>YES</td>
</tr>
<tr>
<td>65</td>
<td><code>#define ALG_RSAES</code></td>
<td>YES</td>
</tr>
<tr>
<td>66</td>
<td><code>#define ALG_RSAPSS</code></td>
<td>YES</td>
</tr>
<tr>
<td>67</td>
<td><code>#define ALG_OAEF</code></td>
<td>YES</td>
</tr>
<tr>
<td>68</td>
<td><code>#define ALG_ECC</code></td>
<td>YES</td>
</tr>
<tr>
<td>69</td>
<td><code>#define ALG_ECDH</code></td>
<td>YES</td>
</tr>
<tr>
<td>70</td>
<td><code>#define ALG_ECDSA</code></td>
<td>YES</td>
</tr>
<tr>
<td>71</td>
<td><code>#define ALG_ECDA</code></td>
<td>YES</td>
</tr>
<tr>
<td>72</td>
<td><code>#define ALG_SM2</code></td>
<td>YES</td>
</tr>
<tr>
<td>73</td>
<td><code>#define ALG_ECSCHNORR</code></td>
<td>YES</td>
</tr>
<tr>
<td>74</td>
<td><code>#define ALG_ECMQV</code></td>
<td>NO</td>
</tr>
<tr>
<td>75</td>
<td><code>#define ALG_SYMCIPHER</code></td>
<td>YES</td>
</tr>
<tr>
<td>76</td>
<td><code>#define ALG_KDF1_SP800_56a</code></td>
<td>YES</td>
</tr>
<tr>
<td>77</td>
<td><code>#define ALG_KDF2</code></td>
<td>NO</td>
</tr>
<tr>
<td>78</td>
<td><code>#define ALG_KDF1_SP800_108</code></td>
<td>YES</td>
</tr>
<tr>
<td>79</td>
<td><code>#define ALG_CTR</code></td>
<td>YES</td>
</tr>
<tr>
<td>80</td>
<td><code>#define ALG_OFB</code></td>
<td>YES</td>
</tr>
<tr>
<td>81</td>
<td><code>#define ALG_CBC</code></td>
<td>YES</td>
</tr>
<tr>
<td>82</td>
<td><code>#define ALG_CFB</code></td>
<td>YES</td>
</tr>
<tr>
<td>83</td>
<td><code>#define ALG_ECB</code></td>
<td>YES</td>
</tr>
</tbody>
</table>

Table 216 -- Implemented Commands

<table>
<thead>
<tr>
<th>Line</th>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>84</td>
<td><code>#define CC_ActivateCredential</code></td>
<td>YES</td>
</tr>
<tr>
<td>85</td>
<td><code>#define CC_Certify</code></td>
<td>YES</td>
</tr>
<tr>
<td>86</td>
<td><code>#define CC_CertifyCreation</code></td>
<td>YES</td>
</tr>
<tr>
<td>87</td>
<td><code>#define CC_ChangeEPS</code></td>
<td>YES</td>
</tr>
<tr>
<td>88</td>
<td><code>#define CC_ChangePPS</code></td>
<td>YES</td>
</tr>
<tr>
<td>89</td>
<td><code>#define CC_Clear</code></td>
<td>YES</td>
</tr>
<tr>
<td>90</td>
<td><code>#define CC_ClearControl</code></td>
<td>YES</td>
</tr>
</tbody>
</table>
#define CC_ClockRateAdjust YES // 1
#define CC_ClockSet YES // 1
#define CC_Commit ALG_ECC // 1
#define CC_ContextLoad YES // 1
#define CC_ContextSave YES // 1
#define CC_Create YES // 1
#define CC_CreatePrimary YES // 1
#define CC_DictionaryAttackLockReset YES // 1
#define CC_DictionaryAttackParameters YES // 1
#define CC_Duplicate YES // 1
#define CC_ECC_Parameters ALG_ECC // 1
#define CC_ECDH_KeyGen ALG_ECC // 1
#define CC_ECDH_ZGen ALG_ECC // 1
#define CC_EncryptDecrypt YES // 1
#define CC_EventSequenceComplete YES // 1
#define CC_FcEvictControl YES // 1
#define CC_FieldUpgradeData NO // 0
#define CC_FieldUpgradeStart NO // 0
#define CC_FirmwareRead NO // 0
#define CC_FlushContext YES // 1
#define CC_GetCapability YES // 1
#define CC_GetCommandAuditDigest YES // 1
#define CC_GetRandom YES // 1
#define CC_GetSessionAuditDigest YES // 1
#define CC_GetTestResult YES // 1
#define CC_GetTime YES // 1
#define CC_Hash Hash YES // 1
#define CC_HashSequenceStart YES // 1
#define CC_HierarchyChangeAuth YES // 1
#define CC_HierarchyControl YES // 1
#define CC_HMAC YES // 1
#define CC_HMAC_Start YES // 1
#define CC_ImplmentSelfTest YES // 1
#define CC_Load YES // 1
#define CC_LoadExternal YES // 1
#define CC_LoadCredential YES // 1
#define CC_Nv_Certify YES // 1
#define CC_Nv_ChangeAuth YES // 1
#define CC_Nv_DefineSpace YES // 1
#define CC_Nv_Extend YES // 1
#define CC_Nv_GlobalWriteLock YES // 1
#define CC_Nv_Increment YES // 1
#define CC_Nv_Read YES // 1
#define CC_Nv_ReadLock YES // 1
#define CC_Nv_Public YES // 1
#define CC_Nv_SetBits YES // 1
#define CC_Nv.UndefineSpace YES // 1
#define CC_Nv.UndefineSpaceSpecial YES // 1
#define CC_Nv_Write YES // 1
#define CC_Nv_WriteLock YES // 1
#define CC_ObjectChangeAuth YES // 1
#define CC_PCR_Allocate YES // 1
#define CC_PCR_Event YES // 1
#define CC_PCR_Extend YES // 1
#define CC_PCR_Read YES // 1
#defined CC_PCR_Reset YES // 1
#defined CC_PCR_SetAuthPolicy YES // 1
#defined CC_PCR_SetAuthValue YES // 1
#defined CC_PolicyAuthorize YES // 1
#defined CC_PolicyAuthorizeValue YES // 1
#defined CC_PolicyAuthCode YES // 1
#defined CC_PolicyCounterTimer YES // 1
#defined CC_PolicyCrypt YES // 1
#defined CC_PolicyDuplicationSelect YES // 1
#defined CC_PolicyDuplicationSelect YES // 1
#defined CC_PolicyGetDigest YES // 1
#define CC_PolicyLocality YES // 1
#define CC_PolicyNameHash YES // 1
#define CC_PolicyNV YES // 1
#define CC_PolicyOR YES // 1
#define CC_PolicyPassword YES // 1
#define CC_PolicyPCR YES // 1
#define CC_PolicyPhysicalPresence YES // 1
#define CC_PolicyRestart YES // 1
#define CC_PolicySecret YES // 1
#define CC_PolicySigned YES // 1
#define CC_PolicyTicket YES // 1
#define CC_PP_Commands YES // 1
#define CC_Quote YES // 1
#define CC_ReadClock YES // 1
#define CC_ReadPublic YES // 1
#define CC_Rewrap YES // 1
#define CC_RSA_Decrypt ALG_RSA // 1
#define CC_RSA_Encrypt ALG_RSA // 1
#define CC_SelfTest YES // 1
#define CC_SequenceComplete YES // 1
#define CC_SequenceUpdate YES // 1
#define CC_SetAlgorithmSet YES // 1
#define CC_SetCommandCodeAuditStatus YES // 1
#define CC_SetPrimaryPolicy YES // 1
#define CC_Shutdown YES // 1
#define CC_Sign YES // 1
#define CC_StartAuthSession YES // 1
#define CC_Startup YES // 1
#define CC_StirRandom YES // 1
#define CC_TestParms YES // 1
#define CC_Unseal YES // 1
#define CC_VerifySignature YES // 1
#define CC_ZGen_2Phase YES // 1
#define CC_EC_Ephemeral YES // 1

Table 217 -- RSA Algorithm Constants

#define RSA_KEY_SIZES_BITS {1024, 2048} // {1024,2048}
#define MAX_RSA_KEY_BITS 2048
#define MAX_RSA_KEY_BYTES ((MAX_RSA_KEY_BITS + 7) / 8) // 256

Table 218 -- ECC Algorithm Constants

#define ECC_CURVES {\nTPM_ECC_NIST_P256, TPM_ECC_BN_P256, TPM_ECC_SM2_P256\}
#define ECC_KEY_SIZES_BITS {256}
#define MAX_ECC_KEY_BITS 256
#define MAX_ECC_KEY_BYTES ((MAX_ECC_KEY_BITS + 7) / 8) // 32

Table 219 -- AES Algorithm Constants

#define AES_KEY_SIZES_BITS {128}
#define MAX_AES_KEY_BITS 128
#define MAX_AES_BLOCK_SIZE_BYTES 16
#define MAX_AES_KEY_BYTES ((MAX_AES_KEY_BITS + 7) / 8) // 16

Table 220 -- SM4 Algorithm Constants

#define SM4_KEY_SIZES_BITS {128}
#define MAX_SM4_KEY_BITS 128
#define MAX_SM4_BLOCK_SIZE_BYTES 16
#define MAX_SM4_KEY_BYTES ((MAX_SM4_KEY_BITS + 7) / 8) // 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 PRIMARY_SEED_SIZE 32
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_AES
250 #define ALG_ID_LAST TPM_ALG_ECC
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_VEND_SPECIFIC_BYTES 0
258 #define MAX_RSA_KEY_BYTES (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
# undef MAX_DIGEST_SIZE
#define MAX_DIGEST_SIZE SHA1_DIGEST_SIZE
#endif
#if (SHA256_BLOCK_SIZE * ALG_SHA256) > MAX_HASH_BLOCK_SIZE
#endif
#define MAX_HASH_BLOCK_SIZE SHA256_BLOCK_SIZE
#if (SHA256_DIGEST_SIZE * ALG_SHA256) > MAX_DIGEST_SIZE
#endif
#define MAX_DIGEST_SIZE SHA256_DIGEST_SIZE
#if (SHA384_BLOCK_SIZE * ALG_SHA384) > MAX_HASH_BLOCK_SIZE
#endif
#undef MAX_HASH_BLOCK_SIZE
#define MAX_HASH_BLOCK_SIZE SHA384_BLOCK_SIZE
#if (SHA384_DIGEST_SIZE * ALG_SHA384) > MAX_DIGEST_SIZE
#endif
#undef MAX_DIGEST_SIZE
#define MAX_DIGEST_SIZE SHA384_DIGEST_SIZE
#if (SHA512_BLOCK_SIZE * ALG_SHA512) > MAX_HASH_BLOCK_SIZE
#endif
#undef MAX_HASH_BLOCK_SIZE
#define MAX_HASH_BLOCK_SIZE SHA512_BLOCK_SIZE
#if (SHA512_DIGEST_SIZE * ALG_SHA512) > MAX_DIGEST_SIZE
#endif
#undef MAX_DIGEST_SIZE
#define MAX_DIGEST_SIZE SHA512_DIGEST_SIZE
#if (SM3_256_BLOCK_SIZE * ALG_SM3_256) > MAX_HASH_BLOCK_SIZE
#endif
#undef MAX_HASH_BLOCK_SIZE
#define MAX_HASH_BLOCK_SIZE SM3_256_BLOCK_SIZE
#if (SM3_256_DIGEST_SIZE * ALG_SM3_256) > MAX_DIGEST_SIZE
#endif
#undef MAX_DIGEST_SIZE
#define MAX_DIGEST_SIZE SM3_256_DIGEST_SIZE
#define HASH_COUNT (ALG_SHA1+ALG_SHA256+ALG_SHA384+ALG_SHA512+ALG_SM3_256)

Table 7 -- TPM_ALG_ID Constants <l/O,S>

typedef UINT16 TPM_ALG_ID;
#define TPM_ALG_ERROR (TPM_ALG_ID)(0x0000) // a: ; D:
#define TPM_ALG_FIRST (TPM_ALG_ID)(0x0001) // a: ; D:
#define TPM_ALG_RSA (TPM_ALG_ID)(0x0001) // a: A O; D:
#define TPM_ALG_SHA (TPM_ALG_ID)(0x0004) // a: H S O; D:
#define TPM_ALG_SHA1 (TPM_ALG_ID)(0x0004) // a: H D:
#define TPM_ALG_SHA512 (TPM_ALG_ID)(0x0004) // a: D:
#define TPM_ALG_HMAC (TPM_ALG_ID)(0x0005) // a: H X D:
#define TPM_ALG_AES (TPM_ALG_ID)(0x0006) // a: S D:
#define TPM_ALG_MGF1 (TPM_ALG_ID)(0x0007) // a: H M D:
#define TPM_ALG_KEYEDHASH (TPM_ALG_ID)(0x0008) // a: H E X O; D:
#define TPM_ALG_XOR (TPM_ALG_ID)(0x0008) // a: Y O; D:
```c
#define TPM_ALG_XOR (TPM_ALG_ID)(0x000A)  // a: H S; D:
#endif
#define TPM_ALG_SHA256 (TPM_ALG_ID)(0x000B)  // a: H; D:
#endif
#define TPM_ALG_SHA384 (TPM_ALG_ID)(0x000C)  // a: H; D:
#endif
#define TPM_ALG_SHA512 (TPM_ALG_ID)(0x000D)  // a: H; D:
#endif
#define TPM_ALG_NULL (TPM_ALG_ID)(0x0010)  // a: ; D:
#endif
#define TPM_ALG_SM3_256 (TPM_ALG_ID)(0x0012)  // a: H; D:
#endif
#define TPM_ALG_SM4 (TPM_ALG_ID)(0x0013)  // a: S; D:
#endif
#define TPM_ALG_RSASSA (TPM_ALG_ID)(0x0014)  // a: A X; D: RSA
#endif
#define TPM_ALG_RSAES (TPM_ALG_ID)(0x0015)  // a: A E; D: RSA
#endif
#define TPM_ALG_RSAPSS (TPM_ALG_ID)(0x0016)  // a: A X; D: RSA
#endif
#define TPM_ALG_OAEP (TPM_ALG_ID)(0x0017)  // a: A E; D
#endif
#define TPM_ALG_ECC (TPM_ALG_ID)(0x0023)  // a: A O; D:
#endif
#define TPM_ALG_SYMCIPHER (TPM_ALG_ID)(0x0025)  // a: O; D:
#endif
#define TPM_ALG_CTR (TPM_ALG_ID)(0x0040)  // a: S E; D:
```
if ALG_OFB == YES || ALG_ALL == YES
#define TPM_ALG_OFB               (TPM_ALG_ID)(0x0041)   // a: S E; D:
#endif

if ALG_CBC == YES || ALG_ALL == YES
#define TPM_ALG_CBC               (TPM_ALG_ID)(0x0042)   // a: S E; D:
#endif

if ALG_CFB == YES || ALG_ALL == YES
#define TPM_ALG_CFB               (TPM_ALG_ID)(0x0043)   // a: S E; D:
#endif

if ALG_ECB == YES || ALG_ALL == YES
#define TPM_ALG_ECB               (TPM_ALG_ID)(0x0044)   // a: S E; D:
#endif
#define TPM_ALG_LAST              (TPM_ALG_ID)(0x0044)   // a: ; D:

typedef UINT16 TPM_ECC_CURVE;
#define TPM_ECC_NONE         (TPM_ECC_CURVE)(0x0000)
#define TPM_ECC_NIST_P192    (TPM_ECC_CURVE)(0x0001)
#define TPM_ECC_NIST_P224    (TPM_ECC_CURVE)(0x0002)
#define TPM_ECC_NIST_P256    (TPM_ECC_CURVE)(0x0003)
#define TPM_ECC_NIST_P384    (TPM_ECC_CURVE)(0x0004)
#define TPM_ECC_NIST_P521    (TPM_ECC_CURVE)(0x0005)
#define TPM_ECC_BN_P256      (TPM_ECC_CURVE)(0x0010)
#define TPM_ECC_BN_P638      (TPM_ECC_CURVE)(0x0011)
#define TPM_ECC_SM2_P256     (TPM_ECC_CURVE)(0x0020)
#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

```c
#include <string.h>
#include <openssl/aes.h>
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <openssl/ec.h>
#include <openssl/rand.h>
#include <openssl/bn.h>
#include <openssl/ec_lcl.h>
#define ALG_ALL_YES
#define CRYPTO_ENGINE
#include "CryptoBaseTypes.h"
#include "CryptPri.h"
#include "TpmError.h"
#include "<K>bool.h"
#include "<swap.h"
#include "Implementation.h"
#include "TPMB.h"
#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : b)
#endif
#define MAX_2B_BYTES MAX((MAX_RSA_KEY_BYTES * ALG_RSA),
                      MAX((MAX_ECC_PARAMETER_BYTES * ALG_ECC),
                          MAX_DIGEST_SIZE))
#include "Platform.h"
```

These are structures that can't be shared with CryptUtil() This types is used in CryptoEngine() to hold a hash state

```c
typedef BYTE HASH_STATE_BUFFER[MAX_HASH_STATE_SIZE];
#define OSSL_HASH_STATE_DATA_SIZE (MAX_HASH_STATE_SIZE - 8)
typedef struct {
    union {
        EVP_MD_CTX context;
        BYTE data[OSSL_HASH_STATE_DATA_SIZE];
    } u;
    INT16 copySize;
```
This is a structure to hold the parameters for the version of \textit{KDFa()} used by the \textit{CryptoEngine()}. This structure allows the state to be passed between multiple functions that use the same pseudo-random sequence.

\begin{verbatim}
typedef struct {
    HASH_STATE_BUFFER     iPadCtx;
    HASH_STATE_BUFFER     oPadCtx;
    TPM2B                 *extra;
    UINT32                *outer;
    TPM_ALG_ID            hashAlg;
    UINT16                keySizeInBits;
} KDFa_CONTEXT;

#define assert2Bsize(a)   pAssert((a).size <= <K>sizeof((a).buffer))
\end{verbatim}

Include the function prototypes when all the types are defined.

\begin{verbatim}
#include <CpriCryptPri_fp.h>
#include <MathFunctions_fp.h>
#include <CpriRNG_fp.h>
#include <CpriHash_fp.h>
#include <CpriSym_fp.h>
#ifdef TPM_ALG_RSA
    #ifdef RSA_KEY_SIEVE
        #include "RsaKeySieve.h"
        #include "RsaKeySieve_fp.h"
    #endif
    #include "CpriRSA_fp.h"
#endif
#ifdef TPM_ALG_ECC
    #include "CpriDataEcc.h"
    #include "CpriECC_fp.h"
#endif
#define MAX_ECC_PARAMETER_BYTES 32
#endif // CRYPTO_ENGINE_H
\end{verbatim}
9.18 CryptPri.h

9.18.1.1 Introduction

This file contains constant definition shared by CryptUtil() and and the parts of the CryptoEngine().

```c
#ifndef _CRYPT_PRI_H
#define _CRYPT_PRI_H
#endif

#ifndef CRYPTO_ENGINE
#include "BaseTypes.h"
#endif

#include "tpmError.h"
#include "swap.h"
#include "Implementation.h"
#include "TPMB.h"
#include "bool.h"

#ifndef NULL
#define NULL    0
#endif

typedef UINT16 NUMBYTES;    // When a size is a number of bytes
typedef UINT32 NUMDIGITS;    // When a size is a number of "digits"

extern UINT32 g_entropySize;
extern BYTE g_entropy[ ];
```

9.18.1.2 Hash-related Structures

```c
typedef struct {
    const TPM_ALG_ID     alg;
    const NUMBYTES      digestSize;
    const NUMBYTES      blockSize;
    const NUMBYTES      derSize;
    const BYTE          der[20];
} HASH_INFO;

#define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
//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.

```c
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.

```c
typedef struct _HASH_STATE
{
    HASH_STATE_ARRAY     state;
    TPM_ALG_ID           hashAlg;
} CPRI_HASH_STATE, *PCPRI_HASH_STATE;
extern const HASH_INFO   g_hashData[HASH_COUNT + 1];
```
9.18.1.3 Asymmetric Structures and values

```c
#ifdef 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

```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.

```c
typedef struct {
    UINT32            curveID;       // The curve identifier
    TPM_ECC_POINT    *publicPoint;   // Pointer to the public point
    TPM2B            *privateKey;    // Pointer to the private key
} ECC_KEY;
#endif // TPM_ALG_ECC

9.18.1.5 RSA-related Structures

This structure is a succinct representation of the cryptographic components of an RSA key.

```c
typedef struct {
    UINT32        exponent;        // The public exponent pointer
    TPM2B        *publicKey;       // Pointer to the public modulus
    TPM2B        *privateKey;      // The private exponent (not a prime)
} RSA_KEY;
#endif // TPM_ALG_RSA
#endif // TPM_ALG_RSA

#else // RSA but no ECC
#define MAX_NUMBER_SIZE              MAX_RSA_KEY_BYTES
#endif

#define CRYPT_RESULT_MIN    INT16_MIN
#define CRYPT_RESULT_MAX    INT16_MAX
#define CRYPT_FAIL          ((CRYPT_RESULT)  1)
#define CRYPT_SUCCESS       ((CRYPT_RESULT)  0)
#define CRYPT_NO_RESULT     ((CRYPT_RESULT) -1)
#define CRYPT_SCHEME        ((CRYPT_RESULT) -2)
```

<table>
<thead>
<tr>
<th>CRYPT RESULT</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>recoverable error</td>
</tr>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>command specific return value</td>
</tr>
</tbody>
</table>

(generally a digest size)
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

```c
#ifndef _CRYPTO_BASETYPES_H
#define _CRYPTO_BASETYPES_H

Avoid include of baseTypes.h if this file is included

#define _BASETYPES_H
#include "stdint.h"

typedef uint8_t UINT8;
typedef uint8_t BYTE;
typedef int8_t INT8;
typedef int BOOL;
typedef uint16_t UINT16;
typedef int16_t INT16;
typedef uint64_t UINT64;
typedef int64_t INT64;

typedef struct {
    UINT16 size;
    BYTE buffer[1];
} TPM2B;
#endif
```
B.3 CpriData.c

This file should be included by the library hash module.

```c
const HASH_INFO   g_hashData[HASH_COUNT + 1] = {
    #if  ALG_SHA1 == YES
        {TPM_ALG_SHA1,    SHA1_DIGEST_SIZE,   SHA1_BLOCK_SIZE,
            SHA1_DER_SIZE,   SHA1_DER},
    #endif
    #if  ALG_SHA256 == YES
        {TPM_ALG_SHA256,  SHA256_DIGEST_SIZE, SHA256_BLOCK_SIZE,
            SHA256_DER_SIZE, SHA256_DER},
    #endif
    #if  ALG_SHA384 == YES
        {TPM_ALG_SHA384,  SHA384_DIGEST_SIZE, SHA384_BLOCK_SIZE,
            SHA384_DER_SIZE, SHA384_DER},
    #endif
    #if  ALG_SHA512 == YES
        {TPM_ALG_SHA512,  SHA512_DIGEST_SIZE, SHA512_BLOCK_SIZE,
            SHA512_DER_SIZE, SHA512_DER},
    #endif
    #if  ALG_SM3_256 == YES
        {TPM_ALG_SM3_256, SM3_256_DIGEST_SIZE, SM3_256_BLOCK_SIZE,
            SM3_256_DER_SIZE, SM3_256_DER},
    #endif
    {TPM_ALG_NULL,0,0,0,{0}}
};
```
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.

#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no significant bytes, value is zero</td>
</tr>
<tr>
<td>&gt;0</td>
<td>number of significant bytes</td>
</tr>
</tbody>
</table>

UINT16 _math__Normalize2B( TPM2B *b ) {  // IN/OUT: number to normalize
    UINT16 from;
    UINT16 to;
    UINT16 size = b->size;
    for(from = 0; b->buffer[from] == 0 && from < size; from++);
    b->size -= from;
    for(to = 0; from < size; to++, from++)
        b->buffer[to] = b->buffer[from];
    return b->size;  }
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>number denormalized</td>
</tr>
<tr>
<td>FALSE</td>
<td>number already larger than the desired size</td>
</tr>
</tbody>
</table>

```c
BOOL _math__Denormalize2B(
    TPM2B       *in, // IN:OUT TPM2B number to denormalize
    UINT32       size // IN: the desired size
) {
    UINT32 to;
    UINT32 from;
    // If the current size is greater than the requested size, see if this can be
    // normalized to a value smaller than the requested size and then de-normalize
    if(in->size > size)
        { _math__Normalize2B(in);
        if(in->size > size)
            return FALSE;
        }
    // If the size is already what is requested, leave
    if(in->size == size)
        return TRUE;
    // move the bytes to the 'right'
    for(from = in->size, to = size; from > 0;)
        in->buffer[--to] = in->buffer[--from];
    // 'to' will always be greater than 0 because we checked for equal above.
    for(; to > 0;)
        in->buffer[--to] = 0;
    in->size = (UINT16)size;
    return TRUE;
}
```

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 \( b \).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if (a &gt; b) so no borrow</td>
</tr>
<tr>
<td>0</td>
<td>if (a = b) so no borrow and b == a</td>
</tr>
<tr>
<td>-1</td>
<td>if (a &lt; b) so there was a borrow</td>
</tr>
</tbody>
</table>

```c
int _math__sub(
    const UINT32       aSize,       // IN: size of a
    const BYTE         *a,          // IN: a
    const UINT32       bSize,       // IN: size of b
    const BYTE         *b,          // IN: b
    UINT16             *cSize,       // OUT: set to MAX(aSize, bSize)
    const BYTE         *c               // OUT: the difference
)
```
{  
  int borrow = 0;
  int notZero = 0;
  int i;
  int i2;
  
  // set c to the longer of a or b
  *cSize = (UINT16)((aSize > bSize) ? aSize : bSize);
  // pick the shorter of a and b
  i = (aSize > bSize) ? bSize : aSize;
  i2 = *cSize - i;
  a = &a[aSize - 1];
  b = &b[bSize - 1];
  c = &c[*cSize - 1];
  for(; i > 0; i--)
  {
    borrow = *a-- - *b-- + borrow;
    *c-- = (BYTE)borrow;
    notZero = notZero || borrow;
    borrow >>= 8;
  }
  if(aSize > bSize)
  {
    for(; i2 > 0; i2--)
    {
      borrow = *a-- + borrow;
      *c-- = (BYTE)borrow;
      notZero = notZero || borrow;
      borrow >>= 8;
    }
  }  
  else if(aSize < bSize)
  {
    for(; i2 > 0; i2--)
    {
      borrow = 0 - *b-- + borrow;
      *c-- = (BYTE)borrow;
      notZero = notZero || borrow;
      borrow >>= 8;
    }
  }  
  // if there is a borrow, then b > a
  if(borrow)
  {
    return -1;
  }  
  // either a > b or they are the same
  return notZero;
}

B.4.2.4. _math___Inc()

This function increments a large, big-endian number value by one.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>result is zero</td>
</tr>
<tr>
<td>!0</td>
<td>result is not zero</td>
</tr>
</tbody>
</table>

int _math___Inc(
UINT32 aSize,     // IN: size of a
BYTE *a          // IN: a)
)
for (a = &a[aSize-1]; aSize > 0; aSize--)
{
    if (*(a--) += 1) != 0
        return 1;
}
return 0;

B.4.2.5. _math__Dec

This function decrements a large, ENDIAN value by one.

void _math__Dec(
    UINT32 aSize,  // IN: size of a
    BYTE *a       // IN: a
)
{
    for (a = &a[aSize-1]; aSize > 0; aSize--)
    {
        if (*(a--) -= 1) != 0xff)
            return;
    }
    return;
}

B.4.2.6. _math__Mul()

This function is used to multiply two large integers: \( p = a \times b \). 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 \( aSize + pSize \).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>indicates an error</td>
</tr>
<tr>
<td>&gt;= 0</td>
<td>the size of the product</td>
</tr>
</tbody>
</table>

int _math__Mul(
    const UINT32 aSize,  // IN: size of a
    const BYTE *a,      // IN: a
    const UINT32 bSize,  // IN: size of b
    const BYTE *b,      // IN: b
    UINT32 *pSize,      // IN/OUT: size of the product
    BYTE *p,            // OUT: product. length of product = aSize + bSize
)
{
    BIGNUM *bnA;
    BIGNUM *bnB;
    BIGNUM *bnP;
    BN_CTX *context;
    int retVal = 0;

    // First check that pSize is large enough if present
    if ((pSize != NULL) && (*pSize < (aSize + bSize)))
        return CRYPTO_PARAMETER;
    pAssert(*pSize < MAX_2B_BYTES);
}
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 // Convert the inputs to BIGNUMs
164 //
165 if (BN_bin2bn(a, aSize, bnA) == NULL || BN_bin2bn(b, bSize, bnB) == NULL)
166 FAIL(FATAL_ERROR_INTERNAL);
167 // Perform the multiplication
168 //
169 if (BN_mul(bnP, bnA, bnB, context) != 1)
170 FAIL(FATAL_ERROR_INTERNAL);
171 // If the size of the results is allowed to float, then set the return
172 // size. Otherwise, it might be necessary to denormalize the results
173 retVal = BN_num_bytes(bnP);
174 if (pSize == NULL)
175 {
176 BN_bn2bin(bnP, &p[aSize + bSize - retVal]);
177 memset(p, 0, aSize + bSize - retVal);
178 retVal = aSize + bSize;
179 }
180 else
181 {
182 BN_bn2bin(bnP, p);
183 *pSize = retVal;
184 }
185 BN_CTX_end(context);
186 BN_CTX_free(context);
187 return retVal;
188
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>operation complete</td>
</tr>
<tr>
<td>CRYPT_UNDERFLOW</td>
<td>( q ) or ( r ) is too small to receive the result</td>
</tr>
</tbody>
</table>

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;
BIGNUM          *bnR;
BN_CTX          *context;
CRYPT_RESULT    retVal = CRYPT_SUCCESS;

// Get structures for the big number representations
context = BN_CTX_new();
if (context == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);
BN_CTX_start(context);
bnN = BN_CTX_get(context);
bnD = BN_CTX_get(context);
bnQ = BN_CTX_get(context);
bnR = BN_CTX_get(context);

// Errors in BN_CTX_get() are sticky so only need to check the last allocation
if (   bnR == NULL
     || BN_bin2bn(n->buffer, n->size, bnN) == NULL
     || BN_bin2bn(d->buffer, d->size, bnD) == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

// Check for divide by zero.
if (BN_num_bits(bnD) == 0)
    FAIL(FATAL_ERROR_DIVIDE_ZERO);

// Perform the division
if (BN_div(bnQ, bnR, bnN, bnD, context) != 1)
    FAIL(FATAL_ERROR_INTERNAL);

// Convert the BIGNUM result back to our format
if (q != NULL)  // If the quotient is being returned
    {  
        if (!BnTo2B(q, bnQ, q->size))
            {  
                retVal = CRYPT_UNDERFLOW;
                goto Done;
            }
    }
if (r != NULL)  // If the remainder is being returned
    {  
        if (!BnTo2B(r, bnR, r->size))
            retVal = CRYPT_UNDERFLOW;
    }

Done:
BN_CTX_end(context);
BN_CTX_free(context);
return retVal;

B.4.2.8. _math__uComp()

This function compare two unsigned values.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if (a &gt; b)</td>
</tr>
<tr>
<td>0</td>
<td>if (a = b)</td>
</tr>
<tr>
<td>-1</td>
<td>if (a &lt; b)</td>
</tr>
</tbody>
</table>

int
_math__uComp();
B.4.2.9. \_math\_Comp()

Compare two signed integers:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#a&gt;b</td>
</tr>
<tr>
<td>0</td>
<td>#a=b</td>
</tr>
<tr>
<td>-1</td>
<td>#a&lt;b</td>
</tr>
</tbody>
</table>

```c
int \_math\_Comp(
    const UINT32 aSize, // IN: size of a
    const BYTE *a,    // IN: a
    const UINT32 bSize, // IN: size of b
    const BYTE *b    // IN: b
) {
    int signA, signB; // sign of a and b
    // For positive or 0, sign a is 1
    // for negative, sign a is 0
    signA = static_cast<int>((a[0] & 0x80) == 0) ? 1 : 0;
    signB = static_cast<int>((b[0] & 0x80) == 0) ? 1 : 0;
    int borrow = 0;
    int notZero = 0;
    int i;
    // If a has more digits than b, then a is greater than b if
    // any of the more significant bytes is non zero
    if ((i = (int)aSize - (int)bSize) > 0)
        for (; i > 0; i--)
            if (*a++) // means a > b
                return 1;
    // If b has more digits than a, then b is greater if any of the
    // more significant bytes is non zero
    if (i < 0)  // Means that b is longer than a
        for (; i < 0; i++)
            if (*b++) // means that b > a
                return -1;
    // Either the values are the same size or the upper bytes of a or b are
    // all zero, so compare the rest
    i = (aSize > bSize) ? bSize : aSize;
    a = &a[i-1];
    b = &b[i-1];
    for (; i > 0; i--)
        { // if there is a borrow, then b > a
            borrow = *a-- - *b-- + borrow;
            notZero = notZero || borrow;
            borrow >>= 8;
        }
    // if there is a borrow, then b > a
    if (borrow)
        return -1;
    // either a > b or they are the same
    return notZero;
}
```
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: \( c = m^e \mod n \) (RSA encrypt) and \( m = c^d \mod n \) (RSA decrypt). When doing decryption, the \( e \) parameter of the function will contain the private exponent \( d \) instead of the public exponent \( e \).

If the results will not fit in the provided buffer, an error is returned (CRYPT_ERROR_UNDERFLOW). If the results is smaller than the buffer, the results is de-normalized.

This version is intended for use with RSA and requires that \( m \) be less than \( n \).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>exponentiation succeeded</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>number to exponentiate is larger than the modulus</td>
</tr>
<tr>
<td>CRYPT_UNDERFLOW</td>
<td>result will not fit into the provided buffer</td>
</tr>
</tbody>
</table>

CRYPT_RESULT

_math__ModExp(

326     UINT32           cSize, // IN: size of the results
327     BYTE            *c, // OUT: results buffer
328     const UINT32     mSize, // IN: size of number to be exponentiated
329     const BYTE      *m, // IN: number to be exponentiated
330     const UINT32     eSize, // IN: size of power
331     const BYTE      *e, // IN: power
332     const UINT32     nSize, // IN: modulus size
333     const BYTE      *n // IN: modulus
334 }
335 }
336     CRYPT_RESULT     retVal = CRYPT_SUCCESS;
337     BN_CTX           *context;
338     BIGNUM           *bnC;
339     BIGNUM           *bnM;
340     BIGNUM           *bnE;
341     BIGNUM           *bnN;
342     INT32            i;
343     context = BN_CTX_new();
344     if(context == NULL)
345     FAIL(FATAL_ERROR_ALLOCATION);
346     BN_CTX_start(context);
347     bnC = BN_CTX_get(context);
348     bnM = BN_CTX_get(context);
349     bnE = BN_CTX_get(context);
350     bnN = BN_CTX_get(context);
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
372  // Perform the exponentiation
373  if (!BN_mod_exp(bnC, bnM, bnE, bnN, context))
374    FAIL(FATAL_ERROR_INTERNAL);
375
376  // Convert the results
377  // Make sure that the results will fit in the provided buffer.
378  if ((unsigned)BN_num_bytes(bnC) > cSize)
379  {
380    retVal = CRYPT_UNDERFLOW;
381    goto Cleanup;
382  }
383  i = cSize - BN_num_bytes(bnC);
384  BN_bn2bin(bnC, &c[i]);
385  memset(c, 0, i);
386
387  Cleanup:
388  // Free up allocated BN values
389  BN_CTX_end(context);
390  BN_CTX_free(context);
391  return retVal;
392
B.4.2.11. _math__IsPrime()

Check if an integer is probably a prime.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if the integer is probably a prime</td>
</tr>
<tr>
<td>FALSE</td>
<td>if the integer is definitely not a prime</td>
</tr>
</tbody>
</table>

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;
BIGNUM *p;

// Assume the size variables are not overflow, which should not happen in
// the contexts that this function will be called.
pAssert((int) primeSize >= 0);
if((p = BN_new()) == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);
if(BN_bin2bn(prime, primeSize, p) == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

// BN_is_prime returning -1 means that it ran into an error.
// It should only return 0 or 1
if((isPrime = BN_is_prime_ex(p, BN_prime_checks, NULL, NULL)) < 0)
    FAIL(FATAL_ERROR_INTERNAL);

if(p != NULL)
    BN_clear_free(p);
return (isPrime == 1);
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.

```
#include "CryptoEngine.h"
```

B.5.1. Initialization and Shutdown

B.5.1.1. _cpri__InitCryptoUnits()

Initialize crypto units

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPTO_RESULT</td>
<td></td>
</tr>
</tbody>
</table>

```
翠_InitCryptoUnits(void)
{
    _cpri__RngStartup();
    _cpri__HashStartup();
    _cpri__SymStartup();

    #ifdef TPM_ALG_RSA
    _cpri__RsaStartup();
    #endif

    #ifdef TPM_ALG_ECC
    _cpri__EccStartup();
    #endif

    return 0;
}
```

B.5.1.2. _cpri__StopCryptoUnits()

Shut down the crypto function units

```
void _cpri__StopCryptoUnits(void)
{
    return;
}
```

B.5.1.3. _cpri__Startup()

Start the crypto after startup

```
BOOL _cpri__Startup()
{
    return ( _cpri__HashStartup()}
```
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().

```c
CRYPT_RESULT _cpri__IncrementalSelfTest(
    TPML_ALG *toTest,       // IN: list of algorithms to be tested
    TPML_ALG *toDoList      // OUT: list of algorithms need test
)
{
    // Always copy toTest list to todoList
    *todoList = *toTest;
    return CRYPT_SUCCESS;
}
```

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 CRYPT_ENGINE_H is defined. Any file that include CryptEngine.h is considered to be part of the CryptoEngine().

```c
//%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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>failure (probably fatal)</td>
</tr>
<tr>
<td>1</td>
<td>conversion successful</td>
</tr>
</tbody>
</table>
BnTo2B()

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   if(size == 0)
61     size = outVal->size;
62   else
63     outVal->size = size;
64   size = size - (((UINT16) BN_num_bits(inVal) + 7) / 8);
65   if(size < 0)
66     return FALSE;
67   for(;size > 0; size--)
68       *pb++ = 0;
69   BN_bn2bin(inVal, pb);
70   return TRUE;
71 }

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 }
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 _cpri__RngStartup(void)
9  {
10      memcpy(randomSeed.t.buffer,
11            "Special version of the RNG to be used only during TPM debug!!!!",
12            64);
13      randomSeed.t.size = 64;
14      rngCounter = 923;
15      return TRUE;
16  }

B.6.1.1. _cpri__StirRandom()

Set random entropy

18  CRYPT_RESULT _cpri__StirRandom(
19      INT32 entropySize,
20      BYTE *entropy
21  )
22  {
23      if (entropySize >= 0)
24      {
25          randomSeed.t.size = (entropySize > 64) ? 64 : entropySize;
26          memcpy(randomSeed.t.buffer, entropy, randomSeed.t.size);
27          rngCounter = 0;
28      }
29      return CRYPT_SUCCESS;
30  }

B.6.1.2. _cpri__GenerateRandom()

Generate a randomSize number or random bytes.

33  UINT16 _cpri__GenerateRandom(
34      INT32 randomSize,
35      BYTE *buffer
36  )
37  {
38    //
39    // We don't do negative sizes or ones that are too large
40    if (randomSize < 0 || randomSize > UINT16_MAX)
41      return 0;
42    // RAND_bytes uses 1 for success and we use 0
43    _cpri__KDFa(TPM_ALG_SHA256,
44      &randomSeed.b,
45      "Not really random numbers",
46      NULL,
47      NULL,
48      randomSize * 8,
buffer,
    &rngCounter,
    FALSE);
  return randomSize;
}
#else //%

B.6.2. Random Number Generation

BOOL _cpri__RngStartup(void)
{
  UINT32 entropySize;
  BYTE entropy[MAX_RNG_ENTROPY_SIZE];
  // Collect entropy until we have enough
  for(entropySize = 0;
    entropySize < MAX_RNG_ENTROPY_SIZE;
    entropySize += _plat__GetEntropy(&entropy[entropySize],
    MAX_RNG_ENTROPY_SIZE - entropySize));
  // Seed OpenSSL with entropy
  RAND_seed(entropy, entropySize);
  return TRUE;
}

B.6.2.1. _cpri__StirRandom()

Set random entropy

CRYPT_RESULT _cpri__StirRandom(
    INT32 entropySize,
    BYTE *entropy
{
  if (entropySize >= 0)
  {
    RAND_add((const void *)entropy, (int) entropySize, 0.0);
  }
  return CRYPT_SUCCESS;
}

B.6.2.2. _cpri__GenerateRandom()

Generate a random size number or random bytes.

UINT16 _cpri__GenerateRandom(
    INT32 randomSize,
    BYTE *buffer
{
  // We don't do negative sizes or ones that are too large
  if (randomSize < 0 || randomSize > UINT16_MAX)
    return 0;
  // RAND_bytes uses 1 for success and we use 0
  if (RAND_bytes(buffer, randomSize) == 1)
    return (UINT16)randomSize;
  else
    return 0;
}  
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>the number of bytes of buf used.</td>
</tr>
</tbody>
</table>

```c
static UINT16
MarshalHashState(EVP_MD_CTX      *ctxt, // IN: Context to marshal BYTE            *buf       // OUT: The buffer that will receive the context. This buffer is at least MAX_HASH_STATE_SIZE bytes)
{
    // make sure everything will fit
    pAssert(ctxt->digest->ctx_size <= (MAX_HASH_STATE_SIZE - sizeof(INT16) - sizeof(TPM_ALG_ID)));

    // Copy the context data
    memcpy(buf, (void*) ctxt->md_data, ctxt->digest->ctx_size);
    return (UINT16)ctxt->digest->ctx_size;
}
```

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).

```c
static UINT16
GetHashState(EVP_MD_CTX      *ctxt, // OUT: The context structure to receive TPM_ALG_ID       algType, // IN: The hash algorithm selector BYTE            *buf       // IN: Buffer containing marshaled hash data)
{
    EVP_MD          *evpmdAlgorithm = NULL;

    pAssert(ctxt != NULL);
    EVP_MD_CTX_init(ctxt);
    evpmdAlgorithm = GetHashServer(algType);
    if(evpmdAlgorithm == NULL)
        return 0;

    // This also allocates the ctxt->md_data
    if((EVP_DigestInit_ex(ctxt, evpmdAlgorithm, NULL)) != 1)
        FAIL(FATAL_ERROR_INTERNAL);
    memcpy(ctxt->md_data, buf, ctxt->digest->ctx_size);
    return (UINT16)ctxt->digest->ctx_size;
}
```

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     // Get the table size of g_hashData
91     tableSize = sizeof(g_hashData) / sizeof(g_hashData[0]);
92     for(i = 0; i < tableSize - 1; i++)
93     {
94         if(g_hashData[i].alg == hashAlg)
95             return &g_hashData[i];
96     }
97     pAssert(hashAlg == TPM_ALG_NULL);
98     return &g_hashData[tableSize-1];
99 }

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 and index of 2 will return the last. All other index values will return TPM_ALG_NULL.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_ALG_xxx()</td>
<td>a hash algorithm</td>
</tr>
<tr>
<td>TPM_ALG_NULL</td>
<td>this can be used as a stop value</td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>the algorithm is not a supported hash</td>
</tr>
<tr>
<td>≥</td>
<td>the digest size (0 for TPM_ALG_NULL)</td>
</tr>
</tbody>
</table>

```c
UINT16 _cpri__GetHashBlockSize(
    TPM_ALG_ID hashAlg  // IN: hash algorithm to look up
) {
    return GetHashInfoPointer(hashAlg)->blockSize;
}
```

### B.7.4.4. _cpri__GetHashDER

This function returns a pointer to the DER string for the algorithm and indicates its size.

```c
UINT16 _cpri__GetHashDER(
    TPM_ALG_ID                      hashAlg,   // IN: the algorithm to look up
    const BYTE **p
) {
    const HASH_INFO       *q;
    q = GetHashInfoPointer(hashAlg);
    *p = &q->der[0];
    return q->derSize;
}
```

### B.7.4.5. _cpri__GetDigestSize()

Gets the digest size of the algorithm. The algorithm is required to be supported.

```c
UINT16 _cpri__GetDigestSize(
    TPM_ALG_ID hashAlg       // IN: hash algorithm to look up
) {
    return GetHashInfoPointer(hashAlg)->digestSize;
}
```

### B.7.4.6. _cpri__GetContextAlg()

This function returns the algorithm associated with a hash context.

```c
TPM_ALG_ID _cpri__GetContextAlg(
    void *hashState  // IN: the hash context
) {
    CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
    return state->hashAlg;
}
```
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.

```c
UINT16 _cpri__CopyHashState ( 
    void *out,    // OUT: destination of the state
    void *in      // IN: source of the state
)
{
    CPRI_HASH_STATE *i = (CPRI_HASH_STATE *)in;
    CPRI_HASH_STATE *o = (CPRI_HASH_STATE *)out;
    EVP_MD_CTX_init(&o->u.context);
    EVP_MD_CTX_copy_ex(&o->u.context, &i->u.context);
    o->size = i->size;
    o->hashAlg = i->hashAlg;
    return sizeof(CPRI_HASH_STATE);
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hash is TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt;0</td>
<td>digest size</td>
</tr>
</tbody>
</table>

```c
UINT16 _cpri__StartHash( 
    TPM_ALG_ID hashAlg,    // IN: hash algorithm
    BOOL sequence,          // IN: TRUE if the state should be saved
    void *hashState     // OUT: the state of hash stack.
)
{
    EVP_MD_CTX localState;
    CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
    EVP_MD_CTX *context;
    EVP_MD *evpmdAlgorithm = NULL;
    UINT16 retVal = 0;
    if(sequence)
        context = &localState;
    else
        context = &state->u.context;
    state->hashAlg = hashAlg;
    EVP_MD_CTX_init(context);
    evpmdAlgorithm = GetHashServer(hashAlg);
    if(evpmdAlgorithm == NULL)
        goto Cleanup;
    if(EVP_DigestInit_ex(context, evpmdAlgorithm, NULL) != 1)
        FAIL(FATAL_ERROR_INTERNAL);
```
\begin{verbatim}
196     retVal = (CRYPT_RESULT)EVP_MD_CTX_size(context);
197
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                 }
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 _cpri__UpdateHash(
222     void *hashState, // IN: the hash context information
223     UINT32   dataSize, // IN: the size of data to be added to the digest
224     BYTE     *data    // IN: data to be hashed
225 )
226 {
227     EVP_MD_CTX    localContext;
228     CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
229     EVP_MD_CTX     *context;
230     CRYPT_RESULT   retVal = CRYPT_SUCCESS;
231
232     if(state->size == 0)
233         return;
234     if(state->size > 0)
235         {
236             context = &localContext;
237             if((retVal = GetHashState(context, state->hashAlg, state->u.data)) <= 0)
238                 return;
239         }
240     else
241         context = &state->u.context;
242
243     if(EVP_DigestUpdate(context, data, dataSize) != 1)
244         FAIL(FATAL_ERROR_INTERNAL);
245     else if( state->size > 0
246             && (retVal= MarshalHashState(context, state->u.data)) >= 0)
247         {
248             // retVal is the size of the marshaled data. Make sure that it is consistent
249             // by ensuring that we didn't get more than allowed
250             if(retVal < state->size)
251                 FAIL(FATAL_ERROR_INTERNAL);
252             else
253                 EVP_MD_CTX_cleanup(context);
\end{verbatim}
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 <= 0.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no data returned</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the number of bytes in the digest</td>
</tr>
</tbody>
</table>

```
UINT16
_cpri__CompleteHash(
    void *hashState, // IN: the state of hash stack
    UINT32 dOutSize, // IN: size of digest buffer
    BYTE *dOut // OUT: hash digest
)
{
    EVP_MD_CTX localState;
    CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
    EVP_MD_CTX *context;
    UINT16 retVal;
    int hLen;
    BYTE temp[MAX_DIGEST_SIZE];
    BYTE *rBuffer = dOut;

    if(state->size == 0)
        return 0;
    if(state->size > 0)
    {
        context = &localState;
        if(retVal = GetHashState(context, state->hashAlg, state->u.data)) <= 0)
            goto Cleanup;
    }
    else
    {
        context = &state->u.context;
        hLen = EVP_MD_CTX_size(context);
        if((unsigned)hLen > dOutSize)
            rBuffer = temp;
        if(EVP_DigestFinal_ex(context, rBuffer, NULL) == 1)
        {
            if(rBuffer != dOut)
            {
                if(dOut != NULL)
                {
                    memcpy(dOut, temp, dOutSize);
                }
                retVal = (UINT16)dOutSize;
            }
            else
            {
                retVal = (UINT16)hLen;
            }
        }
        else
        {
            state->size = 0;
        }
    }
    else
    {
        retVal = 0; // Indicate that no data is returned
```
}
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>number of bytes in digest (may be zero)</td>
</tr>
</tbody>
</table>

```c
UINT16 _cpri__HashBlock(
    TPM_ALG_ID    hashAlg, // IN: The hash algorithm
    UINT32        dataSize, // IN: size of buffer to hash
    BYTE          *data,   // IN: the buffer to hash
    UINT32        digestSize, // IN: size of the digest buffer
    BYTE          *digest   // OUT: hash digest
)
{
    EVP_MD_CTX    hashContext;
    EVP_MD       *hashServer = NULL;
    UINT16        retVal = 0;
    BYTE          b[MAX_DIGEST_SIZE]; // temp buffer in case digestSize not
    // a full digest
    unsigned int  dSize = _cpri__GetDigestSize(hashAlg);

    // If there is no digest to compute return
    if(dSize == 0)
        return 0;

    // After the call to EVP_MD_CTX_init(), will need to call EVP_MD_CTX_cleanup()
    EVP_MD_CTX_init(&hashContext); // Initialize the local hash context
    hashServer = GetHashServer(hashAlg); // Find the hash server

    // It is an error if the digest size is non-zero but there is no server
    if( (hashServer == NULL) || (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1) || (EVP_DigestUpdate(&hashContext, data, dataSize) != 1))
        FAIL(FATAL_ERROR_INTERNAL);
    else
    {
        // If the size of the digest produced (dSize) is larger than the available
        // buffer (digestSize), then put the digest in a temp buffer and only copy
        // the most significant part into the available buffer.
        if(dSize > digestSize)
        {
            if(EVP_DigestFinal_ex(&hashContext, b, &dSize) != 1)
                FAIL(FATAL_ERROR_INTERNAL);
            memcpy(digest, b, digestSize);
            retVal = (UINT16)digestSize;
        }
        else
        {
            if((EVP_DigestFinal_ex(&hashContext, digest, &dSize)) != 1)
                FAIL(FATAL_ERROR_INTERNAL);
            retVal = (UINT16) dSize;
        }
    }
}
```
EVP_MD_CTX_cleanup(&hashContext);
return retVal;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>number of bytes in digest produced by hashAlg (may be zero)</td>
</tr>
</tbody>
</table>

```c
UINT16 _cpri__StartHMAC(
    TPM_ALG_ID       hashAlg,   // IN: the algorithm to use
    BOOL             sequence, // IN: indicates if the state should be saved
    void *state,     // IN/OUT: the state buffer
    UINT16           keySize,  // IN: the size of the HMAC key
    BYTE *key,       // IN: the HMAC key
    TPM2B *oPadKey   // OUT: the key prepared for the oPad round
) {
    CPRI_HASH_STATE  localState;
    UINT16           blockSize = _cpri__GetHashBlockSize(hashAlg);
    UINT16           digestSize;
    BYTE            *pb;   // temp pointer
    UINT32           i;

    if(keySize > blockSize)
    {
        if((digestSize = _cpri__StartHash(hashAlg, FALSE, &localState)) == 0)
            return 0;
        _cpri__UpdateHash(&localState, keySize, key);
        _cpri__CompleteHash(&state, digestSize, oPadKey->buffer);
        oPadKey->size = digestSize;
    }
    else
    {
        memcpy(oPadKey->buffer, key, keySize);
        oPadKey->size = keySize;
    }
    // XOR the key with iPad (0x36)
    pb = oPadKey->buffer;
    for(i = oPadKey->size; i > 0; i--)
        *pb++ ^= 0x36;
    // if the keySize is smaller than a block, fill the rest with 0x36
    for(i = blockSize - oPadKey->size; i > 0; i--)
        *pb++ = 0x36;
    // Increase the oPadSize to a full block
    oPadKey->size = blockSize;
    // Start a new hash with the HMAC key
    // This will go in the caller’s state structure and may be a sequence or not
```
if((digestSize = _cpri__StartHash(hashAlg, sequence, state)) > 0)
{
    _cpri__UpdateHash(state, oPadKey->size, oPadKey->buffer);
    // XOR the key block with 0x5c ^ 0x36
    for(pb = oPadKey->buffer, i = blockSize; i > 0; i--)
        *pb++ ^= (0x5c ^ 0x36);
}
return digestSize;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>number of bytes in dOut (may be zero)</td>
</tr>
</tbody>
</table>

UINT16 _cpri__CompleteHMAC(
void *hashState, // IN: the state of hash stack
TPM2B *oPadKey, // IN: the HMAC key in oPad format
UINT32 dOutSize, // IN: size of digest buffer
BYTE *dOut // OUT: hash digest )
{
    BYTE digest[MAX_DIGEST_SIZE];
    CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
    CPRI_HASH_STATE localState;
    UINT16 digestSize = _cpri__GetDigestSize(state->hashAlg);
    // Using the local hash state, do a hash with the oPad
    if(_cpri__StartHash(state->hashAlg, FALSE, &localState) != digestSize)
        return 0;
    _cpri__UpdateHash(&localState, oPadKey->size, oPadKey->buffer);
    _cpri__UpdateHash(&localState, digestSize, digest);
    return _cpri__CompleteHash(&localState, dOutSize, dOut);
}

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(seed || 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 mSize

### CRYPT_RESULT

```c
_Cpri__MGF1(
    UINT32 mSize,       // IN: length of the mask to be produced
    BYTE *mask,         // OUT: buffer to receive the mask
    TPM_ALG_ID hashAlg, // IN: hash to use
    UINT32 sSize,       // IN: size of the seed
    BYTE *seed          // IN: seed size
)
```
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 \(2^{18}-1 = 256K\) bits (32385 bytes).

The `once` parameter is set to allow incremental generation of a large value. If this flag is TRUE, `sizeInBits` will be used in the HMAC computation but only one iteration of the KDF is performed. This would be used for XOR obfuscation so that the mask value can be generated in digest-sized chunks rather than having to be generated all at once in an arbitrarily large buffer and then XORed() into the result. If `once` is TRUE, then `sizeInBits` must be a multiple of 8.

Any error in the processing of this command is considered fatal.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hash algorithm is not supported or is TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the number of bytes in the <code>keyStream</code> buffer</td>
</tr>
</tbody>
</table>

```c
UINT16 _cpri_KDFa(
    TPM_ALG_ID   hashAlg,    // IN: hash algorithm used in HMAC
    TPM2B       *key,     // IN: HMAC key
    const char  *label,    // IN: a 0-byte terminated label used in KDF
    TPM2B       *contextU, // IN: context U
    TPM2B       *contextV, // IN: context V
    UINT32       sizeInBits, // IN: size of generated key in bits
    BYTE        *keyStream,  // OUT: key buffer
    UINT32     *counterInOut, // IN/OUT: caller may provide the iteration counter
    bool        once       // IN: TRUE if only one iteration is performed
    // FALSE if iteration count determined by "sizeInBits"
) {
    UINT32 counter = 0;       // counter value
    INT32 lLen;               // length of the label
    INT16 hLen;               // length of the hash
    INT16 bytes;              // number of bytes to produce
    BYTE *stream = keyStream;
    BYTE marshaledUint32[4];
    BYTE hmacState[MAX_HASH_STATE_SIZE];
    TPM2B_MAX_HASH_BLOCK hmacKey;

    pAssert(key != NULL && keyStream != NULL);
    pAssert(once == FALSE || (sizeInBits & 7) == 0);
```
if (counterInOut != NULL)
    counter = *counterInOut;

// Prepare label buffer. Calculate its size and keep the last 0 byte
for(lLen = 0; label[lLen++] != 0; );

// Get the hash size. If it is less than or 0, either the
// algorithm is not supported or the hash is TPM_ALG_NULL
// In either case the digest size is zero. This is the only return
// other than the one at the end. All other exits from this function
// are fatal errors. After we check that the algorithm is supported
// anything else that goes wrong is an implementation flaw.
if((hLen = (INT16) _cpri__GetDigestSize(hashAlg)) == 0)
    return 0;

// If the size of the request is larger than the numbers will handle,
// it is a fatal error.
pAssert(((sizeInBits + 7)/ 8) <= INT16_MAX);
bytes = once ? hLen : (INT16)((sizeInBits + 7) / 8);

// Generate required bytes
for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
{
    if(bytes < hLen)
        hLen = bytes;

    counter++;
    // Start HMAC
    if(_cpri__StartHMAC(hashAlg,
                        FALSE,
                        &hashState,
                        key->size,
                        &key->buffer[0],
                        &hmacKey.b) <= 0)
        FAIL(FATAL_ERROR_INTERNAL);

    // Adding counter
    UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
    _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);

    // Adding label
    if(label != NULL)
        _cpri__UpdateHash(&hashState,  lLen, (BYTE *)label);

    // Adding contextU
    if(contextU != NULL)
        _cpri__UpdateHash(&hashState, contextU->size, contextU->buffer);

    // Adding contextV
    if(contextV != NULL)
        _cpri__UpdateHash(&hashState, contextV->size, contextV->buffer);

    // Adding size in bits
    UINT32_TO_BYTE_ARRAY(sizeInBits, marshaledUint32);
    _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);

    // Compute HMAC. At the start of each iteration, hLen is set
    // to the smaller of hLen and bytes. This causes bytes to decrement
    // exactly to zero to complete the loop
    _cpri__CompleteHMAC(&hashState, &hmacKey.b, hLen, stream);
}

// Mask off bits if the required bits is not a multiple of byte size
if((sizeInBits % 8) != 0)
    keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
if (counterInOut != NULL)
    *counterInOut = counter;
return (CRYPT_RESULT)((sizeInBits + 7)/8);

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 $(2^{18}) - 1 = 256K$ bits (32385 bytes). Any error in the processing of this command is considered fatal.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hash algorithm is not supported or is TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the number of bytes in the _keyStream buffer</td>
</tr>
</tbody>
</table>

UINT16 _cpri__KDFe(
    TPM_ALG_ID hashAlg,       // IN: hash algorithm used in HMAC
    TPM2B *Z,                // IN: Z
    const char *label,       // IN: a 0-byte terminated label using in KDF
    TPM2B *partyUInfo,      // IN: PartyUInfo
    TPM2B *partyVInfo,      // IN: PartyVInfo
    UINT32 _sizeInBits,     // IN: size of generated key in bits
    BYTE *keyStream         // OUT: key buffer
)

UINT32 counter = 0;       // counter value
UINT32 lSize = 0;
BYTE *stream = keyStream;
BYTE hashState[MAX_HASH_STATE_SIZE];
INT16 hLen = (INT16)_cpri__GetDigestSize(hashAlg);
INT16 bytes;             // number of bytes to generate
BYTE marshaledUint32[4];

pAssert(keyStream != NULL
    && Z != NULL
    && ((sizeInBits + 7) / 8) < INT16_MAX);
if (hLen == 0)
    return 0;

bytes = (INT16)((sizeInBits + 7) / 8);

// Prepare label buffer. Calculate its size and keep the last 0 byte
if (label != NULL)
    for(lSize = 0; label[lSize++] != 0;);

// Generate required bytes
//The inner loop of that KDF uses:
// Hashi := H(counter | Z | OtherInfo) (5)
// Where:
// Hashi the hash generated on the i-th iteration of the loop.
// H() an approved hash function
// counter a 32-bit counter that is initialized to 1 and incremented
// on each iteration
// Z the X coordinate of the product of a public ECC key and a
different private ECC key.
// OtherInfo a collection of qualifying data for the KDF defined below.
// In this specification, OtherInfo will be constructed by:
// OtherInfo := Use | PartyUInfo | PartyVInfo
for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
{
    if (bytes < hLen)
        hLen = bytes;

    counter++;
    // Start hash
    if (_cpri__StartHash(hashAlg, FALSE, &hashState) == 0)
        return 0;

    // Add counter
    UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
    _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);

    // Add Z
    if (Z != NULL)
        _cpri__UpdateHash(&hashState, Z->size, Z->buffer);

    // Add label
    if (label != NULL)
        _cpri__UpdateHash(&hashState, lSize, (BYTE *)label);
    else
        // The SP800-108 specification requires a zero between the label 
        // and the context.
        _cpri__UpdateHash(&hashState, 1, (BYTE *)"");

    // Add PartyUInfo
    if (partyUInfo != NULL)
        _cpri__UpdateHash(&hashState, partyUInfo->size, partyUInfo->buffer);

    // Add PartyVInfo
    if (partyVInfo != NULL)
        _cpri__UpdateHash(&hashState, partyVInfo->size, partyVInfo->buffer);

    // Compute Hash. hLen was changed to be the smaller of bytes or hLen 
    // at the start of each iteration.
    _cpri__CompleteHash(&hashState, hLen, stream);

    // Mask off bits if the required bits is not a multiple of byte size
    if ((sizeInBits % 8) != 0)
        keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);

    return (CRYPT_RESULT)((sizeInBits + 7) / 8);
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 OpesnSSL().

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 0</td>
<td>cipher not supported</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the cipher block size in bytes</td>
</tr>
</tbody>
</table>

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     default:
35         return 0;
36     }
37 }

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>if success</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>dInSize is not a multiple of the block size</td>
</tr>
</tbody>
</table>

CRYPT_RESULT

_cpri__AESEncryptCBC(

BYTE *dOut,  // OUT:
UINT32 keySizeInBits,  // IN: key size in bits
BYTE *key,  // IN: key buffer. The size of this buffer
          // in bytes is (keySizeInBits + 7) / 8
BYTE *iv,  // IN/OUT: IV for decryption.
UINT32 dInSize,  // IN: data size (is required to be a multiple
          // of 16 bytes
BYTE *dIn  // IN/OUT: data buffer

{  
AES_KEY AesKey;
BYTE *pIv;
INT32 dSize;  // Need a signed version
int i;

pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);

if(dInSize == 0)
    return CRYPT_SUCCESS;

pAssert(dInSize <= INT32_MAX);
dSize = (INT32)dInSize;

// For CBC, the data size must be an even multiple of the
// cipher block size
if((dSize % 16) != 0)
    return CRYPT_PARAMETER;

// Create AES encrypt key schedule
if(AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
    FAIL(FATAL_ERROR_INTERNAL);

// XOR the data block into the IV, encrypt the IV into the IV
// and then copy the IV to the output
for(; dSize > 0; dSize -= 16)
{  

Family "02" Published
Level 00 Revision 00.96 Copyright © TCG 2006-2013

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>if success</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>dInSize is not a multiple of the block size</td>
</tr>
</tbody>
</table>

CRYPT_RESULT

```c
إمكانية
```
```c
BYTE   *dOut,     // OUT: the decrypted data
UINT32  keySizeInBits,  // IN: key size in bits
BYTE   *key,      // IN: key buffer. The size of this buffer
        // in bytes is (keySizeInBits + 7) / 8
BYTE   *iv,       // IN/OUT: IV for decryption. The size of
        // this buffer if 16 byte.
UINT32  dInSize,   // IN: data size
BYTE   *dIn       // IN: data buffer
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>

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    return CRYPT_SUCCESS;
139 }

141    return CRYPT_SUCCESS;
142 }
143
144    pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
145
146    if (dInSize == 0)
147        return CRYPT_SUCCESS;
148
149    pAssert(dInSize <= INT32_MAX);
150    dSize = (INT32)dInSize;
151
152    pAssert(dInSize <= INT32_MAX);
153    dSize = (INT32)dInSize;
154
155    AES_KEY AesKey;
156    INT32 dSize;  // Need a signed version of dInSize
157    int i;
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         }
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>

```c
return CRYPT_SUCCESS;
}

CRYPT_RESULT
__cpri___AESDecryptCFB(
BYTE *dOut,       // OUT: the decrypted data
UINT32 keySizeInBits, // IN: key size in bit
BYTE *key,       // IN: key buffer. The size of this buffer
    // in bytes is (keySizeInBits + 7) / 8
BYTE *iv,       // IN/OUT: IV for decryption.
UINT32 dInSize, // IN: data size
BYTE *dIn        // IN/OUT: data buffer
)
{
BYTE *pIv;
BYTE tmp[16];
int i;
BYTE *pT;
AES_KEY AesKey;
INT32 dSize;

pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);

if(dInSize == 0)
    return CRYPT_SUCCESS;
pAssert(dInSize <= INT32_MAX);
dSize = (INT32)dInSize;

    // Create AES encryption key schedule
if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
    FAIL(FATAL_ERROR_INTERNAL);

    for(; dSize > 0; dSize -= 16)
    {
        // Encrypt the IV into the temp buffer
        AES_encrypt(iv, tmp, &AesKey);
pT = tmp;
pIv = iv;
for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
            // Copy the current cipher text to IV, XOR
            // with the temp buffer and put into the output
            *dOut++ = *pT++ ^ (*pIv++ = *dIn++);

    } // If the inner loop (i loop) was smaller than 16, then dSize
    // would have been smaller than 16 and it is now negative
    // If it is negative, then it indicates how many fill bytes
    // are needed to pad out the IV for the next round.
for(; dSize < 0; dSize++)
    *pIv++ = 0;

return CRYPT_SUCCESS;
```
B.8.4.5. _cpri__AESEncryptCTR()

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT
_cpri__AESEncryptCTR(
    BYTE *dOut,              // OUT: the encrypted data
    UINT32 keySizeInBits,    // IN: key size in bits
    BYTE *key,               // IN: key buffer. The size of this buffer
                             // in bytes is (keySizeInBits + 7) / 8
    BYTE *iv,                // IN/OUT: IV for decryption.
    UINT32 dInSize,          // IN: data size
    BYTE *dIn                // IN: data buffer
) {
    BYTE tmp[16];
    BYTE *pT;
    AES_KEY AesKey;
    int i;
    INT32 dSize;

    pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
    if(dInSize == 0)
        return CRYPT_SUCCESS;
    pAssert(dInSize <= INT32_MAX);
    dSize = (INT32)dInSize;

    // Create AES encryption schedule
    if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
        FAIL(FATAL_ERROR_INTERNAL);

    for(; dSize > 0; dSize -= 16)
        { // Encrypt the current value of the IV(counter)
            AES_encrypt(iv, (BYTE *)tmp, &AesKey);

            //increment the counter (counter is big-endian so start at end)
            for(i = 15; i >= 0; i--)
                if((iv[i] += 1) != 0)
                    break;

            // XOR the encrypted counter value with input and put into output
            pT = tmp;
            for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
                *dOut++ = *dIn++ ^ *pT++;
        }
    return CRYPT_SUCCESS;
}
```
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)

```c
//% #define _cpri__AESDecryptCTR(dOut, keySize, key, iv, dInSize, dIn) \
//% _cpri__AESEncryptCTR( \
//%     ((BYTE *)dOut), \
//%     ((UINT32)keySize), \
//%     ((BYTE *)key), \
//%     ((BYTE *)iv), \
//%     ((UINT32)dInSize), \
//%     ((BYTE *)dIn)
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT
_cpri__AESEncryptECB(
    BYTE    *dOut,       // OUT: encrypted data
    UINT32   keySizeInBits, // IN: key size in bit
    BYTE    *key,        // IN: key buffer. The size of this buffer
    //     in bytes is (keySizeInBits + 7) / 8
    UINT32   dInSize,    // IN: data size
    BYTE    *dIn         // IN: clear text buffer
) {
    AES_KEY    AesKey;
    INT32      dSize;

    pAssert(dOut != NULL && key != NULL && dIn != NULL);
    if(dInSize == 0)
        return CRYPT_SUCCESS;
    pAssert(dInSize <= INT32_MAX);
    dSize = (INT32)dInSize;

    // For ECB, the data size must be an even multiple of the
    // cipher block size
    if((dSize % 16) != 0)
        return CRYPT_PARAMETER;
    // Create AES encrypting key schedule
    if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
        FAIL(FATAL_ERROR_INTERNAL);

    for(; dSize > 0; dSize -= 16)
    {
        AES_encrypt(dIn, dOut, &AesKey);
        dIn = &dIn[16];
        dOut = &dOut[16];
    }
```
B.8.4.8. _cpri__AESDecryptECB()

This function performs AES decryption using ECB (not recommended). The cipher text dIn is decrypted into dOut.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>

```c
return CRYPT_SUCCESS;
}
```

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 nth encryption of the IV, where n is the number of blocks (full or partial) in the data stream.

```c
BYTE *dOut, // OUT: the clear text data
UINT32 keySizeInBits, // IN: key size in bits
BYTE *key, // IN: key buffer. The size of this buffer in bytes is (keySizeInBits + 7) / 8
UINT32 dInSize, // IN: data size
BYTE *dIn // IN: cipher text buffer
}
```

```c
AES_KEY AesKey;
INT32 dSize;

pAssert(dOut != NULL && key != NULL && dIn != NULL);

if(dInSize == 0)
    return CRYPT_SUCCESS;

pAssert(dInSize <= INT32_MAX);
dSize = (INT32)dInSize;

// For ECB, the data size must be an even multiple of the cipher block size
if((dSize % 16) != 0)
    return CRYPT_PARAMETER;

// Create AES decryption key schedule
if (AES_set_decrypt_key(key, keySizeInBits, &AesKey) != 0)
    FAIL(FATAL_ERROR_INTERNAL);

for(; dSize > 0; dSize -= 16)
{
    AES_decrypt(dIn, dOut, &AesKey);
    dIn = &dIn[16];
    dOut = &dOut[16];
}
return CRYPT_SUCCESS;
```
Return Value | Meaning
---|---
CRYPT_SUCCESS | no non-fatal errors

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)
### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>if success</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td><code>dInSize</code> is not a multiple of the block size</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT
_cpri__SM4EncryptCBC(
    BYTE *dOut,      // OUT:
    UINT32 keySizeInBits, // IN: key size in bits
    BYTE *key,       // IN: key buffer. The size of this buffer
                     // in bytes is (keySizeInBits + 7) / 8
    BYTE *iv,       // IN/OUT: IV for decryption.
    UINT32 dInSize, // IN: data size (is required to be a multiple
                     // of 16 bytes
    BYTE *dIn       // IN/OUT: data buffer
) {
    SM4_KEY Sm4Key;
    BYTE *pIv;
    INT32 dSize; // Need a signed version
    int i;

    pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
    if(dInSize == 0)
        return CRYPT_SUCCESS;
    pAssert(dInSize <= INT32_MAX);
    dSize = (INT32)dInSize;
    // For CBC, the data size must be an even multiple of the
    // cipher block size
    if((dSize % 16) != 0)
        return CRYPT_PARAMETER;
    // Create SM4 encrypt key schedule
    if((SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
        FAIL(FATAL_ERROR_INTERNAL);
    // XOR the data block into the IV, encrypt the IV into the IV
    // and then copy the IV to the output
    for(; dSize > 0; dSize -= 16)
    {
        pIv = iv;
        for(i = 16; i > 0; i--)
            *pIv++ ^= *dIn++;
        SM4_encrypt(iv, iv, &Sm4Key);
        pIv = iv;
        for(i = 16; i > 0; i--)
            *dOut++ = *pIv++;
    }
    return CRYPT_SUCCESS;
```
B.8.5.2. _cpri__SM4DecryptCBC()

This function performs SM4 decryption in CBC chain mode. The input \textit{dIn} buffer is decrypted into \textit{dOut}.

The input \textit{iv} buffer is required to have a size equal to the block size (16 bytes). The \textit{dInSize} is required to be a multiple of the block size.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>if success</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>\textit{dInSize} is not a multiple of the block size</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT
_cpri__SM4DecryptCBC(
  BYTE  *dOut,  // OUT: the decrypted data
  UINT32  keySizeInBits,  // IN: key size in bits
  BYTE  *key,  // IN: key buffer. The size of this buffer
  BYTE  *iv,  // IN/OUT: IV for decryption. The size of
              // this buffer if 16 byte.
  UINT32  dInSize,  // IN: data size
  BYTE  *dIn  // IN: data buffer
) {
  SM4_KEY  Sm4Key;
  BYTE  *pIv;
  int  i;
  BYTE  tmp[16];
  BYTE  *pT = NULL;
  INT32  dSize;

  pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);

  if(dInSize == 0)
    return CRYPT_SUCCESS;

  pAssert(dInSize <= INT32_MAX);
  dSize = (INT32)dInSize;

  // For CBC, the data size must be an even multiple of the
  // cipher block size
  if((dSize % 16) != 0)
    return CRYPT_PARAMETER;

  // Create SM4 key schedule
  if (SM4_set_decrypt_key(key, keySizeInBits, &Sm4Key) != 0)
    FAIL(FATAL_ERROR_INTERNAL);

  // Copy the input data to a temp buffer, decrypt the buffer into the output;
  // XOR in the IV, and copy the temp buffer to the IV and repeat.
  for(; dSize > 0; dSize -= 16)
    { //
      pT = tmp;
      for(i = 16; i > 0; i--)
        *pT++ = *dIn++;
      SM4_decrypt(tmp, dOut, &Sm4Key);
      pIv = iv;
      pT = tmp;
      for(i = 16; i > 0; i--)
        { //
          *dOut++ ^= *pIv;
```
B.8.5.3. \_cpri\_SM4EncryptCFB()

This function performs SM4 encryption in CFB chain mode. The \textit{dOut} buffer receives the values encrypted \textit{dIn}. The input \textit{iv} is assumed to be the size of an encryption block (16 bytes). The \textit{iv} buffer will be modified to contain the last encrypted block.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>

```
*CpIv++ = *pT++;
}
return CRYPT_SUCCESS;
}

B.8.5.4. \_cpri\_SM4DecryptCFB()

This function performs SM4 decrypt in CFB chain mode. The \textit{dOut} buffer receives the values decrypted from \textit{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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT
_cpri__SM4DecryptCFB(
    BYTE *dOut,      // OUT: the decrypted data
    UINT32 keySizeInBits, // IN: key size in bit
    BYTE *key,       // IN: key buffer. The size of this buffer
                      // in bytes is (keySizeInBits + 7) / 8
    BYTE *iv,        // IN/OUT: IV for decryption.
    UINT32 dInSize,  // IN: data size
    BYTE *dIn        // IN/OUT: data buffer
)
```

```c

// Create SM4 encryption key schedule
if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
    FAIL(FATAL_ERROR_INTERNAL);

for(; dSize > 0; dSize -= 16)
    // Encrypt the IV into the temp buffer
    SM4_encrypt(iv, tmp, &Sm4Key);

    pT = tmp;
    pIV = iv;
    for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
        // Copy the current cipher text to IV, XOR
        // with the temp buffer and put into the output
        *dOut++ = *pT++ ^ (*pIv++ = *dIn++);

// If the inner loop (i loop) was smaller than 16, then dSize
// would have been smaller than 16 and it is now negative
// If it is negative, then it indicates how many fill bytes
// are needed to pad out the IV for the next round.
for(; dSize < 0; dSize++)
    *iv++ = 0;

return CRYPT_SUCCESS;
```

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

<table>
<thead>
<tr>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT
_cpri__SM4EncryptCTR(
    BYTE *dOut,        // OUT: the encrypted data
    UINT32 keySizeInBits, // IN: key size in bits
    BYTE *key,         // IN: key buffer. The size of this buffer
    // in bytes is (keySizeInBits + 7) / 8
    BYTE *iv,          // IN/OUT: IV for decryption.
    UINT32 dInSize,    // IN: data size
    BYTE *dIn          // IN: data buffer
)
{
    BYTE tmp[16];
    BYTE *pT;
    SM4_KEY Sm4Key;
    int i;
    INT32 dSize;
    pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
    if(dInSize == 0)
        return CRYPT_SUCCESS;
    pAssert(dInSize <= INT32_MAX);
    dSize = (INT32)dInSize;
    // Create SM4 encryption schedule
    if(SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
        FAIL(FATAL_ERROR_INTERNAL);
    for(; dSize > 0; dSize--)
    {
        // Encrypt the current value of the IV(counter)
        SM4_encrypt(iv, (BYTE *)tmp, &Sm4Key);
        // Increment the counter
        for(i = 0; i < 16; i++)
            if((iv[i] += 1) != 0)
                break;
        // XOR the encrypted counter value with input and put into output
        pT = tmp;
        for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
            *dOut++ = *dIn++ ^ *pT++;
    }
    return CRYPT_SUCCESS;
}

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)
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>

The function is defined as follows:

```c
CRYPT_RESULT _cpri__SM4EncryptECB(
BYTE       *dOut,    // OUT: encrypted data
UINT32     keySizeInBits, // IN: key size in bits
BYTE       *key,      // IN: key buffer. The size of this buffer
UINT32     dInSize,   // IN: data size
BYTE       *dIn)      // IN: clear text buffer
{
    SM4_KEY Sm4Key;
    INT32   dSize;
    pAssert(dOut != NULL && key != NULL && dIn != NULL);
    if(dInSize == 0)
        return CRYPT_SUCCESS;
    pAssert(dInSize <= INT32_MAX);
    dSize = (INT32)dInSize;
    // For ECB, the data size must be an even multiple of the cipher block size
    if((dSize % 16) != 0)
        return CRYPT_PARAMETER;
    // Create SM4 encrypting key schedule
    if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
        FAIL(FATAL_ERROR_INTERNAL);
    for(; dSize > 0; dSize -= 16)
        { SM4_encrypt(dIn, dOut, &Sm4Key);
          dIn = &dIn[16];
          dOut = &dOut[16];
        }
    return CRYPT_SUCCESS;
}
```

**B.8.5.8. _cpri__SM4DecryptECB()**

This function performs SM4 decryption using ECB (not recommended). The cipher text dIn is decrypted into dOut.

```c
CRYPT_RESULT _cpri__SM4DecryptECB(
BYTE       *dOut,    // OUT: encrypted data
UINT32     keySizeInBits, // IN: key size in bits
BYTE       *key,      // IN: key buffer. The size of this buffer
UINT32     dInSize,   // IN: data size
BYTE       *dIn)      // IN: clear text buffer
{
    SM4_KEY Sm4Key;
    INT32   dSize;
    pAssert(dOut != NULL && key != NULL && dIn != NULL);
    if(dInSize == 0)
        return CRYPT_SUCCESS;
    pAssert(dInSize <= INT32_MAX);
    dSize = (INT32)dInSize;
    // For ECB, the data size must be an even multiple of the cipher block size
    if((dSize % 16) != 0)
        return CRYPT_PARAMETER;
    // Create SM4 encrypting key schedule
    if (SM4_set_decrypt_key(key, keySizeInBits, &Sm4Key) != 0)
        FAIL(FATAL_ERROR_INTERNAL);
    for(; dSize > 0; dSize -= 16)
        { SM4_decrypt(dIn, dOut, &Sm4Key);
          dIn = &dIn[16];
          dOut = &dOut[16];
        }
    return CRYPT_SUCCESS;
}```
Return Value | Meaning
---|---
CRYPT_SUCCESS | no non-fatal errors

`_cpri_SM4DecryptECB()`

```c
CRYPT_RESULT
_cpri_SM4DecryptECB(
    BYTE *dOut,  // OUT: the clear text data
    UINT32 keySizeInBits,  // IN: key size in bit
    BYTE *key,  // IN: key buffer. The size of this buffer
    in bytes is (keySizeInBits + 7) / 8
    UINT32 dInSize,  // IN: data size
    BYTE *dIn  // IN: cipher text buffer
)
```

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 nth encryption of the IV, where n is the number of blocks (full or partial) in the data stream.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>no non-fatal errors</td>
</tr>
</tbody>
</table>
Part 4: Supporting Routines

Trusted Platform Module Library

```c
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   pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
758   if(dInSize == 0)
759       return CRYPT_SUCCESS;
760   pAssert(dInSize <= INT32_MAX);
761   dSize = (INT32)dInSize;
762   // Create SM4 key schedule
763   if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
764       FAIL(FATAL_ERROR_INTERNAL);
765   // This is written so that dIn and dOut may be the same
766   for(; dSize > 0; dSize -= 16)
767   {
768     // Encrypt the current value of the "IV"
769     SM4_encrypt(iv, iv, &Sm4Key);
770     // XOR the encrypted IV into dIn to create the cipher text (dOut)
771     pIv = iv;
772     for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
773        *dOut++ = (*pIv++ ^ *dIn++);
774   }
775   return CRYPT_SUCCESS;
776 }  
```

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().

```c
784   //%#define _cpri__SM4DecryptOFB(dOut,keySizeInBits, key, iv, dInSize, dIn)  
785   //%   _cpri__SM4EncryptOFB ((BYTE *)dOut),  
786   //%   ((UINT32)keySizeInBits),  
787   //%   ((BYTE *)key),  
788   //%   ((BYTE *)iv),  
789   //%   ((UINT32)dInSize),  
790   //%   ((BYTE *)dIn)  
791   //% )  
792   //% 
793   #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

```c
#include "CryptoEngine.h"
```

B.9.1.3 Local Functions

B.9.1.3.1 RsaPrivateExponent()

This function computes the private exponent \( d = 1 \mod (p-1)*(q-1) \) The inputs are the public modulus and one of the primes.

The results are returned in the key->private structure. The size of that structure is expanded to hold the private exponent. If the computed value is smaller than the public modulus, the private exponent is denormalized.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPTO_SUCCESS</td>
<td>private exponent computed</td>
</tr>
<tr>
<td>CRYPTO_PARAMETER</td>
<td>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</td>
</tr>
</tbody>
</table>

```c
static CRYPTO_RESULT
RsaPrivateExponent(
    RSA_KEY* key       // IN: the key to augment with the private exponent
)
{
    BN_CTX    *context;
    BIGNUM    *bnD;
    BIGNUM    *bnN;
    BIGNUM    *bnP;
    BIGNUM    *bnE;
    BIGNUM    *bnPhi;
    BIGNUM    *bnQ;
    BIGNUM    *bnQr;
    UINT32    fill;
    CRYPTO_RESULT retval = CRYPTO_SUCCESS;        // Assume success
    /* e.g., filler pattern */

    pAssert(key != NULL && key->privateKey != NULL && key->publicKey != NULL);
    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);

    /* code for computing private exponent */
}
```
```c
BN_CTX_start(context);
bnE = BN_CTX_get(context);
bnD = BN_CTX_get(context);
bnN = BN_CTX_get(context);
bnP = BN_CTX_get(context);
bnPhi = BN_CTX_get(context);
bnQ = BN_CTX_get(context);
bnQr = BN_CTX_get(context);

if (bnQr == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);

// Assume the size of the public key value is within range
pAssert(key->publicKey->size <= MAX_RSA_KEY_BYTES);

if (BN_bin2bn(key->publicKey->buffer, key->publicKey->size, bnN) == NULL
    || BN_bin2bn(key->privateKey->buffer, key->privateKey->size, bnP) == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

// If P size is not 1/2 of n size, then this is not a valid value for this
// implementation. This will also catch the case were P is input as zero.
// This generates a return rather than an assert because the key being loaded
// might be SW generated and wrong.
if(BN_num_bits(bnP) < BN_num_bits(bnN)/2) {
    retVal = CRYPT_PARAMETER;
    goto Cleanup;
}

// Get q = n/p;
if (BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
    FAIL(FATAL_ERROR_INTERNAL);

// If there is a remainder, then this is not a valid n
if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP)) {
    retVal = CRYPT_PARAMETER;  // problem may be recoverable
    goto Cleanup;
}

// Get compute Phi = (p-1)(q-1) = pq - p - q + 1 = n - p - q + 1
if (BN_copy(bnPhi, bnN) == NULL
    || !BN_sub(bnPhi, bnPhi, bnP)
    || !BN_sub(bnPhi, bnPhi, bnQ)
    || !BN_add_word(bnPhi, 1))
    FAIL(FATAL_ERROR_INTERNAL);

// Compute the multiplicative inverse
BN_set_word(bnE, key->exponent);
if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL) {
    // Going to assume that the error is caused by a bad
    // set of parameters. Specifically, an exponent that is
    // not compatible with the primes. In an implementation that
    // has better visibility to the error codes, this might be
    // refined so that failures in the library would return
    // a more informative value. Should not assume here that
    // the error codes will remain unchanged.
    retVal = CRYPT_PARAMETER;
    goto Cleanup;
}

fill = key->publicKey->size - BN_num_bytes(bnD);
BN_bn2bin(bnD, &key->privateKey->buffer[fill]);
memset(key->privateKey->buffer, 0, fill);
```
// Change the size of the private key so that it is known to contain
// a private exponent rather than a prime.
key->privateKey->size = key->publicKey->size;

Cleanup:
BN_CTX_end(context);
BN_CTX_free(context);
return retVal;

B.9.1.3.2. _cpr__TestKeyRSA()

This function computes the private exponent $d = 1 \mod (p-1)(q-1)$ The inputs are the public modulus and one of the primes or two primes.

If both primes are provided, the public modulus is computed. If only one prime is provided, the second prime is computed. In either case, a private exponent is produced and placed in $d$.

If no modular inverse exists, then CRYPT_PARAMETER is returned.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>private exponent (d) was generated</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>one or more parameters are invalid</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT
_cpr__TestKeyRSA(
TPM2B        *d,       // OUT: the address to receive the private exponent
UINT32       exponent, // IN: the public modulus
TPM2B        *publicKey, // IN/OUT: an input if only one prime is provided.
        // an output if both primes are provided
TPM2B        *prime1,  // IN: a first prime
TPM2B        *prime2   // IN: an optional second prime
)
{
    BN_CTX          *context;
    BIGNUM          *bnD;
    BIGNUM          *bnN;
    BIGNUM          *bnP;
    BIGNUM          *bnE;
    BIGNUM          *bnPhi;
    BIGNUM          *bnQ;
    UINT32          fill;

    CRYPT_RESULT    retVal = CRYPT_SUCCESS;  // Assume success

    pAssert(publicKey != NULL && prime1 != NULL);
    // Make sure that the sizes are within range

    pAssert(    prime1->size <= MAX_RSA_KEY_BYTES/2
              && publicKey->size <= MAX_RSA_KEY_BYTES);
    pAssert(    prime2 == NULL || prime2->size < MAX_RSA_KEY_BYTES/2);

    if(publicKey->size/2 != prime1->size)
        return CRYPT_PARAMETER;

    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);

    bnE = BN_CTX_get(context);  // public exponent (e)
    bnD = BN_CTX_get(context);  // private exponent (d)
    bnN = BN_CTX_get(context);  // public modulus (n)
    bnP = BN_CTX_get(context);  // prime1 (p)
```
bnPhi = BN_CTX_get(context); // (p-1)(q-1)
bnQ = BN_CTX_get(context); // prime2 (q)
bnQr = BN_CTX_get(context); // n mod p

if(bnQr == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);
if(BN_bin2bn(prime1->buffer, prime1->size, bnP) == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

// If prime2 is provided, then compute n
if(prime2 != NULL)
{
    // Two primes provided so use them to compute n
    if(BN_bin2bn(prime2->buffer, prime2->size, bnQ) == NULL)
    {
        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }

    // Make sure that the sizes of the primes are compatible
    if(BN_num_bits(bnQ) != BN_num_bits(bnP))
    {
        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }

    // Multiply the primes to get the public modulus
    if(BN_mul(bnN, bnP, bnQ, context) != 1)
        FAIL(FATAL_ERROR_INTERNAL);

    // if the space provided for the public modulus is large enough,
    // save the created value
    if(BN_num_bits(bnN) != (publicKey->size * 8))
    {
        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }

    BN_bn2bin(bnN, publicKey->buffer);
}
else
{
    // One prime provided so find the second prime by division
    BN_bin2bn(publicKey->buffer, publicKey->size, bnN);

    // Get q = n/p:
    if(BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
        FAIL(FATAL_ERROR_INTERNAL);

    // If there is a remainder, then this is not a valid n
    if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
    {
        retVal = CRYPT_PARAMETER; // problem may be recoverable
        goto Cleanup;
    }

    // Compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
    BN_copy(bnPhi, bnN);
    BN_sub(bnPhi, bnPhi, bnP);
    BN_sub(bnPhi, bnPhi, bnQ);
    BN_add_word(bnPhi, 1);

    // Compute the multiplicative inverse
    BN_set_word(bnE, exponent);
    if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
    {
        // Going to assume that the error is caused by a bad set of parameters.
        // Specifically, an exponent that is not compatible with the primes.
        // In an implementation that has better visibility to the error codes,
        // this might be refined so that failures in the library would return
        // (p-1)(q-1)
        bnQ = BN_CTX_get(context); // prime2 (q)
        bnQr = BN_CTX_get(context); // n mod p
        if(bnQr == NULL)
            FAIL(FATAL_ERROR_ALLOCATION);
        if(BN_bin2bn(prime1->buffer, prime1->size, bnP) == NULL)
            FAIL(FATAL_ERROR_INTERNAL);

        // If prime2 is provided, then compute n
        if(prime2 != NULL)
        {
            // Two primes provided so use them to compute n
            if(BN_bin2bn(prime2->buffer, prime2->size, bnQ) == NULL)
            {
                retVal = CRYPT_PARAMETER;
                goto Cleanup;
            }

            // Make sure that the sizes of the primes are compatible
            if(BN_num_bits(bnQ) != BN_num_bits(bnP))
            {
                retVal = CRYPT_PARAMETER;
                goto Cleanup;
            }

            // Multiply the primes to get the public modulus
            if(BN_mul(bnN, bnP, bnQ, context) != 1)
                FAIL(FATAL_ERROR_INTERNAL);

            // if the space provided for the public modulus is large enough,
            // save the created value
            if(BN_num_bits(bnN) != (publicKey->size * 8))
            {
                retVal = CRYPT_PARAMETER;
                goto Cleanup;
            }

            BN_bn2bin(bnN, publicKey->buffer);
        }
        else
        {
            // One prime provided so find the second prime by division
            BN_bin2bn(publicKey->buffer, publicKey->size, bnN);

            // Get q = n/p:
            if(BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
                FAIL(FATAL_ERROR_INTERNAL);

            // If there is a remainder, then this is not a valid n
            if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
            {
                retVal = CRYPT_PARAMETER; // problem may be recoverable
                goto Cleanup;
            }

            // Compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
            BN_copy(bnPhi, bnN);
            BN_sub(bnPhi, bnPhi, bnP);
            BN_sub(bnPhi, bnPhi, bnQ);
            BN_add_word(bnPhi, 1);

            // Compute the multiplicative inverse
            BN_set_word(bnE, exponent);
            if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
            {
                // Going to assume that the error is caused by a bad set of parameters.
                // Specifically, an exponent that is not compatible with the primes.
                // In an implementation that has better visibility to the error codes,
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).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>encryption complete</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>number to exponentiate is larger than the modulus</td>
</tr>
</tbody>
</table>

static CRYPT_RESULT

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  e = key->exponent;
227  if(e == 0)
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>decryption succeeded</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>the value to decrypt is larger than the modulus</td>
</tr>
</tbody>
</table>

```c
static CRYPT_RESULT

RSADP(        
    UINT32   dInOutSize, // IN/OUT: size of decrypted data
    BYTE    *dInOut,  // IN/OUT: the decrypted data
    RSA_KEY  *key     // IN: the key
)        
{
    CRYPT_RESULT retVal;

    //!!! Can put check for RSA tested here
    // Make sure that the pointers are provided and that the private key is present
    // If the private key is present it is assumed to have been created by
    // so is presumed good _cpri_PrivateExponent
    pAssert(key != NULL && dInOut != NULL &&
            key->publicKey->size == key->publicKey->size);

    // make sure that the value to be decrypted is smaller than the modulus
    // note: this check is redundant as is also performed by _math__ModExp()
    // which is optimized for use in RSA operations
    if(_math__uComp(key->(publicKey->size, key->publicKey->buffer,
                      dInOutSize, dInOut) <= 0)
        return CRYPT_PARAMETER;

    // _math__ModExp can return CRYPT_PARAMTER or CRYPT_UNDERFLOW but actual
    // underflow is not possible because everything is in the same buffer.
    retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut,
                            key->privateKey->size, key->privateKey->buffer,
                            key->publicKey->size, key->publicKey->buffer);

    // Exponentiation result is stored in-place, thus no space shortage is possible.
    pAssert(retVal != CRYPT_UNDERFLOW);

    return retVal;
}
```

#### 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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>encode successful</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>hashAlg is not valid</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>message size is too large</td>
</tr>
</tbody>
</table>

```c
static CRYPT_RESULT

OaepEncode(        
    UINT32 paddedSize,   // IN: pad value size
    BYTE   *padded,     // OUT: the pad data
    TPM_ALG_ID hashAlg, // IN: algorithm to use for padding
    const char *label,  // IN: null-terminated string (may be NULL)
    UINT32 messageSize, // IN: the message size
    BYTE   *message     // IN: the message being padded
)

#ifdef TEST_RSA
```
,   BYTE   *testSeed   // IN: optional seed used for testing.
#endif  // TEST_RSA
)
{
    UINT32   padLen;
    UINT32   dbSize;
    UINT32   i;
    BYTE     mySeed[MAX_DIGEST_SIZE];
    BYTE     *seed = mySeed;
    INT32     hLen = _cpri__GetDigestSize(hashAlg);
    BYTE     mask[MAX_RSA_KEY_BYTES];
    BYTE     *pp;
    BYTE     *pm;
    UINT32    lSize = 0;
    CRYPT_RESULT    retVal = CRYPT_SUCCESS;

    pAssert(padded != NULL && message != NULL);

    // A value of zero is not allowed because the KDF can’t produce a result
    // if the digest size is zero.
    if(hLen <= 0)
        return CRYPT_PARAMETER;

    // If a label is provided, get the length of the string, including the
    // terminator
    if(label != NULL)
        lSize = (UINT32)strlen(label) + 1;

    // Basic size check
    // messageSize <= k 2hLen 2
    if(messageSize > paddedSize - 2 * hLen - 2)
        return CRYPT_FAIL;

    // Hash L even if it is null
    // Offset into padded leaving room for masked seed and byte of zero
    pp = &padded[hLen + 1];
    retVal = _cpri__HashBlock(hashAlg, lSize, (BYTE *)label, hLen, pp);

    // concatenate PS of k mLen 2hLen 2
    padLen = paddedSize - messageSize - (2 * hLen) - 2;
    memset(pp[hLen], 0, padLen);
    pp[hLen+padLen] = 0x01;
    padLen += 1;
    memcpy(pp[hLen+padLen], message, messageSize);

    // The total size of db = hLen + pad + mSize;
    dbSize = hLen+padLen+messageSize;

    // If testing, then use the provided seed. Otherwise, use values
    // from the RNG
    ifdef    TEST_RSA
        if(testSeed != NULL)
            seed = testSeed;
        else
            _cpri__GenerateRandom(hLen, mySeed);
    endif    // TEST_RSA

    // mask = MGF1 (seed, nSize hLen 1)
    if((retVal = _cpri__MGF1(dbSize, mask, hashAlg, hLen, seed)) < 0)
        return retVal;   // Don’t expect an error because hash size is not zero
                            // was detected in the call to _cpri__HashBlock() above.

    // Create the masked db
    pm = mask;
for(i = dbSize; i > 0; i--)
    *pp++ ^= *pm++;
pp = &padded[hLen + 1];

// Run the masked data through MGF1
if((retVal = _cpri__MGF1(hLen, &padded[1], hashAlg, dbSize, pp)) < 0)
    return retVal; // Don't expect zero here as the only case for zero
    // was detected in the call to _cpri__HashBlock() above.

// Now XOR the seed to create masked seed
pp = &padded[1];
pm = seed;
for(i = hLen; i > 0; i--)
    *pp++ ^= *pm++;

// Set the first byte to zero
*padded = 0x00;
return CRYPT_SUCCESS;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>decode complete</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>the value to decode was larger than the modulus</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>the padding is wrong or the buffer to receive the results is too small</td>
</tr>
</tbody>
</table>

static CRYPT_RESULT
OaepDecode(
    UINT32      *dataOutSize, // IN/OUT: the recovered data size
    BYTE        *dataOut, // OUT: the recovered data
    TPM_ALG_ID  hashAlg, // IN: algorithm to use for padding
    const char  *label, // IN: null-terminated string (may be NULL)
    UINT32       paddedSize, // IN: the size of the padded data
    BYTE        *padded // IN: the padded data
)
{
    UINT32       dSizeSave;
    UINT32       i;
    BYTE         seedMask[MAX_DIGEST_SIZE];
    INT32        hLen = _cpri__GetDigestSize(hashAlg);
    BYTE         mask[MAX_RSA_KEY_BYTES];
    BYTE        *pp;
    BYTE        *pm;
    UINT32       lSize = 0;
    CRYPT_RESULT retVal = CRYPT_SUCCESS;

    // Unknown hash
    pAssert(hLen > 0 && dataOutSize != NULL && dataOut != NULL && padded != NULL);

    // If there is a label, get its size including the terminating 0x00
    if(label != NULL)
        lSize = (UINT32)strlen(label) + 1;

    // Set the return size to zero so that it doesn't have to be done on each
// failure
dSizeSave = *dataOutSize;
*dataOutSize = 0;

// Strange size (anything smaller can't be an OAEP padded block)
// Also check for no leading 0
if (paddedSize < ((<K>unsigned)((2 * hLen) + 2) || *padded != 0)
    return CRYPT_FAIL;

// Use the hash size to determine what to put through MGF1 in order
// to recover the seedMask
if((retVal = _cpri__MGF1(hLen, seedMask, hashAlg,
paddedSize-hLen-1, &padded[hLen+1])) < 0)
    return retVal;

// Recover the seed into seedMask
pp = &padded[1];
pm = seedMask;
for(i = hLen; i > 0; i--)
    *pm++ ^= *pp++;

// Use the seed to generate the data mask
if((retVal = _cpri__MGF1(paddedSize-hLen-1, mask, hashAlg,
    hLen, seedMask)) < 0)
    return retVal;

// Use the mask generated from seed to recover the padded data
pp = &padded[hLen+1];
pm = mask;
for(i = paddedSize-hLen-1; i > 0; i--)
    *pm++ ^= *pp++;

// Make sure that the recovered data has the hash of the label
// Put trial value in the seed mask
if((retVal = _cpri__HashBlock(hashAlg, lSize, (BYTE *)label, hLen, seedMask)) < 0)
    return retVal;

if(memcmp(seedMask, mask, hLen) != 0)
    return CRYPT_FAIL;

// find the start of the data
pm = &mask[hLen];
for(i = paddedSize-(2*hLen)-1; i > 0; i--)
{
    if(*pm++ != 0)
        break;
}
if(i == 0)
    return CRYPT_PARAMETER;

// pm should be pointing at the first part of the data
// and i is one greater than the number of bytes to move
i--;
if(i > dSizeSave)
{
    // Restore dSize
    *dataOutSize = dSizeSave;
    return CRYPT_FAIL;
}
memcpy(dataOut, pm, i);
*dataOutSize = i;
return CRYPT_SUCCESS;
B.9.1.3.7. PKSC1v1_5Encode()

This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>data encoded</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>message size is too large</td>
</tr>
</tbody>
</table>

```c
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      // move the message to the end of the buffer
479      memcpy(&padded[paddedSize - messageSize], message, messageSize);
480      // Set the first byte to 0x00 and the second to 0x02
481      *padded = 0;
482      padded[1] = 2;
483      // Fill with random bytes
484      _cpri__GenerateRandom(ps, &padded[2]);
485      // Set the delimiter for the random field to 0
486      padded[2+ps] = 0;
487      // Now, the only messy part. Make sure that all the ps bytes are non-zero
488      // In this implementation, use the value of the current index
489      for(ps++; ps > 1; ps--)
490      {
491          if(padded[ps] == 0)
492              padded[ps] = 0x55; // In the < 0.5% of the cases that the random
493              // value is 0, just pick a value to put into
494              // the spot.
495      }
496      return CRYPT_SUCCESS;
497  }
```

B.9.1.3.8. RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>decode successful</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>decoding error or results would no fit into provided buffer</td>
</tr>
</tbody>
</table>

```c
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 )
```
\begin{verbatim}

\{ 
    BOOL fail = FALSE;
    UINT32 ps;

    fail = (codedSize < 11);
    fail |= (coded[0] != 0x00) || (coded[1] != 0x02);
    for (ps = 2; ps < codedSize; ps++)
        { 
            if (coded[ps] == 0)
                break;
        }
    ps++;

    // Make sure that ps has not gone over the end and that there are at least 8 
    // bytes of pad data.
    fail |= ((ps >= codedSize) || ((ps-2) < 8));
    if (*messageSize < codedSize - ps) || fail)
        return CRYPT_FAIL;

    *messageSize = codedSize - ps;
    memcpy(message, &coded[ps], codedSize - ps);
    return CRYPT_SUCCESS;
\}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>encode successful</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>hashAlg is not a supported hash algorithm</td>
</tr>
</tbody>
</table>

\textbf{static CRYPT_RESULT}

\textbf{PssEncode}

\texttt{UINT32 eOutSize, // IN: size of the encode data buffer}
\texttt{BYTE *eOut, // OUT: encoded data buffer}
\texttt{TPM_ALG_ID hashAlg, // IN: hash algorithm to use for the encoding}
\texttt{UINT32 hashInSize, // IN: size of digest to encode}
\texttt{BYTE *hashIn // IN: the digest}
\texttt{#ifdef TEST_RSA //}
\texttt{, BYTE *saltIn // IN: optional parameter for testing}
\texttt{#endif // TEST_RSA //}

\texttt{)

\{ 
    INT32 hLen = _cpri__GetDigestSize(hashAlg);
    BYTE salt[MAX_RSA_KEY_BYTES - 1];
    UINT16 saltSize;
    BYTE *ps = salt;
    CRYPT_RESULT retVal;
    UINT16 mLen;
    CPRI_HASH_STATE hashState;

    // These are fatal errors indicating bad TPM firmware
    pAssert(eOut != NULL && hLen > 0 && hashIn != NULL );

    // Get the size of the mask
    mLen = (UINT16)(eOutSize - hLen - 1);

    // Use the maximum salt size
    saltSize = mLen - 1;
\}
\end{verbatim}
//using eOut for scratch space
// Set the first 8 bytes to zero
memset(eOut, 0, 8);

// Get set the salt
#ifdef TEST_RSA
if(saltIn != NULL)
{
    saltSize = hLen;
    memcpy(salt, saltIn, hLen);
}
#else
#endif // TEST_RSA

_cpri__GenerateRandom(saltSize, salt);

// Create the hash of the pad || input hash || salt
_cpri__StartHash(hashAlg, FALSE, &hashState);
cpri__UpdateHash(&hashState, 8, eOut);
cpri__UpdateHash(&hashState, hashInSize, hashIn);
cpri__UpdateHash(&hashState, saltSize, salt);
cpri__CompleteHash(&hashState, hLen, &eOut[eOutSize - hLen - 1]);

// Create a mask
if((retVal = _cpri__MGF1(mLen, eOut, hashAlg, hLen, &eOut[mLen])) < 0)
{
    // Currently _cpri__MGF1 is not expected to return a CRYPT_RESULT error.
pAssert(0);
    return retVal;
}

// Since this implementation uses key sizes that are all even multiples of
// 8, just need to make sure that the most significant bit is CLEAR
eOut[0] &= 0x7f;

// Before we mess up the eOut value, set the last byte to 0xbc
eOut[eOutSize - 1] = 0xbc;

// XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
eOut = &eOut[mLen - saltSize - 1];
*eOut++ ^= 0x01;

// XOR the salt data into the buffer
for(; saltSize > 0; saltSize--)
*eOut++ ^= *ps++;

// and we are done
return CRYPT_SUCCESS;

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 enforce 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 and Meaning

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>decode successful</td>
</tr>
<tr>
<td>CRYPT_SCHEME</td>
<td>hashAlg is not a supported hash algorithm</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>decode operation failed</td>
</tr>
</tbody>
</table>

```c
static CRYPT_RESULT
PssDecode(
    TPM_ALG_ID hashAlg,   // IN: hash algorithm to use for the encoding
    UINT32 dInSize,       // IN: size of the digest to compare
    BYTE *dIn,            // In: the digest to compare
    UINT32 eInSize,       // IN: size of the encoded data
    BYTE *eIn,            // IN: the encoded data
    UINT32 saltSize       // IN: the expected size of the salt
)
{
    INT32 hLen = _cpri__GetDigestSize(hashAlg);
    BYTE mask[MAX_RSA_KEY_BYTES];
    BYTE *pm = mask;
    BYTE pad[8] = {0};
    UINT32 i;
    UINT32 mLen;
    BOOL fail = FALSE;
    CRYPT_RESULT retVal;
    CPRI_HASH_STATE hashState;

    // These errors are indicative of failures due to programmer error
    pAssert(dIn != NULL && eIn != NULL);

    // check the hash scheme
    if(hLen == 0)
        return CRYPT_SCHEME;

    // most significant bit must be zero
    fail |= ((eIn[0] & 0x80) != 0);

    // last byte must be 0xbc
    fail |= (eIn[eInSize - 1] != 0xbc);

    // Use the hLen bytes at the end of the buffer to generate a mask
    // Doesn't start at the end which is a flag byte
    mLen = eInSize - hLen - 1;
    if((retVal = _cpri__MGF1(mLen, mask, hashAlg, hLen, &eIn[mLen])) < 0)
        return retVal;
    if(retVal == 0)
        return CRYPT_FAIL;

    // Clear the MSO of the mask to make it consistent with the encoding.
    mask[0] &= 0x7F;

    // XOR the data into the mask to recover the salt. This sequence
    // advances eIn so that it will end up pointing to the seed data
    // which is the hash of the signature data
    for(i = mLen; i > 0; i--)
        *pm++ ^= *eIn++;

    // Find the first byte of 0x01 after a string of all 0x00
    for(pm = mask, i = mLen; i > 0; i--)
    {
        if(*pm == 0x01)
            break;
        else
            fail |= (*pm++ != 0);
    }
```
fail |= (i == 0);

// if we have failed, will continue using the entire mask as the salt value so
// that the timing attacks will not disclose anything (I don't think that this
// is a problem for TPM applications but, usually, we don't fail so this
// doesn't cost anything).
if (fail)
{
    i = mLen;
    pm = mask;
}
else
{
    pm++;
    i--;
}
// If the salt size was provided, then the recovered size must match
fail |= (saltSize != 0 && i != saltSize);

// i contains the salt size and pm points to the salt. Going to use the input
// hash and the seed to recreate the hash in the lower portion of eIn.
_cpri__StartHash(hashAlg, FALSE, &hashState);

// add the pad of 8 zeros
_cpri__UpdateHash(&hashState, 8, pad);

// add the provided digest value
_cpri__UpdateHash(&hashState, dInSize, dIn);

// and the salt
_cpri__UpdateHash(&hashState, i, pm);

// get the result
retVal = _cpri__CompleteHash(&hashState, MAX_DIGEST_SIZE, mask);

// retVal will be the size of the digest or zero. If not equal to the indicated
// digest size, then the signature doesn't match
fail |= (retVal != hLen);
fail |= (memcmp(mask, eIn, hLen) != 0);
if (fail)
    return CRYPTO_FAIL;
else
    return CRYPTO_SUCCESS;

B.9.1.3.11. PKSC1v1_5SignEncode()

Encode a message using PKCS1v1(). 5 method.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPTO_SUCCESS</td>
<td>encode complete</td>
</tr>
<tr>
<td>CRYPTO_SCHEME</td>
<td>hashAlg is not a supported hash algorithm</td>
</tr>
<tr>
<td>CRYPTO_PARAMETER</td>
<td>eOutSize is not large enough or hInSize does not match the digest size of hashAlg</td>
</tr>
</tbody>
</table>

static CRYPTO_RESULT
RSASSA_Encode(
    UINT32           eOutSize,     // IN: the size of the resulting block
    BYTE            *eOut,        // OUT: the encoded block
    TPM_ALG_ID      hashAlg,      // IN: hash algorithm for PKSC1v1_5
    UINT32           hInSize,      // IN: size of hash to be signed
    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 'hashAl' 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  
742     // Start filling  
743     *eOut++ = 0; // initial byte of zero  
744     *eOut++ = 1; // byte of 0x01  
745     for(; fillSize > 0; fillSize--)  
746         *eOut++ = 0xff; // bunch of 0xff  
747     *eOut++ = 0; // another 0  
748     for(; derSize > 0; derSize--)  
749         *eOut++ = *der++; // copy the DER  
750     for(; hInSize > 0; hInSize--)  
751         *eOut++ = *hIn++; // copy the hash  
752     return CRYPT_SUCCESS;  
753 }  
754  
755  
B.9.1.3.12. RSASSA_Decode()  
756  
This function performs the RSASSA decoding of a signature.  
757  
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>decode successful</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>decode unsuccessful</td>
</tr>
<tr>
<td>CRYPT_SCHEME</td>
<td>hasAlg is not supported</td>
</tr>
</tbody>
</table>

758  static CRYPT_RESULT  
759  RSASSA_Decode(  
760      TPM_ALG_ID       hashAlg,  // IN: hash algorithm to use for the encoding  
761      UINT32           hInSize,  // IN: size of the digest to compare  
762      BYTE            *hIn,    // In: the digest to compare  
763      UINT32           eInSize,  // IN: size of the encoded data  
764      BYTE            *eIn;    // IN: the encoded data  
765  )  
766  {  
767      BOOL            fail = FALSE;  
768      BYTE            *der;  
769      INT32           derSize = _cpri__GetHashDER(hashAlg, &der);  
770      INT32           hashSize = _cpri__GetDigestSize(hashAlg);  
771      INT32           fillSize;  
772  
773      pAssert(hIn != NULL && eIn != NULL);  
774  
775      // Can't use this scheme if the algorithm doesn't have a DER string
if (derSize == 0 || (unsigned)hashSize != hInSize)
    return CRYPTO_SCHEME;

// Make sure that this combination will fit in the provided space
// Since no data movement takes place, can just walk through this
// and accept nearly random values. This can only be called from
// __cpri__ValidateSignature() so eInSize is known to be in range.
fillSize = eInSize - derSize - hashSize - 3;

// Start checking
fail |= (*eIn++ != 0); // initial byte of zero
fail |= (*eIn++ != 1); // byte of 0x01
for (; fillSize > 0; fillSize--)
    fail |= (*eIn++ != 0xff); // bunch of 0xff
fail |= (*eIn++ != 0); // another 0
for (; derSize > 0; derSize--)
    fail |= (*eIn++ != *der++); // match the DER
for (; hInSize > 0; hInSize--)
    fail |= (*eIn++ != *hIn++); // match the hash
if (fail)
    return CRYPTO_FAIL;
return CRYPTO_SUCCESS;

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.

BOOL __cpri__RsaStartup()
{
    return TRUE;
}

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 RSAEP. This is because the high order bytes of dIn might have a numeric value that is greater than
the value of the key modulus. If this had low-order zeros added, it would have a numeric value larger than the
modulus even though it started out with a lower numeric value.
Return Value | Meaning
--- | ---
CRYPT_SUCCESS | encryption complete
CRYPT_PARAMETER | cOutSize is too small (must be the size of the modulus)
CRYPT_SCHEME | padType is not a supported scheme

```c
CRYPT_RESULT _cpr__EncryptRSA(
UINT32  *cOutSize, // OUT: the size of the encrypted data
BYTE     *cOut,     // OUT: the encrypted data
RSA_KEY  *key,      // IN: the key to use for encryption
TPM_ALG_ID padType, // IN: the type of padding
UINT32   dInSize,   // IN: the amount of data to encrypt
BYTE     *dIn,      // IN: the data to encrypt
TPM_ALG_ID hashAlg, // IN: in case this is needed
const char *label   // IN: in case it is needed
) {
CRYPT_RESULT  retVal = CRYPT_SUCCESS;
pAssert(cOutSize != NULL);

// All encryption schemes return the same size of data
if(*cOutSize < key->publicKey->size)
   return CRYPT_PARAMETER;
   *cOutSize = key->publicKey->size;

switch (padType)
{
case TPM_ALG_NULL: // 'raw' encryption
   {
      // dIn can have more bytes than cOut as long as the extra bytes
      // are zero
      for(; dInSize > *cOutSize; dInSize--)
      {
         if(*dIn++ != 0)
            return CRYPT_PARAMETER;
      }
   }
   // If dIn is smaller than cOut, fill cOut with zeros
   if(dInSize < *cOutSize)
      memset(cOut, 0, *cOutSize - dInSize);
   // Copy the rest of the value
   memcpy(&cOut[*cOutSize-dInSize], dIn, dInSize);
   // If the size of dIn is the same as cOut dIn could be larger than
   // the modulus. If it is, then RSAEP() will catch it.
   break;
   case TPM_ALG_RSAES:
      retVal = RSAES_PKCS1v1_5Encode(*cOutSize, cOut, dInSize, dIn);
      break;
   case TPM_ALG_OAEP:
      retVal = OaepEncode(*cOutSize, cOut, hashAlg, label, dInSize, dIn
#ifdef TEST_RSA
   ,NULL
#else
#endif
   break;
   default:
      return CRYPTO_SCHEME;

   // All the schemes that do padding will come here for the encryption step
   // Check that the Encoding worked
```


if(retVal != CRYPT_SUCCESS)
    return retVal;

// Padding OK so do the encryption
return RSAEP(*cOutSize, cOut, key);
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>successful completion</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>cInSize is not the same as the size of the public modulus of key; or numeric value of the encrypted data is greater than the modulus</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>dOutSize is not large enough for the result</td>
</tr>
<tr>
<td>CRYPT_SCHEME</td>
<td>padType is not supported</td>
</tr>
</tbody>
</table>

CRYPT_RESULT _cpri__DecryptRSA()

UINT32 *dOutSize, // OUT: the size of the decrypted data
BYTE *dOut, // OUT: the decrypted data
RSA_KEY *key, // IN: the key to use for decryption
TPM_ALG_ID padType, // IN: the type of padding
UINT32 cInSize, // IN: the amount of data to decrypt
BYTE *cIn, // IN: the data to decrypt
TPM_ALG_ID hashAlg, // IN: in case this is needed for the scheme
const char *label // IN: in case it is needed for the scheme

// Make sure that the necessary parameters are provided
pAssert(cIn != NULL && dOut != NULL && dOutSize != NULL && key != NULL);

// Size is checked to make sure that the decryption works properly
if(cInSize != key->publicKey->size)
    return CRYPT_PARAMETER;

// For others that do padding, do the decryption in place and then
// go handle the decoding.
if((retVal = RSADP(cInSize, cIn, key)) != CRYPT_SUCCESS)
    return retVal; // Decryption failed

// Remove padding
switch (padType)
{
    case TPM_ALG_NULL:
        if(*dOutSize < key->publicKey->size)
            return CRYPT_FAIL;
        *dOutSize = key->publicKey->size;
        memcpy(dOut, cIn, *dOutSize);
        return CRYPT_SUCCESS;
    case TPM_ALG_RSAES:
        return RSAES_Decode(dOutSize, dOut, cInSize, cIn);
        break;
    case TPM_ALG_OAEP:
        return OaepDecode(dOutSize, dOut, hashAlg, label, cInSize, cIn);
        break;
    default:
return CRYPT_SCHEME;
break;
}

B.9.1.4.4. _cpri__SignRSA()

This function is used to generate an RSA signature of the type indicated in \textit{scheme}.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>sign operation completed normally</td>
</tr>
<tr>
<td>CRYPT_SCHEME</td>
<td>\textit{scheme} or \textit{hashAlg} are not supported</td>
</tr>
<tr>
<td>CRYPT_PARAMETER</td>
<td>\textit{hInSize} does not match \textit{hashAlg} (for RSASSA)</td>
</tr>
</tbody>
</table>

CRYPT_RESULT

.cpri__SignRSA(*sigOutSize, /* OUT: size of signature*/
               *sigOut, /* OUT: signature*/
               key, /* key to use*/
               scheme, /* IN: the scheme to use*/
               hashAlg, /* IN: hash algorithm for PKCS1v1_5*/
               hInSize, /* IN: size of digest to be signed*/
               hIn /* IN: digest buffer*/)

{ return CRYPT_RESULT retVal;

  // Parameter checks
  pAssert(sigOutSize != NULL && sigOut != NULL && key != NULL && hIn != NULL);

  // For all signatures the size is the size of the key modulus
  *sigOutSize = key->publicKey->size;

  switch (scheme)
  {
    case TPM_ALG_NULL:
      *sigOutSize = 0;
      return CRYPT_SUCCESS;
    case TPM_ALG_RSAPSS:
      // PssEncode can return CRYPT_PARAMETER
      retVal = PssEncode(*sigOutSize, sigOut, hashAlg, hInSize, hIn
      #ifdef TEST_RSA
      , NULL
      #endif
      break;
    case TPM_ALG_RSASSA:
      // RSASSA_Encode can return CRYPT_PARAMETER or CRYPT_SCHEME
      retVal = RSASSA_Encode(*sigOutSize, sigOut, hashAlg, hInSize, hIn);
      break;
    default:
      return CRYPT_SCHEME;
  } if(retVal != CRYPT_SUCCESS)
  return retVal;
  // Do the encryption using the private key
  // RSADP can return CRYPT_PARAMETER
  return RSADP(*sigOutSize,sigOut, key);
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>the signature checks</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>the signature does not check</td>
</tr>
<tr>
<td>CRYPT_SCHEME</td>
<td>unsupported scheme or hash algorithm</td>
</tr>
</tbody>
</table>

```
CRYPT_RESULT _cpri__ValidateSignatureRSA(RSA_KEY *key, TPM_ALG_ID scheme, TPM_ALG_ID hashAlg, UINT32 hInSize, BYTE *hIn, UINT32 sigInSize, BYTE *sigIn, UINT16 saltSize)
{
    CRYPT_RESULT retVal;
    // Fatal programming errors
    pAssert(key != NULL && sigIn != NULL && hIn != NULL);
    // Errors that might be caused by calling parameters
    if(sigInSize != key->publicKey->size)
        return CRYPT_FAIL;
    // Decrypt the block
    if((retVal = RSAEP(sigInSize, sigIn, key)) != CRYPT_SUCCESS)
        return CRYPT_FAIL;
    switch (scheme)
    {
    case TPM_ALG_NULL:
        return CRYPT_SCHEME;
        break;
    case TPM_ALG_RSAPSS:
        return PssDecode(hashAlg, hInSize, hIn, sigInSize, sigIn, saltSize);
        break;
    case TPM_ALG_RSASSA:
        return RSASSA_Decode(hashAlg, hInSize, hIn, sigInSize, sigIn);
        break;
    default:
        break;
    }
    return CRYPT_SCHEME;
}
```

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 generation process
1004    TPM2B *seed, // IN: the seed to use
1005    const char *label, // IN: A label for the generation process.
1006    TPM2B *extra, // IN: Party 1 data for the KDF
1007    TPM2B *seed, // IN: the seed to use
1008    UINT32 *counter // IN/OUT: Counter value to allow KDF iteration to be propagated across multiple routines
1009 )
1010 {
1011    UINT32 lLen; // length of the label
1012    /* (counting the terminating 0);*/
1013    UINT16 digestSize = _cpri__GetDigestSize(hashAlg);
1014    TPM2B_HASH_BLOCK oPadKey;
1015    UINT32 outer;
1016    UINT32 inner;
1017    BYTE swapped[4];
1018    CRYPT_RESULT retVal;
1019    int i, fill;
1020    const static char defaultLabel[] = "RSA key";
1021    BYTE *pb;
1022
1023    BYTE h1[MAX_HASH_STATE_SIZE]; // contains the hash of the HMAC key w/ iPad
1024    BYTE h2[MAX_HASH_STATE_SIZE]; // contains the hash of the HMAC key w/ oPad
1025    BYTE h[MAX_HASH_STATE_SIZE]; // the working hash context
1026
1027    BIGNUM *bnP;
1028    BIGNUM *bnQ;
1029    BIGNUM *bnT;
1030    BIGNUM *bnE;
1031    BIGNUM *bnN;
1032    BN_CTX *context;
1033    UINT32 rem;
1034
1035    // Make sure that hashAlg is valid hash
1036    pAssert(digestSize != 0);
1037    // if present, use externally provided counter
1038    if(counter != NULL)
1039        outer = *counter;
1040    else
1041        outer = 1;
1042
1043    // Validate exponent
1044    UINT32_TO_BYTE_ARRAY(e, swapped);
// Need to check that the exponent is prime and not less than 3
if (e != 0 && (e < 3 || !_math__IsPrime(4, swapped))
    return CRYPT_FAIL;

// Get structures for the big number representations
context = BN_CTX_new();
if (context == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);
BN_CTX_start(context);
bnP = BN_CTX_get(context);
bnQ = BN_CTX_get(context);
bnT = BN_CTX_get(context);
bnE = BN_CTX_get(context);
bnN = BN_CTX_get(context);
if (bnN == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

// Set Q to zero. This is used as a flag. The prime is computed in P. When a
// new prime is found, Q is checked to see if it is zero. If so, P is copied
// to Q and a new P is found. When both P and Q are non-zero, the modulus and
// private exponent are computed and a trial encryption/decryption is
// performed. If the encrypt/decrypt fails, assume that at least one of the
// primes is composite. Since we don't know which one, set Q to zero and start
// over and find a new pair of primes.
BN_zero(bnQ);

// Need to have some label
if (label == NULL)
    label = (const char *)&defaultLabel;

// Get the label size
for (lLen = 0; label[lLen++] != 0;);

// Start the hash using the seed and get the intermediate hash value
_cpri__StartHMAC(hashAlg, FALSE, &h1, seed->size, seed->buffer, &oPadKey.b);
_cpri__StartHash(hashAlg, FALSE, &h2);
_cpri__UpdateHash(&h2, oPadKey.b.size, oPadKey.b.buffer);

n->size = keySizeInBits / 8;
pAssert(n->size <= MAX_RSA_KEY_BYTES);
p->size = n->size / 2;
if (e == 0)
    e = RSA_DEFAULT_PUBLIC_EXPONENT;
BN_set_word(bnE, e);

// The first test will increment the counter from zero.
for(outer += 1; outer != 0; outer++)
{
    if(_plat__IsCanceled())
    {
        retVal = CRYPT_CANCEL;
        goto Cleanup;
    }

    // Need to fill in the candidate with the hash
    fill = digestSize;
pb = p->buffer;

    // Reset the inner counter
    inner = 0;
    for(i = p->size; i > 0; i -= digestSize)
    {
        inner++;
        // Initialize the HMAC with saved state
        _cpri__CopyHashState(&h, &h1);
1121    // Hash the inner counter (the one that changes on each HMAC iteration)
1122    UINT32_TO_BYTE_ARRAY(inner, swapped);
1123    _cpri__UpdateHash(&h, 4, swapped);
1124    _cpri__UpdateHash(&h, lLen, (BYTE *)label);
1125
1126    // Is there any party 1 data
1127    if(extra != NULL)
1128        _cpri__UpdateHash(&h, extra->size, extra->buffer);
1129
1130    // Include the outer counter (the one that changes on each prime
1131    // prime candidate generation
1132    UINT32_TO_BYTE_ARRAY(outer, swapped);
1133    _cpri__UpdateHash(&h, 4, swapped);
1134    _cpri__UpdateHash(&h, 2, (BYTE *)&keySizeInBits);
1135    if(i < fill)
1136        fill = i;
1137    _cpri__CompleteHash(&h, fill, pb);
1138
1139    // Restart the oPad hash
1140    _cpri__CopyHashState(&h, &h2);
1141
1142    // Add the last hashed data
1143    _cpri__UpdateHash(&h, fill, pb);
1144
1145    // gives a completed HMAC
1146    _cpri__CompleteHash(&h, fill, pb);
1147    pb += fill;
1148 }
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
1170    // Make sure that the prime candidate (p) is not divisible by the exponent
1171    // and that (p-1) is not divisible by the exponent
1172    // Get the remainder after dividing by the modulus
1173    rem = BN mod_word(bnP, e);
1174    if(rem == 0)  // evenly divisible so add two keeping the number odd and
1175        if(BN_add_word(bnP, 2);
1176    else if(rem == 1)  // leaves a remainder of 1 so subtract two keeping the
1177        if(BN num_bits(bnT) < 100)  <Q>    // Difference has to be at least 100 bits
1178            continue;
1179}
1180
1181    // Have a candidate, check for primality
1182    if((retVal = (CRYPT_RESULT)BN_is_prime_ex(bnP,
1183        BN_prime_checks, NULL, NULL)) < 0)
1184        FAIL(FATAL_ERROR_INTERNAL);
1185
1186    if(retVal != 1)
1187        continue;
// Found a prime, is this the first or second.
if (BN_is_zero(bnQ))
{
    // copy p to q and compute another prime in p
    BN_copy(bnQ, bnP);
    continue;
}

// Form the public modulus
BN_mul(bnN, bnP, bnQ, context);
if (BN_num_bits(bnN) != keySizeInBits)
    FAIL(FATAL_ERROR_INTERNAL);

// Save the public modulus
BnTo2B(n, bnN, 0);  // Fills the buffer with the correct size

// And one prime
BnTo2B(p, bnP, 0);

// Finish by making sure that we can form the modular inverse of PHI
// with respect to the public exponent
// Compute PHI = (p - 1)(q - 1) = n - p - q + 1
// Make sure that we can form the modular inverse
BN_sub(bnT, bnN, bnP);
BN_sub(bnT, bnT, bnQ);
BN_add_word(bnT, 1);

// find d such that (Phi * d) mod e == 1
// If there isn't then we are broken because we took the step
// of making sure that the prime != 1 mod e so the modular inverse
// must exist
if (BN_mod_inverse(bnT, bnE, bnT, context) == NULL || BN_is_zero(bnT))
    FAIL(FATAL_ERROR_INTERNAL);

// Do a trial encryption and decryption of the seed to see if this
// gives a valid result
BN_bin2bn(seed->buffer, (n->size)-1, bnP);
BN_copy(bnQ, bnP);
BN_mod_exp(bnQ, bnQ, bnE, bnN, context);
BN_mod_exp(bnQ, bnQ, bnT, bnN, context);
if (BN_cmp(bnP, bnQ) != 0) // Trial encrypt decrypt failed. Start
    // over with new primes
    BN_zero(bnQ);
    continue;
}

retVal = CRYPT_SUCCESS;
goto Cleanup;
}

retVal = CRYPT_FAIL;

Cleanup:
// Close out the hash sessions
_cpri__CompleteHash(&h2, 0, NULL);
_cpri__CompleteHash(&h1, 0, NULL);

// Free up allocated BN values
BN_CTX_end(context);
BN_CTX_free(context);
if (counter != NULL)
    *counter = outer;
return retVal;
#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 for parameterization of the Sieve and RNG used by the RSA module

```c
#define PRIME_DIFF_TABLE_512_BYTE_PAGES 13
extern BYTE primeDiffTable[PRIME_DIFF_TABLEgetBytes];
```

This determines the number of bits in the sieve field. This must be a power of two.
#define FIELD_POWER 14 // This is the only value in this group that should be changed
#define FIELD_BITS (1 << FIELD_POWER)
#define MAX_FIELD_SIZE ((FIELD_BITS / 8) + 1)

This is the pre-sieved table. It already has the bits for multiples of 3, 5, and 7 cleared.

#define SEED_VALUES_SIZE 105
const extern BYTE seedValues[SEED_VALUES_SIZE];

This allows determination of the number of bits that are set in a byte without having to count them individually.

const extern BYTE bitsInByte[256];

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.

typedef struct {
    UINT16 lastPrime;
    UINT16 index;
    UINT16 final;
} PRIME_ITERATOR;

#ifdef RSA_INSTRUMENT
    # define INSTRUMENT_SET(a, b) ((a) = (b))
    # define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
    # define INSTRUMENT_INC(a) (a) = (a) + 1
    extern UINT32 failedAtIteration[10];
    extern UINT32 MillerRabinTrials;
    extern UINT32 totalFieldsSieved;
    extern UINT32 emptyFieldsSieved;
    extern UINT32 noPrimeFields;
    extern UINT32 primesChecked;
    extern UINT16 lastSievePrime;
#else
    #define INSTRUMENT_SET(a, b)
    #define INSTRUMENT_ADD(a, b)
    #define INSTRUMENT_INC(a)
#endif
#ifdef RSA_DEBUG
    extern UINT16 defaultFieldSize;
#else
    #define defaultFieldSize MAX_FIELD_SIZE
#endif

B.9.2.3. RSAKeySieve.c

/* (Copyright)
Microsoft Confidential Contribution to a TCG Specification or Design Guide under Article 15 of "The Bylaws of the Trusted Computing Group" as Amended through March 20, 2003
*/

/** Introduction */
#include "CryptoEngine.h"
#ifdef RSA_KEY_SIEVE
#else
#endif
14 // This next line will show up in the header file for this code. It will
15 // make the local functions public when debugging.
16 //%#ifdef   RSA_DEBUG
17
18 ///** Bit Manipulation Functions
19 ///** Introduction
20 /// These functions operate on a bit array. A bit array is an array of
21 /// bytes with the 0th byte being the byte with the lowest memory address.
22 /// Within the byte, bit 0 is the least significant bit.
23 ///
24 ///** ClearBit()
25 /// This function will CLEAR a bit in a bit array.
26 void
27 ClearBit(
28 unsigned char *a,               // IN: A pointer to an array of bytes
29     int  i                          // IN: the number of the bit to CLEAR
30    )
31 {
32     a[i >> 3] & 0xff ^ (1 << (i & 7));
33 }

B.9.2.3.1.1. SetBit()

Function to SET a bit in a bit array.

void
SetBit(
    unsigned char *a,              // IN: A pointer to an array of bytes
    int  i                    // IN: the number of the bit to SET
    )
{
    a[i >> 3] |= (1 << (i & 7));
}

B.9.2.3.1.2. IsBitSet()

Function to test if a bit in a bit array is SET.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bit is CLEAR</td>
</tr>
<tr>
<td>1</td>
<td>bit is SET</td>
</tr>
</tbody>
</table>

UINT32
IsBitSet(
    unsigned char *a,          // IN: A pointer to an array of bytes
    int  i                     // IN: the number of the bit to test
    )
{
    return ((a[i >> 3] & (1 << (i & 7))) != 0);
}

B.9.2.3.1.3. BitsInArray()

This function counts the number of bits set in an array of bytes.

int
BitsInArray(
    unsigned char *a,           // IN: A pointer to an array of bytes
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 return FALSE.

```c
UINT32 FindNthSetBit(
    const UINT16     aSize,   // IN: the size of the array to check
    const BYTE      *a,       // IN: the array to check
    const UINT32     n         // IN, the number of the SET bit
) {
    UINT32       i;
    const BYTE  *pA = a;
    UINT32       retValue;
    BYTE         sel;

    //find the bit
    for(i = 0; i < n; i += bitsInByte[*pA++]);

    // The chosen bit is in the byte that was just accessed
    // Compute the offset to the start of that byte
    pA--;
    retValue = (pA - a) * 8;

    // Subtract the bits in the last byte added.
    i -= bitsInByte[*pA];

    // Now process the byte, one bit at a time.
    for(sel = *pA; sel != 0 ; sel = sel >> 1) {
        if(sel & 1) {
            i += 1;
            if(i == n) return retValue;
        }
        retValue += 1;
    }
    FAIL(FATAL_ERROR_INTERNAL);
    return 0;  // This is just to keep the compiler from complaining
}
```

B.9.2.3.2. **Miscellaneous Functions**

**B.9.2.3.2.1. RandomForRsa()**

This function uses a special form of `KDFa()` to produces a pseudo random sequence. It's input is a structure that contains pointers to a pre-computed set of hash contexts that are set up for the HMAC computations using the seed.

This function will test that ktx. outer will not wrap to zero if incremented. If so, the function returns FALSE. Otherwise, the ktx. outer is incremented before each number is generated.
void RandomForRsa(
    KDFa_CONTEXT *ktx,       // IN: a context for the KDF
    const char *label,       // IN: a use qualifying label
    TPM2B *p                 // OUT: the pseudo random result
)
{
    INT16 i;
    UINT32 inner;
    BYTE swapped[4];
    UINT16 fill;
    BYTE *pb;
    UINT16 lLen = 0;
    UINT16 digestSize = _cpri__GetDigestSize(ktx->hashAlg);
    BYTE h[MAX_HASH_STATE_SIZE];     // the working hash context

    if(label != NULL)
        for(lLen = 0; label[lLen++];);
    fill = digestSize;
    pb = p->buffer;
    inner = 0;
    *(ktx->outer) += 1;
    for(i = p->size; i > 0; i -= digestSize)
        inner++;

    // Initialize the HMAC with saved state
    _cpri__CopyHashState(&h, &(ktx->iPadCtx));

    // Hash the inner counter (the one that changes on each HMAC iteration)
    UINT32_TO_BYTE_ARRAY(inner, swapped);
    _cpri__UpdateHash(&h, 4, swapped);
    if(lLen != 0)
        _cpri__UpdateHash(&h, lLen, (BYTE *)label);

    // Is there any party 1 data
    if(ktx->extra != NULL)
        _cpri__UpdateHash(&h, ktx->extra->size, ktx->extra->buffer);

    // Include the outer counter (the one that changes on each prime candidate generation)
    UINT32_TO_BYTE_ARRAY(*(ktx->outer), swapped);
    _cpri__UpdateHash(&h, 4, swapped);
    _cpri__UpdateHash(&h, 2, (BYTE *)&ktx->keySizeInBits);
    if(i < fill)
        fill = i;
    _cpri__CompleteHash(&h, fill, pb);

    // Restart the oPad hash
    _cpri__CopyHashState(&h, &(ktx->oPadCtx));

    // Add the last hashed data
    _cpri__UpdateHash(&h, fill, pb);

    // gives a completed HMAC
    _cpri__CompleteHash(&h, fill, pb);
    pb += fill;
}

return;
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.

```c
UINT32  MillerRabinRounds(
    UINT32      bits   // IN: Number of bits in the RSA prime
)
{
    if(bits < 511) return 8; // don't really expect this
    if(bits < 1536) return 5; // for 512 and 1K primes
    return 4; // for 3K public modulus and greater
}
```

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. I all likelihood, if the number is not prime, the first test fails.

If a KDFa(), PRNG context is provide ('ktx'), then it is used to provide the random values. Otherwise, the random numbers are retrieved from the random number generator.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>probably prime</td>
</tr>
<tr>
<td>FALSE</td>
<td>composite</td>
</tr>
</tbody>
</table>

```c
BOOL  MillerRabin(
    BIGNUM              *bnW,
    int                  iterations,
    KDFa_CONTEXT        *ktx,
    BN_CTX              *context
)
{
    BN_CTX_start(context);
    bnWm1 = BN_CTX_get(context);
    bnB = BN_CTX_get(context);
    bnZ = BN_CTX_get(context);
    bnM = BN_CTX_get(context);
    if(bnM == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    // Let a be the largest integer such that 2^a divides w-1.
    pAssert(BN_is_bit_set(bnW, 0));
    INSTRUMENT_INC(MillerRabinTrials); // Instrumentation
    BN_CTX_start(context);
    bnWm1 = BN_CTX_get(context);
    bnB = BN_CTX_get(context);
    bnZ = BN_CTX_get(context);
    bnM = BN_CTX_get(context);
    if(bnM == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_copy(bnWm1, bnW);
    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
262 // 4.6 Return COMPOSITE.
263 step4point6:
264 if(i > 9)
265 INSTRUMENT_INC(failedAtIteration[9]);
266 else
267 INSTRUMENT_INC(failedAtIteration[i]);
268 goto end;
step4point7:
    continue;
// 5. Return PROBABLY PRIME
ret = TRUE;
end:
    BN_CTX_end(context);
    return ret;
}

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.

UINT16
NextPrime(
    PRIME_ITERATOR      *iter
)
{
    if (iter->index >= iter->final)
        return (iter->lastPrime = 0);
    return (iter->lastPrime += primeDiffTable[iter->index++]);
}

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.

UINT16
AdjustNumberOfPrimes(
    UINT16      p
)
{
    p = ((p + 511) / 512) * 512;
    if (p == 0 || p > PRIME_DIFF_TABLE_BYTES)
        p = PRIME_DIFF_TABLE_BYTES;
    return p;
}

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.

UINT16
PrimeInit(
    UINT16           first,   // IN: the initial prime
    PRIME_ITERATOR  *iter,   // IN/OUT: the iterator structure
    UINT16           primes   // IN: the table length
)
{
    iter->lastPrime = 1;
iter->index = 0;
iter->final = AdjustNumberOfPrimes(primes);
while(iter->lastPrime < first)
    NextPrime(iter);
return iter->lastPrime;
}

B.9.2.3.2.7. SetDefaultNumberOfPrimes()

This macro sets the default number of primes to the indicated value.

//%#define SetDefaultNumberOfPrimes(p) (primeTableBytes = AdjustNumberOfPrimes(p))

B.9.2.3.2.8. IsPrimeWord()

Checks to see if a UINT32 is prime

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>number is prime</td>
</tr>
<tr>
<td>FAIL</td>
<td>number is not prime</td>
</tr>
</tbody>
</table>

BOOL
IsPrimeWord(
    UINT32      p    // IN: number to test
)
{
    if defined RSA_KEY_SIEVE && (PRIME_DIFF_TABLE_BYTES >= 6542)
    
    UINT32      test;
    UINT32      index;
    UINT32      stop;

    if((p & 1) == 0)
        return FALSE;
    if(p == 1 || p == 3)
        return TRUE;

    // Get a high value for the stopping point
    for(index = p, stop = 0; index; index >>= 2)
        stop = (stop << 1) + 1;
    stop++;

    // If the full prime difference value table is present, can check here
    test = 3;
    for(index = 1; index < PRIME_DIFF_TABLE_BYTES; index += 1)
        
        if((p % test) == 0)
            return (p == test);
    return TRUE;

    test += primeDiffTable[index];
    return TRUE;

    #else

    BYTE        b[4];
    if(p = RSA_DEFAULT_PUBLIC_EXPONENT || p == 1 || p == 3)
        return TRUE;
    if((p & 1) == 0)
return FALSE;
UINT32_TO_BYTE_ARRAY(p,b);
return _math__IsPrime(4, b);
#endif
}

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

void
SetDefaultSieveFieldSize(
    UINT16      f       // IN: the size of the sieve field. This should be
                   //   a power of two. The actual field size will be one
                   //   byte larger.
)
{
    UINT16      i;
    if(f == 0 || f > MAX_FIELD_SIZE)
        defaultFieldSize = MAX_FIELD_SIZE;
    else
    {
        for(i = 1, f >>= 1; f != 0; f >>= 1, i <<= 1);
        defaultFieldSize = i + 1;
    }
}

typedef struct {
    UINT16     prime;
    UINT16     count;
} SIEVE_MARKS;
const SIEVE_MARKS sieveMarks[5] = {
    {31, 7}, {73, 5}, {241, 4}, {1621, 3}, {UINT16_MAX, 2}};

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).

UINT16
PrimeSieve(
    BIGNUM      *bnN,      // IN/OUT: number to sieve
    UINT16      fieldSize,  // IN: size of the field area in bytes
    BYTE        *field,     // IN: field
    UINT16      primes,     // IN: the number of primes to use
)
{
    UINT16     i;
    UINT32     j;
    UINT16     fieldBits = fieldSize * 8;
    UINT32     r;
    const BYTE  *p1;
    BYTE       *p2;
    PRIME_ITERATOR   iter;
    UINT32     adjust;
    UINT32     mark = 0;
    UINT16     count = sieveMarks[0].count;
    UINT16     stop = sieveMarks[0].prime;
    UINT32     composite;
// UINT64 test;   //DEBUG

pAssert(field != NULL && bnN != NULL);

// Need to have a field that has a size of 2^n + 1 bytes
pAssert(BitsInArray((BYTE *)fieldSize, 2) == 2);

Primes = AdjustNumberOfPrimes(primes);

// If the remainder is odd, then subtracting the value
// will give an even number, but we want an odd number,
// so subtract the 105+rem. Otherwise, just subtract
// the even remainder.
adjust = BN_mod_word(bnN, 105);
if (adjust & 1)
    adjust += 105;

// seed the field
// This starts the pointer at the nearest byte to the input value
pl = &seedValues[adjust / 16];

// Reduce the number of bytes to transfer by the amount skipped
j = sizeof(seedValues) - adjust / 16;
adjust = adjust % 16;
BN_sub_word(bnN, adjust);
adjust >>= 1;

// This offsets the field
p2 = field;
for (i = fieldSize; i > 0; i--)
    {*p2++ = *pl++;
     if (--j == 0)
        {j = sizeof(seedValues);
         pl = seedValues;
        }
    }

// Mask the first bits in the field and the last byte in order to eliminate
// bytes not in the field from consideration.
field[0] &= 0xff << adjust;
field[fieldSize - 1] &= 0xff >> (8 - adjust);

// Cycle through the primes, clearing bits
// Have already done 3, 5, and 7
PrimeInit(7, &iter, primes);

// Get the next N primes where N is determined by the mark in the sieveMarks
while (composite = NextPrime(&iter))
    {
        UINT16 pList[8];
        UINT16 next = 0;
        i = count;
pList[i++] = composite;
for (; i > 0; i--)
        {
            next = NextPrime(&iter);
pList[i] = next;
            if (next != 0)
                composite *= next;
        }

        composite = BN_mod_word(bnN, composite);
        for (i = count; i > 0; i--)
            {
                next = pList[i];
                if (next == 0)
Part 4: Supporting Routines

Trusted Platform Module Library

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.

```c
BOOL PrimeSelectWithSieve(
    BIGNUM *bnP,          // IN/OUT: The candidate to filter
    KDFa_CONTEXT *ktx,    // IN: KDFa iterator structure
    UINT32 e,             // IN: the exponent
    BN_CTX *context       // IN: the big number context to play in
    #ifdef RSA_DEBUG       //%
        UINT16 fieldSize,  // IN: number of bytes in the field, as
        UINT16 primes,     // IN: number of primes to use.
    #endif               //%
) {
    BYTE field[MAX_FIELD_SIZE];
    UINT32 first;
    UINT32 ones;
    UINT32 chosen;
    UINT16 rounds = MillerRabinRounds(BN_num_bits(bnP));
    #ifdef RSA_DEBUG
    UINT16 primes;
    UINT32 fieldSize;
    // Adjust the field size and prime table list to fit the size of the prime
    // being tested.
    primes = BN_num_bits(bnP);
    if(primes <= 512) {
        primes = AdjustNumberOfPrimes(2048);
        fieldSize = 65;
    } else if(primes <= 1024) {
        primes = AdjustNumberOfPrimes(4096);
```
fieldSize = 129;
}
else
{
    primes = AdjustNumberOfPrimes(0);  // Set to the maximum
    fieldSize = MAX_FIELD_SIZE;
}
if(fieldSize > MAX_FIELD_SIZE)
    fieldSize = MAX_FIELD_SIZE;
#endif

// Save the low-order word to use as a search generator and make sure that
// it has some interesting range to it
first = bnP->d[0] | 0x80000000;

// Align to field boundary
bnP->d[0] &= ~((UINT32)(fieldSize-3));
pAssert(BN_is_bit_set(bnP, 0));
bnP->d[0] &= ((UINT32_MAX << (FIELD_POWER + 1)) + 1);
ones = PrimeSieve(bnP, fieldSize, field, primes);
#endif
pAssert(ones == BitsInArray(field, defaultFieldSize));
#endif
for(; ones > 0; ones--)
{
    if(ones != BitsInArray(field, defaultFieldSize))
        FAIL(FATAL_ERROR_INTERNAL);
    // Decide which bit to look at and find its offset
    if(ones == 1)
    {
        ones = ones;
        chosen = FindNthSetBit(defaultFieldSize, field,((first % ones) + 1));
        if(chosen >= ((defaultFieldSize) * 8))
            FAIL(FATAL_ERROR_INTERNAL);
    }
    // Set this as the trial prime
    BN_add_word(bnP, chosen * 2);
    // Use MR to see if this is prime
    if(MillerRabin(bnP, rounds, ktx, context))
    {
        // Final check is to make sure that 0 != (p-1) mod e
        // This is the same as -1 != p mod e ; or
        // (e - 1) != p mod e
        if((e <= 3) || (BN_mod_word(bnP, e) != (e-1)))
            return TRUE;
    }
    // Back out the bit number
    BN_sub_word(bnP, chosen * 2);
    // Clear the bit just tested
    ClearBit(field, chosen);
}
// Ran out of bits and couldn't find a prime in this field
INSTRUMENT_INC(noPrimeFields);
return FALSE;

B.9.2.3.2.12. AdjustPrimeCandidate()

This function adjusts the candidate prime so that it is odd and > root(2)/2. This allows the product of these two numbers to be .5, which, in fixed point notation means that the most significant bit is 1. For this
routine, the root(2)/2 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.

```c
void AdjustPrimeCandidate(
    BYTE *a,
    UINT16 len
) {
    UINT16 highBytes;
    highBytes = BYTE_ARRAY_TO_UINT16(a);
    // This is fixed point arithmetic on 16-bit values
    highBytes = ((UINT32)highBytes * (UINT32)0x4AFB) >> 16;
    highBytes += 0xB505;
    UINT16_TO_BYTE_ARRAY(highBytes, a);
    a[len-1] |= 1;
}
```

B.9.2.3.2.13. GenerateRandomPrime()
KDFa_CONTEXT *
KDFaContextStart(
    KDFa_CONTEXT *ktx,        // IN/OUT: the context structure to initialize
    TPM2B *seed,              // IN: the seed for the digest process
    TPM_ALG_ID hashAlg,       // IN: the hash algorithm
    TPM2B *extra,             // IN: the extra data
    UINT32 *outer,            // IN: the outer iteration counter
    UINT16 keySizeInBits
)
{
    UINT16 digestSize = _cpri__GetDigestSize(hashAlg);
    TPM2B_HASH_BLOCK oPadKey;
    if(seed == NULL)
        return NULL;

    pAssert(ktx != NULL && outer != NULL && digestSize != 0);

    // Start the hash using the seed and get the intermediate hash value
    _cpri__StartHMAC(hashAlg, FALSE, &(ktx->iPadCtx), seed->size, seed->buffer,
                     &oPadKey.b);
    _cpri__StartHash(hashAlg, FALSE, &(ktx->oPadCtx));
    _cpri__UpdateHash(&(ktx->oPadCtx), oPadKey.b.size, oPadKey.b.buffer);
    ktx->extra = extra;
    ktx->hashAlg = hashAlg;
    ktx->outer = outer;
    ktx->keySizeInBits = keySizeInBits;
    return ktx;
}

void KDFaContextEnd(
    KDFa_CONTEXT *ktx       // IN/OUT: the context structure to close
)
{
    if(ktx != NULL)
    {
        // Close out the hash sessions
        _cpri__CompleteHash(&(ktx->iPadCtx), 0, NULL);
        _cpri__CompleteHash(&(ktx->oPadCtx), 0, NULL);
    }
}

//%#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 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
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_FAIL</td>
<td>exponent is not prime or is less than 3; or could not find a prime using the provided parameters</td>
</tr>
<tr>
<td>CRYPT_CANCEL</td>
<td>operation was cancelled</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT

_comri_generateKeyRSA(
    TPM2B *n,       // OUT: The public modulus
    TPM2B *p,       // OUT: One of the prime factors of n
    UINT16 *keySizeInBits, // IN: Size of the public modulus in bits
    UINT32 *e,       // IN: The public exponent
    TPM_ALG_ID hashAlg, // IN: hash algorithm to use in the key generation process
    TPM2B *seed,     // IN: the seed to use
    const char *label, // IN: A label for the generation process.
    TPM2B *extra,    // IN: Party 1 data for the KDF
    UINT32 *counter  // IN/OUT: Counter value to allow KDF iteration to be propagated across multiple routines
    #ifdef RSA_DEBUG //%
    ,UINT16 primes, // IN: number of primes to test
    UINT16 fieldSize // IN: the field size to use
    #endif //%
    )
{
    CRYPT_RESULT retVal;
    UINT32 myCounter = 0;
    UINT32 *pCtr = (counter == NULL) ? &myCounter : counter;

    // Make sure that the required pointers are provided
    pAssert(n != NULL && p != NULL);

    // If the seed is provided, then use KDFa for generation of the 'random' values
    ktxPtr = KDFaContextStart(&ktx, seed, hashAlg, extra, pCtr, keySizeInBits);

    n->size = keySizeInBits/8;
    p->size = n->size / 2;

    // Validate exponent
    if((e == 0 || e == RSA_DEFAULT_PUBLIC_EXPONENT)
        e = RSA_DEFAULT_PUBLIC_EXPONENT;)
    else
        if(!IsPrimeWord(e))
            return CRYPT_FAIL;

    // Get structures for the big number representations
    context = BN_CTX_new();
    BN_CTX_start(context);
    bnP = BN_CTX_get(context);
    bnQ = BN_CTX_get(context);
    bnT = BN_CTX_get(context);
```
bnE = BN_CTX_get(context);
bnN = BN_CTX_get(context);
if(bnN == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

// Set Q to zero. This is used as a flag. The prime is computed in P. When a
// new prime is found, Q is checked to see if it is zero. If so, P is copied
// to Q and a new P is found. When both P and Q are non-zero, the modulus and
// private exponent are computed and a trial encryption/decryption is
// performed. If the encrypt/decrypt fails, assume that at least one of the
// primes is composite. Since we don't know which one, set Q to zero and start
// over and find a new pair of primes.
BN_zero(bnQ);
BN_set_word(bnE, e);

// Each call to generate a random value will increment ktx.outer
// it doesn't matter if ktx.outer wraps. This lets the caller
// use the initial value of the counter for additional entropy.
for(i = 0; i < UINT32_MAX; i++)
{
    if(_plat__IsCanceled())
    {
        retVal = CRYPT_CANCEL;
        goto end;
    }
    // Get a random prime candidate.
    if(seed == NULL)
        _cpri__GenerateRandom(p->size, p->buffer);
    else
        RandomForRsa(&ktx, label, p);
    AdjustPrimeCandidate(p->buffer, p->size);

    // Convert the candidate to a BN
    if(BN_bin2bn(p->buffer, p->size, bnP) == NULL)
        FAIL(FATAL_ERROR_INTERNAL);
    // If this is the second prime, make sure that it differs from the
    // first prime by at least 2^100. Since BIGNUMS use words, the check
    // below will make sure they are different by at least 128 bits
    if(!BN_is_zero(bnQ))
    {
        // bnQ is non-zero, we have a first value
        UINT32 *pP = (UINT32 *)(bnP->d[4]);
        UINT32 *pQ = (UINT32 *)(bnQ->d[4]);
        int k = (((INT32)bnP->top) - 4;
        for(; k > 0; k--)
            if(*pP++ != *pQ++)
                break;
        // Didn't find any difference so go get a new value
        if(k == 0)
            continue;
    }
    // If PrimeSelectWithSieve returns success, bnP is a prime,
#ifdef RSA_DEBUG
    if(!PrimeSelectWithSieve(bnP, ktxPtr, e, context, fieldSize, primes))
#else
    if(!PrimeSelectWithSieve(bnP, ktxPtr, e, context))
#endif
        continue;  // If not, get another

    // Found a prime, is this the first or second.
    if(!BN_is_zero(bnQ))
    {
        // copy p to q and compute another prime in p
        BN_copy(bnQ, bnP);
        continue;
    }
    //Form the public modulus
    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         || BN_mod_exp(bnQ, bnP, bnE, bnN, context) != 1
843         || BN_mod_exp(bnQ, bnQ, bnT, bnN, context) != 1)
844             FAIL(FATAL_ERROR_INTERNAL);
845     // If the starting and ending values are not the same, start over )-
846     if (BN_ucmp(bnP, bnQ) != 0)
847         {
848             BN_zero(bnQ);
849             continue;
850         }
851     }
852 #endif // EXTENDED_CHECKS
853     retVal = CRYPT_SUCCESS;
854     goto end;
855 }
856     retVal = CRYPT_FAIL;
857 end:
858 KDFaContextEnd(&ktx);
859 // Free up allocated BN values
860 BN_CTX_end(context);
861 BN_CTX_free(context);
862 return retVal;
867  }
868  #else
869  static void noFunction(void)
870  {
871      pAssert(1);
872  }
873  #endif  //%

B.9.2.4. RSAData.c

#include "CryptoEngine.h"
#ifdef  RSA_KEY_SIEVE
#ifdef  RSA_DEBUG
 UINT16  defaultFieldSize = MAX_FIELD_SIZE;
#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.

const BYTE   seedValues[SEED_VALUES_SIZE] = {
0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c,
0x99, 0x96, 0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
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.
0x00,0x0E,0x04,0x1E,0x06,0x06,0x06,0x08,0x06,0x04,0x02,0x0C,0x06,0x04,0x02,0x06,
0x16,0x06,0x02,0x04,0x12,0x02,0x04,0x0C,0x02,0x06,0x04,0x1A,0x06,0x06,0x04,0x08,
0x0A,0x20,0x10,0x02,0x06,0x04,0x02,0x04,0x02,0x0A,0x0E,0x06,0x04,0x08,0x0A,0x06,
0x14,0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
0x04,0x02,0x06,0x01E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x04,0x08,0x06,0x06,
Trusted Platform Module Library

Part 4: Supporting Routines

PRIME_DIFF_TABLE_BYTES > 3072

PRIME_DIFF_TABLE_BYTES > 2816

PRIME_DIFF_TABLE_BYTES > 2560

Copyright © TCG 2006
Published March 15, 2013

Family "02"
Part 4: Supporting Routines

```c
#define PRIME_DIFF_TABLE_BYTES > 3840

if (PRIME_DIFF_TABLE_BYTES > 3840) {
    // 3840
    #if PRIME_DIFF_TABLE_BYTES > 3584
        // 3584
        #if PRIME_DIFF_TABLE_BYTES > 3840
            // 3840
            #if PRIME_DIFF_TABLE_BYTES > 4096
                // 4096
                #if PRIME_DIFF_TABLE_BYTES > 4096
                    // 4096
                #endif
            #endif
        #endif
    #endif
}

// 3840
```

```
0x0C, 0x0C, 0x06, 0x0A, 0x02, 0x0A, 0x01E, 0x02, 0x10, 0x08, 0x04, 0x02, 0x06, 0x12, 0x04, 0x02,
0x06, 0x04, 0x01A, 0x04, 0x08, 0x06, 0x0A, 0x02, 0x04, 0x06, 0x08, 0x04, 0x06, 0x06, 0x0E, 0x0C, 0x02,
0x06, 0x06, 0x04, 0x14, 0x16, 0x08, 0x04, 0x02, 0x04, 0x04, 0x48, 0x08, 0x04, 0x06, 0x16, 0x02, 0x04,
0x0E, 0x0A, 0x02, 0x04, 0x14, 0x06, 0x0A, 0x12, 0x06, 0x14, 0x10, 0x06, 0x08, 0x06, 0x04, 0x14,
0x0C, 0x16, 0x02, 0x04, 0x02, 0x0C, 0x0A, 0x12, 0x02, 0x16, 0x06, 0x12, 0x1E, 0x02, 0x02, 0x0A, 0x0E,
0x0A, 0x08, 0x10, 0x32, 0x06, 0x06, 0x0A, 0x08, 0x0A, 0x0C, 0x06, 0x12, 0x02, 0x16, 0x06, 0x02, 0x04,
0x06, 0x08, 0x06, 0x06, 0x0A, 0x12, 0x02, 0x16, 0x02, 0x10, 0x0E, 0x0A, 0x06, 0x02, 0x0C, 0x0A,
0x14, 0x04, 0x0E, 0x04, 0x06, 0x24, 0x02, 0x04, 0x06, 0x0C, 0x02, 0x04, 0x04, 0x0E, 0x0C, 0x06, 0x04,
0x06, 0x02, 0x06, 0x04, 0x14, 0x0A, 0x02, 0x06, 0x0C, 0x02, 0x18, 0x0C, 0x0C, 0x06, 0x06,
0x04, 0x18, 0x10, 0x02, 0x04, 0x18, 0x02, 0x06, 0x04, 0x06, 0x08, 0x10, 0x06, 0x02, 0x0A, 0x0C, 0x0E,
0x06, 0x22, 0x06, 0x0E, 0x0E, 0x02, 0x04, 0x02, 0x1E, 0x16, 0x08, 0x04, 0x06, 0x08, 0x04, 0x02, 0x1C,
0x02, 0x06, 0x04, 0x1A, 0x12, 0x16, 0x02, 0x06, 0x10, 0x06, 0x02, 0x10, 0x0C, 0x02, 0x04,
0x06, 0x06, 0x06, 0x0A, 0x06, 0x08, 0x0C, 0x04, 0x12, 0x02, 0x0A, 0x08, 0x10, 0x06, 0x06, 0x1E,
0x02, 0x0A, 0x12, 0x02, 0x0A, 0x08, 0x04, 0x08, 0x0C, 0x18, 0x28, 0x02, 0x0C, 0x0A, 0x06, 0x0C,
0x02, 0x02, 0x0C, 0x04, 0x02, 0x06, 0x10, 0x0E, 0x0C, 0x06, 0x04, 0x0E, 0x01E, 0x04, 0x08, 0x0A
```
Trusted Platform Module Library
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437

Part 4: Supporting Routines

0x0C,0x12,0x0A,0x02,0x36,0x04,0x02,0x0A,0x1E,0x0C,0x08,0x04,0x08,0x10,0x0E,0x0C,
0x06,0x04,0x06,0x0C,0x06,0x02,0x04,0x0E,0x0C,0x04,0x0E,0x06,0x18,0x06,0x06,0x0A,
0x0C,0x0C,0x14,0x12,0x06,0x06,0x10,0x08,0x04,0x06,0x14,0x04,0x20,0x04,0x0E,0x0A,
0x02,0x06,0x0C,0x10,0x02,0x04,0x06,0x0C,0x02,0x0A,0x08,0x06,0x04,0x02,0x0A,0x0E,
0x06,0x06,0x0C,0x12,0x22,0x08,0x0A,0x06,0x18,0x06,0x02,0x0A,0x0C,0x02,0x1E,0x0A,
0x0E,0x0C,0x0C,0x10,0x06,0x06,0x02,0x12,0x04,0x06,0x1E,0x0E,0x04,0x06,0x06,0x02
#endif
// 4352
#if PRIME_DIFF_TABLE_BYTES > 4352
,0x06,0x04,0x06,0x0E,0x06,0x04,0x08,0x0A,0x0C,0x06,0x20,0x0A,0x08,0x16,0x02,0x0A,
0x06,0x18,0x08,0x04,0x1E,0x06,0x02,0x0C,0x10,0x08,0x06,0x04,0x06,0x08,0x10,0x0E,
0x06,0x06,0x04,0x02,0x0A,0x0C,0x02,0x10,0x0E,0x04,0x02,0x04,0x14,0x12,0x0A,0x02,
0x0A,0x06,0x0C,0x1E,0x08,0x12,0x0C,0x0A,0x02,0x06,0x06,0x04,0x0C,0x0C,0x02,0x04,
0x0C,0x12,0x18,0x02,0x0A,0x06,0x08,0x10,0x08,0x06,0x0C,0x0A,0x0E,0x06,0x0C,0x06,
0x06,0x04,0x02,0x18,0x04,0x06,0x08,0x06,0x04,0x02,0x04,0x06,0x0E,0x04,0x08,0x0A,
0x18,0x18,0x0C,0x02,0x06,0x0C,0x16,0x1E,0x02,0x06,0x12,0x0A,0x06,0x06,0x08,0x04,
0x02,0x06,0x0A,0x08,0x0A,0x06,0x08,0x10,0x06,0x0E,0x06,0x04,0x18,0x08,0x0A,0x02,
0x0C,0x06,0x04,0x24,0x02,0x16,0x06,0x08,0x06,0x0A,0x08,0x06,0x0C,0x0A,0x0E,0x0A,
0x06,0x12,0x0C,0x02,0x0C,0x04,0x1A,0x0A,0x0E,0x10,0x12,0x08,0x12,0x0C,0x0C,0x06,
0x10,0x0E,0x18,0x0A,0x0C,0x08,0x16,0x06,0x02,0x0A,0x3C,0x06,0x02,0x04,0x08,0x10,
0x0E,0x0A,0x06,0x18,0x06,0x0C,0x12,0x18,0x02,0x1E,0x04,0x02,0x0C,0x06,0x0A,0x02,
0x04,0x0E,0x06,0x10,0x02,0x0A,0x08,0x16,0x14,0x06,0x04,0x20,0x06,0x12,0x04,0x02,
0x04,0x02,0x04,0x08,0x34,0x0E,0x16,0x02,0x16,0x14,0x0A,0x08,0x0A,0x02,0x06,0x04,
0x0E,0x04,0x06,0x14,0x04,0x06,0x02,0x0C,0x0C,0x06,0x0C,0x10,0x02,0x0C,0x0A,0x08,
0x04,0x06,0x02,0x1C,0x0C,0x08,0x0A,0x0C,0x02,0x04,0x0E,0x1C,0x08,0x06,0x04,0x02
#endif
// 4608
#if PRIME_DIFF_TABLE_BYTES > 4608
,0x04,0x06,0x02,0x0C,0x3A,0x06,0x0E,0x0A,0x02,0x06,0x1C,0x20,0x04,0x1E,0x08,0x06,
0x04,0x06,0x0C,0x0C,0x02,0x04,0x06,0x06,0x0E,0x10,0x08,0x1E,0x04,0x02,0x0A,0x08,
0x06,0x04,0x06,0x1A,0x04,0x0C,0x02,0x0A,0x12,0x0C,0x0C,0x12,0x02,0x04,0x0C,0x08,
0x0C,0x0A,0x14,0x04,0x08,0x10,0x0C,0x08,0x06,0x10,0x08,0x0A,0x0C,0x0E,0x06,0x04,
0x08,0x0C,0x04,0x14,0x06,0x28,0x08,0x10,0x06,0x24,0x02,0x06,0x04,0x06,0x02,0x16,
0x12,0x02,0x0A,0x06,0x24,0x0E,0x0C,0x04,0x12,0x08,0x04,0x0E,0x0A,0x02,0x0A,0x08,
0x04,0x02,0x12,0x10,0x0C,0x0E,0x0A,0x0E,0x06,0x06,0x2A,0x0A,0x06,0x06,0x14,0x0A,
0x08,0x0C,0x04,0x0C,0x12,0x02,0x0A,0x0E,0x12,0x0A,0x12,0x08,0x06,0x04,0x0E,0x06,
0x0A,0x1E,0x0E,0x06,0x06,0x04,0x0C,0x26,0x04,0x02,0x04,0x06,0x08,0x0C,0x0A,0x06,
0x12,0x06,0x32,0x06,0x04,0x06,0x0C,0x08,0x0A,0x20,0x06,0x16,0x02,0x0A,0x0C,0x12,
0x02,0x06,0x04,0x1E,0x08,0x06,0x06,0x12,0x0A,0x02,0x04,0x0C,0x14,0x0A,0x08,0x18,
0x0A,0x02,0x06,0x16,0x06,0x02,0x12,0x0A,0x0C,0x02,0x1E,0x12,0x0C,0x1C,0x02,0x06,
0x04,0x06,0x0E,0x06,0x0C,0x0A,0x08,0x04,0x0C,0x1A,0x0A,0x08,0x06,0x10,0x02,0x0A,
0x12,0x0E,0x06,0x04,0x06,0x0E,0x10,0x02,0x06,0x04,0x0C,0x14,0x04,0x14,0x04,0x06,
0x0C,0x02,0x24,0x04,0x06,0x02,0x0A,0x02,0x16,0x08,0x06,0x0A,0x0C,0x0C,0x12,0x0E,
0x18,0x24,0x04,0x14,0x18,0x0A,0x06,0x02,0x1C,0x06,0x12,0x08,0x04,0x06,0x08,0x06
#endif
// 4864
#if PRIME_DIFF_TABLE_BYTES > 4864
,0x04,0x02,0x0C,0x1C,0x12,0x0E,0x10,0x0E,0x12,0x0A,0x08,0x06,0x04,0x06,0x06,0x08,
0x16,0x0C,0x02,0x0A,0x12,0x06,0x02,0x12,0x0A,0x02,0x0C,0x0A,0x12,0x20,0x06,0x04,
0x06,0x06,0x08,0x06,0x06,0x0A,0x14,0x06,0x0C,0x0A,0x08,0x0A,0x0E,0x06,0x0A,0x0E,
0x04,0x02,0x16,0x12,0x02,0x0A,0x02,0x04,0x14,0x04,0x02,0x22,0x02,0x0C,0x06,0x0A,
0x02,0x0A,0x12,0x06,0x0E,0x0C,0x0C,0x16,0x08,0x06,0x10,0x06,0x08,0x04,0x0C,0x06,
0x08,0x04,0x24,0x06,0x06,0x14,0x18,0x06,0x0C,0x12,0x0A,0x02,0x0A,0x1A,0x06,0x10,
0x08,0x06,0x04,0x18,0x12,0x08,0x0C,0x0C,0x0A,0x12,0x0C,0x02,0x18,0x04,0x0C,0x12,
0x0C,0x0E,0x0A,0x02,0x04,0x18,0x0C,0x0E,0x0A,0x06,0x02,0x06,0x04,0x06,0x1A,0x04,
0x06,0x06,0x02,0x16,0x08,0x12,0x04,0x12,0x08,0x04,0x18,0x02,0x0C,0x0C,0x04,0x02,
0x34,0x02,0x12,0x06,0x04,0x06,0x0C,0x02,0x06,0x0C,0x0A,0x08,0x04,0x02,0x18,0x0A,
0x02,0x0A,0x02,0x0C,0x06,0x12,0x28,0x06,0x14,0x10,0x02,0x0C,0x06,0x0A,0x0C,0x02,
0x04,0x06,0x0E,0x0C,0x0C,0x16,0x06,0x08,0x04,0x02,0x10,0x12,0x0C,0x02,0x06,0x10,
0x06,0x02,0x06,0x04,0x0C,0x1E,0x08,0x10,0x02,0x12,0x0A,0x18,0x02,0x06,0x18,0x04,
0x02,0x16,0x02,0x10,0x02,0x06,0x0C,0x04,0x12,0x08,0x04,0x0E,0x04,0x12,0x18,0x06,
0x02,0x06,0x0A,0x02,0x0A,0x26,0x06,0x0A,0x0E,0x06,0x06,0x18,0x04,0x02,0x0C,0x10,
0x0E,0x10,0x0C,0x02,0x06,0x0A,0x1A,0x04,0x02,0x0C,0x06,0x04,0x0C,0x08,0x0C,0x0A
#endif
// 5120
#if PRIME_DIFF_TABLE_BYTES > 5120

Family "02"
Level 00 Revision 00.96

Published
Copyright © TCG 2006-2013

Page 425
March 15, 2013


Part 4: Supporting Routines

Page 426

Published
March 15, 2013

Family "02"

Copyright © TCG 2006-2013

Level 00 Revision 00.96
#ifndef

// 6144
#endif

#ifdef

// 6144
#endif

if (PRIME_DIFF_TABLE_BYTES > 6144)

#endif

// 6400

if (PRIME_DIFF_TABLE_BYTES > 6400)

#endif

// 6542

ifdef

#endif

#ifdef

#endif

#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

#define RSA_DEBUG

const __int16 primes[NUM_PRIMES] =

{ 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
  59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
  137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
  227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311,
  313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
  419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,
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

```c
#ifndef __CRYPTDATAECC_H__
#define __CRYPTDATAECC_H__

Structure for the curve parameters. This is an analog to the TPMS_ALGORITHM_DETAIL_ECC

```typedef struct {
    const TPM2B *p; // a prime number
    const TPM2B *a; // linear coefficient
    const TPM2B *b; // constant term
    const TPM2B *x; // generator x coordinate
    const TPM2B *y; // generator y coordinate
    const TPM2B *n; // the order of the curve
    const TPM2B *h; // cofactor
} ECC_CURVE_DATA;

typedef struct {
    TPM_ECC_CURVE curveId;
    UINT16 keySizeBits;
    TPMT_KDF_SCHEME kdf;
    TPMT_ECC_SCHEME sign;
    const ECC_CURVE_DATA *curveData; // the address of the curve data
} ECC_CURVE;
```

```c
extern const ECC_CURVE_DATA SM2_P256;
extern const ECC_CURVE_DATA NIST_P256;
extern const ECC_CURVE_DATA BN_P256;
extern const ECC_CURVE eccCurves[];
extern const UINT16 ECC_CURVE_COUNT;
#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 CryptUtil.c

NOTE: This file should be included by the Ecc module in the library.

B.10.2.1.2. NIST Prime 256-bit Curve

```c
static const TPM2B_32_BYTE_VALUE NIST_P256_P = {32, {
    0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x01,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
    0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}
};
static const TPM2B_32_BYTE_VALUE NIST_P256_A = {32, {
    0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x01,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
    0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}
};
static const TPM2B_32_BYTE_VALUE NIST_P256_B = {32, {
    0x5a,0xc6,0x35,0xd8,0xaa,0x3a,0x93,0xe7,
    0xb3,0xeb,0xbd,0x55,0x76,0x98,0x86,0x7c,
    0x65,0x1d,0x06,0xb0,0xcc,0x53,0xb0,0xf6,
    0x3b,0xce,0x3c,0x3e,0x27,0xd2,0x60,0x4b}
};
```
static const TPM2B_32_BYTE_VALUE NIST_P256_X = {32,  
0x6b,0x17,0xd1,0xf2,0xe1,0x2c,0x42,0x47,  
0x77,0x03,0x7d,0x81,0x2d,0xeb,0x33,0xa0,  
0x4f,0x4a,0x39,0x45,0xd8,0x98,0xc2,0x96};

static const TPM2B_32_BYTE_VALUE NIST_P256_Y = {32,  
0x4f,0xe3,0x42,0xe2,0xfe,0x1a,0x7f,0x9b,  
0x8e,0xe7,0xeb,0x4a,0x7c,0x0f,0x9e,0x16,  
0x2b,0xce,0x33,0x57,0x6b,0x31,0x5e,0xc2,  
0xf3,0xb9,0x68,0x37,0xbf,0x51};

static const TPM2B_32_BYTE_VALUE NIST_P256_N = {32,  
0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,  
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,  
0xbc,0xe6,0xfa,0xad,0xa7,0x17,0x84,0x98,  
0xf3,0xb9,0x68,0x37,0xbf,0x51};

static const TPM2B_1_BYTE_VALUE NIST_P256_H = {1, {1}};

const ECC_CURVE_DATA NIST_P256 = {
&NIST_P256_P.b, &NIST_P256_A.b, &NIST_P256_B.b,  
&NIST_P256_X.b, &NIST_P256_Y.b, &NIST_P256_N.b,  
&NIST_P256_H.b};

B.10.2.1.3. BN Prime 256-bit Curve

static const TPM2B_32_BYTE_VALUE BN_P256_P = {32,  
0xff,0xff,0xff,0xf0,0x0f,0x0f,0x0f,0xcd,  
0x46,0x5e,0x71,0x90,0x9f,0x71,0x0a,0x82,  
0x0c,0x65,0x12,0x98,0x0a,0x82,0x0c,0x65,  
0x0d,0x13,0x30,0x13};

static const TPM2B_1_BYTE_VALUE BN_P256_A = {1, {0}};

static const TPM2B_1_BYTE_VALUE BN_P256_B = {1, {3}};

static const TPM2B_1_BYTE_VALUE BN_P256_X = {1, {1}};

static const TPM2B_1_BYTE_VALUE BN_P256_Y = {1, {2}};

static const TPM2B_32_BYTE_VALUE BN_P256_N = {32,  
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,  
0x48,0x93,0x4a,0x92,0xa7,0x63,0x6c,0x63,  
0x6a,0x6b,0x6c,0x6d,0x6e,0x6f,0x70,0x71,  
0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79};

static const TPM2B_1_BYTE_VALUE BN_P256_H = {1, {1}};

const ECC_CURVE_DATA BN_P256 = {
&BN_P256_P.b, &BN_P256_A.b, &BN_P256_B.b,  
&BN_P256_X.b, &BN_P256_Y.b, &BN_P256_N.b,  
&BN_P256_H.b};

#ifdef TPM_ECC_SM2_P256
#ifndef _SM2_SIGN_DEBUG
These are the actual values for SM2 curve

static const TPM2B_32_BYTE_VALUE SM2_P256_P = {32,  
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,  
0x46,0x5e,0x71,0x90,0x9f,0x71,0x0a,0x82,  
0x0c,0x65,0x12,0x98,0x0a,0x82,0x0c,0x65,  
0x0d,0x13,0x30,0x13};

static const TPM2B_32_BYTE_VALUE SM2_P256_A = {32,  
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,  
0x46,0x5e,0x71,0x90,0x9f,0x71,0x0a,0x82,  
0x0c,0x65,0x12,0x98,0x0a,0x82,0x0c,0x65,  
0x0d,0x13,0x30,0x13};

static const TPM2B_32_BYTE_VALUE SM2_P256_B = {32,  
0x28,0x9e,0x9e,0x9e,0x9e,0x9e,0x9e,0x9e,  
0x4d,0x5a,0x9e,0x4b,0xc9,0x65,0x99,0xa7,  
0xf3,0x97,0x89,0x5f,0x15,0x9b,0x8f,0x92,  
0x0d,0x0b,0x0d,0x41,0x4d,0x94,0x0e,0x93};

static const TPM2B_32_BYTE_VALUE SM2_P256_X = {32,  
0x32,0xc4,0x0a,0x2c,0x1f,0x19,0x81,0x19,  
0x5f,0x99,0x04,0x46,0x6a,0x39,0xc9,0x94,  
0x8f,0xe3,0x0b,0x8f,0xe2,0x66,0x0b,0xe1,  
0x41,0x4d,0x94,0x0e,0x93};

#endif
#endif

These are the actual values for SM2 curve

static const TPM2B_32_BYTE_VALUE SM2_P256_P = {32,  
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,  
0x46,0x5e,0x71,0x90,0x9f,0x71,0x0a,0x82,  
0x0c,0x65,0x12,0x98,0x0a,0x82,0x0c,0x65,  
0x0d,0x13,0x30,0x13};

static const TPM2B_32_BYTE_VALUE SM2_P256_A = {32,  
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,  
0x46,0x5e,0x71,0x90,0x9f,0x71,0x0a,0x82,  
0x0c,0x65,0x12,0x98,0x0a,0x82,0x0c,0x65,  
0x0d,0x13,0x30,0x13};

static const TPM2B_32_BYTE_VALUE SM2_P256_B = {32,  
0x28,0x9e,0x9e,0x9e,0x9e,0x9e,0x9e,0x9e,  
0x4d,0x5a,0x9e,0x4b,0xc9,0x65,0x99,0xa7,  
0xf3,0x97,0x89,0x5f,0x15,0x9b,0x8f,0x92,  
0x0d,0x0b,0x0d,0x41,0x4d,0x94,0x0e,0x93};

static const TPM2B_32_BYTE_VALUE SM2_P256_X = {32,  
0x32,0xc4,0x0a,0x2c,0x1f,0x19,0x81,0x19,  
0x5f,0x99,0x04,0x46,0x6a,0x39,0xc9,0x94,  
0x8f,0xe3,0x0b,0x8f,0xe2,0x66,0x0b,0xe1,  
0x41,0x4d,0x94,0x0e,0x93};
74  static const TPM2B_32_BYTE_VALUE SM2_P256_Y = {32,
75      0x71,0x5A,0x45,0x89,0x33,0x4C,0x74,0xC7};
76  static const TPM2B_32_BYTE_VALUE SM2_P256_N = {32,
77      0xFC,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
78      0x53,0xBB,0xF4,0x09,0x39,0xD5,0x41,0x23};
79  #endif
80  #else //_SM2_SIGN_DEBUG
81   These are the values for debug of SM2 sign
82  static const TPM2B_32_BYTE_VALUE SM2_P256_P = {32,
83      0x85,0x42,0xD6,0x9E,0x4C,0x04,0x4F,0x18,
84      0x29,0x77,0x20,0x63,0x04,0x85,0x62,0x8D,
85      0x5A,0xE7,0x4E,0xE7,0xC3,0x2E,0x79,0xB7};
86  static const TPM2B_32_BYTE_VALUE SM2_P256_A = {32,
87      0x78,0x79,0x68,0xB4,0xFA,0x32,0xC3,0xFD,
88      0x2F,0x3C,0x84,0x8B,0x68,0x31,0xD7,0xE0,
89      0x85,0x42,0xD6,0x9E,0x4C,0x04,0x4F,0x18,
90      0x29,0x77,0x20,0x63,0x04,0x85,0x62,0x8D,
91      0x5A,0xE7,0x4E,0xE7,0xC3,0x2E,0x79,0xB7};
92  static const TPM2B_32_BYTE_VALUE SM2_P256_X = {32,
93      0x42,0x1D,0xEB,0xD6,0x1B,0x62,0xEA,0xB6,
94      0x85,0x42,0xD6,0x9E,0x4C,0x04,0x4F,0x18,
95      0x29,0x77,0x20,0x63,0x04,0x85,0x62,0x8D,
96      0x5A,0xE7,0x4E,0xE7,0xC3,0x2E,0x79,0xB7};
97  static const TPM2B_32_BYTE_VALUE SM2_P256_Y = {32,
98      0x06,0x80,0x51,0x2B,0xCB,0xB4,0x2C,0x07,
99      0x29,0x77,0x20,0x63,0x04,0x85,0x62,0x8D,
100     0x85,0x42,0xD6,0x9E,0x4C,0x04,0x4F,0x18,
101     0x29,0x77,0x20,0x63,0x04,0x85,0x62,0x8D,
102     0x5A,0xE7,0x4E,0xE7,0xC3,0x2E,0x79,0xB7};
103  #endif
104  static const TPM2B_1_BYTE_VALUE   SM2_P256_H = {1, {1}};
105  #endif
106  #ifdef
107     #ifndef
108     const ECC_CURVE_DATA SM2_P256 = {&SM2_P256_P.b, &SM2_P256_A.b, &SM2_P256_B.b,
109         &SM2_P256_X.b, &SM2_P256_Y.b, &SM2_P256_N.b,
110         &SM2_P256_H.b};
111  #endif
112  #endif
113
114  Make sure that this table has algorithms in the same order as the eccCurveValues[] table in CryptUtil.c
115
116  const ECC_CURVE   eccCurves[] =
117  {
118    {TPM_ECC_NIST_P256,  // curveId
119      256,              // key size in bits
120    {TPM_ALG_NULL, {TPM_ALG_NULL}}, // default KDF and hash
121    {TPM_ALG_NULL, {TPM_ALG_NULL}}, // default signing scheme and hash
122    &NIST_P256}          // curve values
123  }
{TPM_ALG_NULL, TPM_ALG_NULL},
&SM2_P256)
#endif
#ifdef TPM_ALG_ECDAA
, {TPM_ECC_BN_P256,
256,
{TPM_ALG_NULL, TPM_ALG_NULL},
{TPM_ALG_ECDAA, TPM_ALG_NULL},
&BN_P256}
#endif
};
const UINT16 ECC_CURVE_COUNT = sizeof(eccCurves) / sizeof(ECC_CURVE);

B.10.3. CpriECC.c

1 /* (Copyright)
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 //** Includes and Defines
9 #include "CryptoEngine.h"
10
11 //** Functions
12
13 //*** _cpri__EccStartup()
14 // This function is called at TPM Startup to initialize the crypto units.
15 // In this implementation, no initialization is performed at startup but a
16 // future version may initialize the self-test functions here.
17 BOOL _cpri__EccStartup(
18 void
19 )
20 {
21 return TRUE;
22 }

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.

TPM_ECC_CURVE
_cpri__GetCurveIdByIndex(
UINT16 i
)
{
if(i >= ECC_CURVE_COUNT)
    return TPM_ECC_NONE;
return eccCurves[i].curveId;
}

UINT32
_cpri__EccGetCurveCount(
void
)
{
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>curve with the indicated TPM_ECC_CURVE value is not implemented</td>
</tr>
<tr>
<td>-&gt;</td>
<td>pointer to the curve data</td>
</tr>
</tbody>
</table>

```
43 const ECC_CURVE *
44 _cpri__EccGetParametersByCurveId(
45 TPM_ECC_CURVE       curveId     // IN: the curveID
46 )
47 { int i;
48   for(i = 0; i < ECC_CURVE_COUNT; i++)
49     { if(eccCurves[i].curveId == curveId)
50       return &eccCurves[i];
51     }
52   return NULL;
53 } static const ECC_CURVE_DATA *
54 GetCurveData(
55 TPM_ECC_CURVE       curveId     // IN: the curveID
56 )
57 { const ECC_CURVE     *curve = _cpri__EccGetParametersByCurveId(curveId);
58   return curve->curveData;
59 }
```

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 { BIGNUM          *bnX;
73   BIGNUM          *bnY;
74   BN_CTX_start(context);
75   bnX = BN_CTX_get(context);
76   bnY = BN_CTX_get(context);
77   if( bnY == NULL
78     // Get the coordinate values
79     || EC_POINT_get_affine_coordinates_GFp(group, ecP, bnX, bnY, context) != 1
80     // Convert x
81     || (!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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>the TPM_ECC_CURVE is not valid</td>
</tr>
<tr>
<td>-&gt;</td>
<td>points to a structure in groupContext static EC_GROUP *</td>
</tr>
</tbody>
</table>

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

// Convert y
|| (!BnTo2B(&p->y.b, bnY, size))
)   FAIL(FATAL_ERROR_INTERNAL);

BN_CTX_end(context);
return TRUE;
B.10.4.1.2. PointFrom2B()

This function sets the coordinates of an existing BN Point from a TPMS_ECC_POINT.

```c
static EC_POINT *
PointFrom2B(
  EC_GROUP  *group,  // IN: the group for the point
  EC_POINT  *ecP,    // IN: an existing BN point in the group
  TPMS_ECC_POINT  *p,  // IN: the 2B coordinates of the point
  BN_CTX     *context  // IN: the BIGNUM context
)
{
  BIGNUM  *bnX;
  BIGNUM  *bnY;

  // If the point is not allocated then just return a NULL
  if(ecP == NULL)
    return NULL;

  BN_CTX_start(context);
  bnX = BN_CTX_get(context);
  bnY = BN_CTX_get(context);
  if(  // Set the coordinates of the point
      !BN_bin2bn(p->x.t.buffer, p->x.t.size, bnX) == NULL
    && !BN_bin2bn(p->y.t.buffer, p->y.t.size, bnY) == NULL
    && !EC_POINT_set_affine_coordinates_GFp(group, ecP, bnX, bnY, context)
  )
    FAIL(FATAL_ERROR_INTERNAL);

  BN_CTX_end(context);
  return ecP;
}
```
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.

```c
static EC_POINT *
EccInitPoint2B(
    EC_GROUP *group,    // IN: group for the point
    TPMS_ECC_POINT *p,   // IN: the coordinates for the point
    BN_CTX *context     // IN: the BIGNUM context
) {
    EC_POINT *ecP;
    BN_CTX_start(context);
    ecP = EC_POINT_new(group);
    if(PointFrom2B(group, ecP, p, context) == NULL)
        FAIL(FATAL_ERROR_INTERNAL);
    BN_CTX_end(context);
    return ecP;
}
```

B.10.4.1.4. PointMul()

This function does a point multiply and checks for the result being the point at infinity. $Q = ([A]G + [B]P)$

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_NO_RESULT</td>
<td>point is at infinity</td>
</tr>
<tr>
<td>CRYPT_SUCCESS</td>
<td>point not at infinity</td>
</tr>
</tbody>
</table>

```c
static CRYPT_RESULT
PointMul(
    EC_GROUP *group,    // IN: group curve
    EC_POINT *ecpQ,     // OUT: result
    BIGNUM *bnA,        // IN: scalar for [A]G
    EC_POINT *ecpP,     // IN: point for [B]P
    BIGNUM *bnB,        // IN: scalar for [B]P
    BN_CTX *context     // IN: working context
) {
    if(EC_POINT_mul(group, ecpQ, bnA, ecpP, bnB, context) != 1)
        FAIL(FATAL_ERROR_INTERNAL);
    if(EC_POINT_is_at_infinity(group, ecpQ))
        return CRYPT_NO_RESULT;
    return CRYPT_SUCCESS;
}
```

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 $2^{nLen/2} <= d < n$.

It is a fatal error if $dOut$ or $pln$ is not provided or if the size of $pln$ is larger than MAX_ECC_KEY_BYTES (the largest buffer size of a TPM2B_ECC_PARAMETER)

```c
static void
GetRandomPrivate(
    TPM2B_ECC_PARAMETER *dOut,    // OUT: the qualified random value
```
const TPM2B *pIn      // IN: the maximum value for the key
}  

pAssert(pIn != NULL && dOut != NULL && pIn->size <= MAX_ECC_KEY_BYTES);

// Set the size of the output

while(TRUE)
{
    _cpri__GenerateRandom(dOut->t.size, dOut->t.buffer);

    if(memcmp(dOut->t.buffer, pIn->buffer, pIn->size) < 0)
    {
        // dIn is less than n so make sure that it is at least greater than 
        // 2^(nLen/2). That is, one of the bytes in the upper half of the 
        // value needs to be non-zero

        for(pb = dOut->t.buffer, i = dOut->t.size/2; i > 0; i--)
        {
            if(*pb++ != 0)
            {
                return;
            }
        }
    }
}

B.10.4.1.6. Mod2B()

Function does modular reduction of TPM2B values.

static CRYPT_RESULT

Mod2B( 
    TPM2B *x,       // IN/OUT: value to reduce
    const TPM2B *n  // IN: mod
)
{
    int compare;

    compare = _math__uComp(x->size, x->buffer, n->size, n->buffer);

    if(compare < 0)
    {
        // if x < n, then mod is x

        return CRYPT_SUCCESS;
    }

    if(compare == 0)
    {
        // if x == n then mod is 0

        x->size = 0;
        x->buffer[0] = 0;

        return CRYPT_SUCCESS;
    }

    return _math__Div(x, n, NULL, x);
}

B.10.4.1.7. _cpri__EccPointMultiply

This function computes ‘R := [dIn]G + [uIn]QIn. Where dIn and uIn are scalars, G and QIn are points on the specified curve and G is the default generator of the curve.

The xOut and yOut parameters are optional and may be set to NULL if not used.

It is not necessary to provide uIn if QIn is specified but one of uIn and dIn must be provided. If dIn and QIn are specified but uIn is not provided, then R = [dIn]QIn.
If the multiply produces the point at infinity, the 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>point multiplication succeeded</td>
</tr>
<tr>
<td>CRYPT_POINT</td>
<td>the point $Qin$ is not on the curve</td>
</tr>
<tr>
<td>CRYPT_NO_RESULT</td>
<td>the product point is at infinity</td>
</tr>
</tbody>
</table>

```c
278 CRYPT_RESULT
279 _cpri__EccPointMultiply(
280     TPM2B_ECC_POINT      *Rout,
281     TPM_ECC_CURVE        *curveId,
282     TPM2B_ECC_PARAMETER  *dIn,
283     TPM2B_ECC_PARAMETER  *Qin,
284     TPM2B_ECC_PARAMETER  *uIn
285 )
286 {
287     context = BN_CTX_new();
288     if(context == NULL)
289         FAIL(FATAL_ERROR_ALLOCATION);
290     context = BN_CTX_start(context);
291     if(Qin != NULL)
292         Q = EccInitPoint2B(group, Qin, context);
293     if(                                                  }
294         assert2Bsize(Qin->x.t);
295     assert2Bsize(Qin->y.t);
296     Q = EccInitPoint2B(group, Qin, context);
297         }
initialize the size values of a point

```c
static void
ClearPoint2B(TPMS_ECC_POINT *p) {  // IN: the point
  if (p != NULL) {
    p->x.t.size = 0;
    p->y.t.size = 0;
  }
}
```

B.10.4.1.8. ClearPoint2B()

Initialize the size values of a point
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>computations completed normally</td>
</tr>
<tr>
<td>CRYPT_NO_RESULT</td>
<td>if $K$, $L$, or $E$ was computed to be the point at infinity</td>
</tr>
<tr>
<td>CRYPT_CANCEL</td>
<td>a cancel indication was asserted during this function</td>
</tr>
</tbody>
</table>

```c
CRYPT_RESULT
(cpri__EccCommitCompute(
    TPM2B_ECC_PARAMETER *d,  // IN: d (required)
    TPM2B_ECC_PARAMETER *r,  // IN: the computed r value (required)
    TPM_ECC_CURVE curveId,  // IN: the curve for the computations
    TPMS_ECC_POINT *M,      // IN: M (optional)
    TPMS_ECC_POINT *B,      // IN: B (optional)
    TPM_ECC_POINT *K,      // OUT: [d]B or [r]Q
    TPM_ECC_POINT *L,      // OUT: [r]B
    TPM_ECC_POINT *E,      // OUT: [r]M
    TPMS_ECC_POINT *EccCommitCompute()
    BN_CTX              *context;
    EC_GROUP            *group;
    UINT16               keySizeInBytes;
    CRYPT_RESULT         retVal = CRYPT_SUCCESS;
    // Validate that the required parameters are provided.
    // Note: E has to be provided if computing $E := [r]Q$ or $E := [r]M$. Will do
    // $E := [r]Q$ if both $M$ and $B$ are NULL.
    pAssert( r != NULL && (K != NULL || B == NULL) && (L != NULL || B == NULL) && (E != NULL || (M == NULL && B != NULL)));
    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    bnR = BN_CTX_get(context);
    bnD = BN_CTX_get(context);
    bnX = BN_CTX_get(context);
    bnY = BN_CTX_get(context);
    if(bnY == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    // Initialize the output points in case they are not computed
    ClearPoint2B(K);
    ClearPoint2B(L);
    ClearPoint2B(E);
    if((group = EccCurveInit(curveId, context)) == NULL)
    {
        retVal = CRYPT_PARAMETER;
        goto Cleanup2;
    }
    keySizeInBytes = (UINT16) BN_num_bytes(&group->field);
```
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  // Convert scalars to BIGNUM
437  BnFrom2B(bnR, &r->b);
438  BnFrom2B(bnD, &d->b);
439  // If B is provided, compute K=[d]B and L=[r]B
440  if(B != NULL)
441  {
442    // Allocate the points to receive the value
443    if(  (pK = EC_POINT_new(group)) == NULL
444       || (pL = EC_POINT_new(group)) == NULL)
445      FAIL(FATAL_ERROR_ALLOCATION);
446    // need to compute K = [d]B
447    // Allocate and initialize BIGNUM version of B
448    pB = EccInitPoint2B(group, B, context);
449    // do the math for K = [d]B
450    if((retVal = PointMul(group, pK, NULL, pB, bnD, context)) != CRYPT_SUCCESS)
451       goto Cleanup;
452    // Convert BN K to TPM2B K
453    Point2B(group, K, pK, keySizeInBytes, context);
454    // compute L= [r]B after checking for cancel
455    if(_plat__IsCanceled())
456    {
457      retVal = CRYPT_CANCEL;
458      goto Cleanup;
459    }
460    // compute L = [r]B
461    if((retVal = PointMul(group, pL, NULL, pB, bnR, context)) != CRYPT_SUCCESS)
462       goto Cleanup;
463    // Convert BN L to TPM2B L
464    Point2B(group, L, pL, keySizeInBytes, context);
465  }
466  if(M != NULL || B == NULL)
467  {
468    // if this is the third point multiply, check for cancel first
469    if(B != NULL && _plat__IsCanceled())
470    {
471      retVal = CRYPT_CANCEL;
472      goto Cleanup;
473    }
474    // Allocate E
475    if((pE = EC_POINT_new(group)) == NULL)
476      FAIL(FATAL_ERROR_ALLOCATION);
477    // Create BIGNUM version of M unless M is NULL
478    if(M != NULL)
479    {
480      // M provided so initialize a BIGNUM M and compute E = [r]M
481      pM = EccInitPoint2B(group, M, context);
482      retVal = PointMul(group, pE, NULL, pM, bnR, context);
483    }
484    else
485    {
486      // compute E = [r]Q (this is only done if M and B are both NULL
487      if(retVal == CRYPT_SUCCESS)
488        // Convert E to 2B format
489        Point2B(group, E, pE, keySizeInBytes, context);
490    }
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__EcclsPointOnCurve()

This function is used to test if a point is on a defined curve. It does this by checking that
\( y^2 \mod p = x^3 + a \cdot x + b \mod p \)

It is a fatal error if \( Q \) is not specified (is NULL).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>point is on curve</td>
</tr>
<tr>
<td>FALSE</td>
<td>point is not on curve or curve is not supported</td>
</tr>
</tbody>
</table>

513 BOOL _cpri__EcclsPointOnCurve(
514     TPM_ECC_CURVE    curveId, // IN: the curve selector
515     TPMS_ECC_POINT   *Q    // IN: the point.
516 )
517 {
518     BN_CTX     *context;
519     BIGNUM     *bnX;
520     BIGNUM     *bnY;
521     BIGNUM     *bnA;
522     BIGNUM     *bnB;
523     BIGNUM     *bnP;
524     BIGNUM     *bn3;
525     const     ECC_CURVE_DATA *curveData = GetCurveData(curveId);
526     BOOL                   retVal;
527     pAssert(Q != NULL && curveData != NULL);
528     if((context = BN_CTX_new()) == NULL)
529         FAIL(FATAL_ERROR_ALLOCATION);
530     BN_CTX_start(context);
531     bnX = BN_CTX_get(context);
532     bnY = BN_CTX_get(context);
533     bnA = BN_CTX_get(context);
534     bnB = BN_CTX_get(context);
535     bn3 = BN_CTX_get(context);
536     bnP = BN_CTX_get(context);
537     if(bnP == NULL)
538         FAIL(FATAL_ERROR_ALLOCATION);
539     // Convert values
540     if (  !BN_bin2bn(Q->x.t.buffer, Q->x.t.size, bnX)
541         || !BN_bin2bn(Q->y.t.buffer, Q->y.t.size, bnY)
542         || !BN_bin2bn(curveData->p->buffer, curveData->p->size, bnP)
543         || !BN_bin2bn(curveData->a->buffer, curveData->a->size, bnA)
544         || !BN_set_word(bn3, 3)
545         || !BN_bin2bn(curveData->b->buffer, curveData->b->size, bnB)
546     )
547         pAssert(FALSE);
548     // Do the check
549     if ((bnY * bnY * bnX * bn3 - (bnA * bnX + bnB) * bnX) \mod bnP)
550         pAssert(FALSE);
551     else
552         pAssert(TRUE);
553     return retVal;
554 }
FAIL(FATAL_ERROR_INTERNAL);

// The following sequence is probably not optimal but it seems to be correct.
// compute x^3 + a*x + b mod p
// first, compute a*x mod p
if(!BN_mod_mul(bnA, bnA, bnX, bnP, context))
  // next, compute x^3 mod p
  if(!BN_mod_add(bnA, bnA, bnB, bnP, context))
    // finally, compute X^3 mod p
      if(!BN_mod_add(bnX, bnX, bnA, bnP, context))
        // then compute y^2
          if(!BN_mod_mul(bnY, bnY, bnY, bnP, context))
            FAIL(FATAL_ERROR_INTERNAL);

retVal = BN_cmp(bnX, bnY) == 0;
BN_CTX_end(context);
BN_CTX_free(context);
return retVal;

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).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPTO_PARAMETER</td>
<td>the hash algorithm is not supported</td>
</tr>
</tbody>
</table>

CRYPT_RESULT
_cpri__GenerateKeyEcc(
  TPMS_ECC_POINT *Qout, // OUT: the public point
  TPM2B_ECC_PARAMETER *dOut, // OUT: the private scalar
  TPM_ECC_CURVE curveId, // IN: the curve identifier
  TPM_ALG_ID hashAlg, // IN: hash algorithm to use in the key generation process
  TPM2B *seed, // IN: the seed to use
  const char *label, // IN: A label for the generation process.
  TPM2B *extra, // IN: Party 1 data for the KDF
  UINT32 *counter // IN/OUT: Counter value to allow KDF iteration to be propagated across multiple functions
)

{ const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
  UINT16 keySizeInBytes;
  UINT32 count = 0;
  CRYPTO_RESULT retVal;
  UINT16 lSize;
  UINT16 hLen = _cpri__GetDigestSize(hashAlg);
Validate parameters (these are fatal)

\[
p\text{Assert}(\text{seed} != \text{NULL} \&\& \text{dOut} != \text{NULL} \&\& \text{Qout} != \text{NULL} \&\& \text{curveData} != \text{NULL});
\]

Non-fatal parameter checks.

\[
\text{if}(\text{hLen} <= 0)
    \text{return CRYPT_PARAMETER;}
\]

// If there is a label, size it
\[
\text{if}(\text{label} != \text{NULL})
    \text{for}(\text{lSize} = 0; \text{label}[\text{lSize}++];);
\]

// allocate the local BN values
\[
\text{context} = \text{BN_CTX\_new();}
\]

// The size of the input scalars is limited by the size of the size of a
// TPM2B\_ECC\_PARAMETER. Make sure that it is not irrational.
\[
\text{assert((int)} \text{curveData->n->size} \leq \text{MAX\_ECC\_KEY\_BYTES;}
\]

// Initialize the count value
\[
\text{if}(\text{counter} != \text{NULL})
    \text{count} = *\text{counter};
\]

// Start search for key (should be quick)
\[
\text{for}(); \text{count} != 0; \text{count}++
\]

{  

UINT32\_TO\_BYTE\_ARRAY(\text{count}, \text{marshaledCounter.t.buffer});

\_cpri\_KDFa(\text{hashAlg, seed, label, extra, &marshaledCounter.b,}

\text{BN\_num\_bits(bnN)+64, withExtra, NULL, FALSE});

// Convert the result and modular reduce

// Assume the size variables do not overflow, which should not happen in
// the contexts that this function will be called.
\[
\text{assert(\text{keySizeInBytes} \leq \text{MAX\_ECC\_KEY\_BYTES;}}
\]

\[
\text{if}\ (\text{BN\_bin2bn(\text{curveData->n->buffer, curveData->n->size, bnN})} == \text{NULL}
\]

\[
\text{|| (keySizeInBytes} = (\text{UINT16)} \text{BN\_num\_bits(bnN))} > \text{MAX\_ECC\_KEY\_BYTES)
\]

\[
\text{fail(FATAL\_ERROR\_INTERNAL));}
\]

// Make sure that the result is in the desired range
\[
\text{if}(\text{BN\_num\_bits(bnD}) >= \text{BN\_num\_bits(bnN)} / 2)
\]

\[
\text{if}(\text{BnTo2B(\&dOut->b, bnD, keySizeInBytes) != 1)}
\]

\[
\text{fail(FATAL\_ERROR\_INTERNAL)};
\]

// 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     if(count == 0) // if counter wrapped, then the TPM should go into failure mode
668         FAIL(FATAL_ERROR_INTERNAL);
669
670     // Free up allocated BN values
671     BN_CTX_end(context);
672     BN_CTX_free(context);
673     if(counter != NULL)
674         *counter = count;
675     return retVal;
676 }

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     pAssert(curveData != NULL);
689     // Keep getting random values until one is found that doesn't create a point
690     // at infinity. This will never, ever, ever, ever, ever, happen but if it does
691     // we have to get a next random value.
692     while(TRUE)
693     {
694         GetRandomPrivate(dOut, curveData->p);
695         // _cpri__EccPointMultiply does not return CRYPT_ECC_POINT if no point is
696         // provided. CRYPT_PARAMTER should not be returned because the curve ID
697         // has to be supported. Thus the only possible error is CRYPT_NO_RESULT.
698         retVal = _cpri__EccPointMultiply(Qout, curveId, dOut, NULL, NULL);
699         if(retVal != CRYPT_NO_RESULT)
700             return retVal; // Will return CRYPT_SUCCESS
701     }
702 #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
TPM2B_ECC_PARAMETER *sOut, // OUT: s component of the signature
TPM2B_ECC_PARAMETER *dIn, // IN: the private key
TPM2B_ECC_PARAMETER *digest // IN: the value to sign

BIGNUM *bnK;
BIGNUM *bnIk;
BIGNUM *bnN;
BIGNUM *bnR;
BIGNUM *bnD;
BIGNUM *bnZ;

TPM2B_ECC_PARAMETER k;
TPMS_ECC_POINT R;
BN_CTX *context;
CRYPT_RESULT retVal = CRYPT_SUCCESS;
const ECC_CURVE_DATA *curveData = GetCurveData(curveId);

pAssert(rOut != NULL && sOut != NULL && dIn != NULL && digest != NULL);
context = BN_CTX_new();
if (context == NULL)
FAIL(FATAL_ERROR_ALLOCATION);
BN_CTX_start(context);
bnN = BN_CTX_get(context);
bnZ = BN_CTX_get(context);
bnR = BN_CTX_get(context);
bnD = BN_CTX_get(context);
bnIk = BN_CTX_get(context);
bnK = BN_CTX_get(context);

// Assume the size variables do not overflow, which should not happen in
// the contexts that this function will be called.
pAssert(curveData->n->size <= MAX_ECC_PARAMETER_BYTES);
if (bnK == NULL || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
FAIL(FATAL_ERROR_INTERNAL);

// The algorithm as described in "Suite B Implementer's Guide to FIPS 186-3(ECDSA)"
// 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a per-message
// secret number and its inverse modulo n. Since n is prime, the
// output will be invalid only if there is a failure in the RBG.
// 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
// multiplication (see [Routines]), where G is the base point included in
// the set of domain parameters.
// 3. Compute r = xR mod n. If r = 0, then return to Step 1. 1.
// 4. Use the selected hash function to compute H = Hash(M).
// 5. Convert the bit string H to an integer e as described in Appendix B.2.
// 6. Compute s = (k^-1 * (e + d * r)) mod n. If s = 0, return to Step 1.2.
// 7. Return (r, s).

// Generate a random value k in the range 1 <= k < n
// Want a K value that is the same size as the curve order
k.t.size = curveData->n->size;

while(TRUE) // This implements the loop at step 6. If s is zero, start over.
{
    while(TRUE)
    {
        // Step 1 and 2 -- generate an ephemeral key and the modular inverse
        // of the private key.
        while(TRUE)
        {
            GetRandomPrivate(&k, curveData->n);
            // Do the point multiply to generate a point and check to see if
```c
// the point it at infinity
if(_cpri_EccPointMultiply(&R, curveId, &k, NULL, NULL) != CRYPT_NO_RESULT)
    break;  // can only be CRYPT_SUCCESS
}

// x coordinate is mod p. Make it mod n
// Assume the size variables do not overflow, which should not happen
// in the contexts that this function will be called.
assert2Bsize(R.x.t);
BN_bin2bn(R.x.t.buffer, R.x.t.size, bnR);
BN_mod(bnR, bnR, bnN, context);

// Make sure that it is not zero;
if(BN_is_zero(bnR))
    continue;

// Make sure that a modular inverse exists
// Assume the size variables do not overflow, which should not happen
// in the contexts that this function will be called.
assert2Bsize(k.t);
BN_bin2bn(k.t.buffer, k.t.size, bnK);
if( BN_mod_inverse(bnIk, bnK, bnN, context) != NULL)
    break;
}
```

842  // d = z + r * d
843   || !BN_add(bnD, bnZ, bnD)
844  // d = k^(-1)(z + r * d)(mod n)
845   || !BN_mod_mul(bnD, bnIk, bnD, bnN, context)
846  // convert to TPM2B format
847   || !BnTo2B(&sOut->b, bnD, curveData->n->size)
848  // and write the modular reduced version of r
849  // NOTE: this was deferred to reduce the number of
850  // error checks.
851   || !BnTo2B(&rOut->b, bnR, curveData->n->size))
852   FAIL(FATAL_ERROR_INTERNAL);
853  if(!BN_is_zero(bnD))
854     break;  // signature not zero so done
855  // if the signature value was zero, start over
856  }
857  // Free up allocated BN values
858  BN_CTX_end(context);
859  BN_CTX_free(context);
860  return retVal;
861  }
862 #endif //%
863 #if defined TPM_ALG_ECDAA || defined TPM_ALG_ECSCHNORR //%

B.10.4.1.14. EcDaa()

This function is used to perform a modified Schnorr signature for ECDA.
This function performs \( s = k + T \cdot d \mod n \) where
a) 'k is a random, or pseudo-random value used in the commit phase
b) \( T \) is the digest to be signed, and
c) \( d \) is a private key.
If \( t\text{ln} \) is NULL then use \( t\text{Out} \) as \( T \)

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>signature created</td>
</tr>
</tbody>
</table>

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 {
881   BN_CTX                  *context;
882   const TPM2B            *n;
883   const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
884   BOOL                    OK = TRUE;
885 // Parameter checks
886   pAssert(   sOut != NULL && dIn != NULL && tOut != NULL

Page 450
Published
Family "02"
March 15, 2013
Copyright © TCG 2006-2013
Level 00 Revision 00.96
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         FAIL(FATAL_ERROR_INTERNAL);
923     // Compute T = T mod n
924     OK = OK && BN_mod(bnT, bnT, bnN, context);
925
926     // compute (s = k + T * d mod n)
927     // d = T * d mod n
928     OK = OK && BN_mod_mul(bnD, bnT, bnD, bnN, context) == 1;
929     // d = k + T * d mod n
930     OK = OK && BN_mod_add(bnD, bnK, bnD, bnN, context) == 1;
931     // s = d
932     OK = OK && BnTo2B(&sOut->b, bnD, n->size);
933     // r = T
934     OK = OK && BnTo2B(&tOut->b, bnT, n->size);
935     if(!OK)
936         FAIL(FATAL_ERROR_INTERNAL);
937
938     // Cleanup
939     BN_CTX_end(context);
940     BN_CTX_free(context);
941
942     return CRYPT_SUCCESS;
943 }
944 #endif /*
945 #ifdef TPM_ALG_ECSCHNORR /*

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) \(s = k + r \cdot ds\)

g) return the tuple \(T, s\)

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>signature created</td>
</tr>
<tr>
<td>CRYPT_SCHEME</td>
<td>hashAlg can't produce zero-length digest</td>
</tr>
</tbody>
</table>

```c
static CRYPT_RESULT
SchnorrEcc(
    TPM2B_ECC_PARAMETER     *rOut,   // OUT: r component of the signature
    TPM2B_ECC_PARAMETER     *sOut,   // OUT: s component of the signature
    TPM_ALG_ID              hashAlg, // IN: hash algorithm used
    TPM_ECC_CURVE           curveId, // IN: the curve used in signing
    TPM2B_ECC_PARAMETER     *dIn,    // IN: the private key
    TPM2B_ECC_PARAMETER     *digest, // IN: the digest to sign
    TPM2B_ECC_PARAMETER     *kIn      // IN: for testing
)
{
    TPM2B_ECC_PARAMETER     k;
    BN_CTX                  *context;
    const TPM2B             *n;
    EC_POINT                *pR = NULL;
    EC_GROUP                *group = NULL;
    CPRI_HASH_STATE         hashState;
    UINT16                   digestSize = _cpri__GetDigestSize(hashAlg);
    const ECC CURVE DATA    *curveData = GetCurveData(curveId);
    TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_PARAMETER_BYTES));
    TPM2B_T                  T2b;
    BOOL                     OK = TRUE;

    // Parameter checks

    // Must have a place for the 'r' and 's' parts of the signature, a private
    // key ('d')
    pAssert(   rOut != NULL && sOut != NULL && dIn != NULL
              && digest != NULL && curveData != NULL);

    // to save key strokes
    n = curveData->n;

    // If the digest does not produce a hash, then null the signature and return
    // a failure.
    if(digestSize == 0)
    {
        rOut->t.size = 0;
        sOut->t.size = 0;
        return CRYPT_SCHEME;
    }

    // Allocate big number values
    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    bnR = BN_CTX_get(context);
    bnN = BN_CTX_get(context);
    bnK = BN_CTX_get(context);
```

bnT = BN_CTX_get(context);
bnD = BN_CTX_get(context);
if (bnD == NULL)
    // initialize the group parameters
    || (group = EccCurveInit(curveId, context)) == NULL
    // allocate a local point
    || (pR = EC_POINT_new(group)) == NULL
)
    FAIL(FATAL_ERROR_ALLOCATION);
if (BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
    FAIL(FATAL_ERROR_INTERNAL);
while (OK)
{
    // a) set k to a random value such that 1 ≤ k ≤ n-1
    if (kIn != NULL)
        {
            Copy2B(&k.b, &kIn->b); // copy input k if testing
            OK = FALSE; // not OK to loop
        }
    else
        // If get a random value in the correct range
        GetRandomPrivate(&k, n);
        // Convert 'k' and generate pR = [k]G
        BnFrom2B(bnK, &k.b);
    // b) compute E ≔ (xE, yE) = [k]G
    if (PointMul(group, pR, bnK, NULL, NULL, context) == CRYPT_NO_RESULT)
        // c) if E is the point at infinity, go to a)
        continue;
    // d) compute e ≔ xE (mod n)
    // Get the x coordinate of the point
    EC_POINT_get_affine_coordinates_GFp(group, pR, bnR, NULL, context);
    // make (mod n)
    BN_mod(bnR, bnR, bnN, context);
    // e) if e is zero, go to a)
    if (BN_is_zero(bnR))
        continue;
    // f) compute r = HashSchemeHash(P || e) (mod n)
    _cpri__StartHash(hashAlg, FALSE, &hashState);
    _cpri__UpdateHash(&hashState, digest->size, digest->buffer);
    _cpri__UpdateHash(&hashState, T2b_t.size, T2b_t.buffer);
    if (_cpri__CompleteHash(&hashState, digestSize, T2b_t.buffer) != digestSize)
        FAIL(FATAL_ERROR_INTERNAL);
    T2b_t.size = digestSize;
    BnFrom2B(bnT, &T2b_t.b);
    BN_div(NULL, bnT, bnT, bnN, context);
    BnTo2B(&rOut->b, bnT, (UINT16)BN_num_bytes(bnT));
    // We have a value and we are going to exit the loop successfully
    OK = TRUE;
    break;
}
// Cleanup
EC_POINT_free(pR);
EC_GROUP_free(group);
BN_CTX_end(context);
BN_CTX_free(context);

// If we have a value, finish the signature
if (OK)
    return EcDaa(rOut, sOut, curveId, dIn, NULL, &k);
else
    return CRYPT_NO_RESULT;
}
#endif //%
#ifdef TPM_ALG_SM2 //%
#ifdef _SM2_SIGN_DEBUG //%
static int cmp_bn2hex(BIGNUM *bn,
                      const char *c
                      )
{  
    result;
    BIGNUM *bnC = BN_new();
    pAssert(bnC != NULL);
    BN_hex2bn(&bnC, c);
    result = BN_ucmp(bn, bnC);
    BN_free(bnC);
    return result;
}
static int cmp_2B2hex(TPM2B *a,
                      const char *c
                      )
{  
    result;
    sl = strlen(c);
    BIGNUM *bnA;
    result = (a->size * 2) - sl;
    if(result != 0)
        return result;
    pAssert((bnA = BN_bin2bn(a->buffer, a->size, NULL)) != NULL);
    result = cmp_bn2hex(bnA, c);
    BN_free(bnA);
    return result;
}
static void cpy_hexTo2B(TPM2B *b,
                        const char *c
                        )
{  
    BIGNUM *bnB = BN_new();
    pAssert((strlen(c) & 1) == 0); // must have an even number of digits
    b->size = strlen(c) / 2;
    BN_hex2bn(&bnB, c);
    pAssert(bnB != NULL);
    BnTo2B(b, bnB, b->size);
    BN_free(bnB);
}
#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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>sign worked</td>
</tr>
</tbody>
</table>

```c
static CRYPT_RESULT
SignSM2(
    TPM2B_ECC_PARAMETER *rOut,  // OUT: r component of the signature
    TPM2B_ECC_PARAMETER *sOut,  // OUT: s component of the signature
    TPM_ECC_CURVE curveId,      // IN: the curve used in signing
    TPM2B_ECC_PARAMETER *dIn,   // IN: the private key
    TPM2B *digest              // IN: the digest to sign
)
{
    BIGNUM                  *bnR;
    BIGNUM                  *bnS;
    BIGNUM                  *bnN;
    BIGNUM                  *bnK;
    BIGNUM                  *bnX1;
    BIGNUM                  *bnD;
    BIGNUM                  *bnT;  // temp
    BIGNUM                  *bnE;
    BN_CTX                  *context;
    TPM2B_TYPE(DIGEST, MAX_DIGEST_SIZE);
    TPM2B_ECC_PARAMETER     k;
    TPM3_ECC_POINT          p2Br;
    const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
    pAssert(curveData != NULL);
    context = BN_CTX_new();
    BN_CTX_start(context);
    bnK = BN_CTX_get(context);
    bnR = BN_CTX_get(context);
    bnS = BN_CTX_get(context);
    bnX1 = BN_CTX_get(context);
    bnN = BN_CTX_get(context);
    bnD = BN_CTX_get(context);
    bnT = BN_CTX_get(context);
    bnE = BN_CTX_get(context);
    if(bnE == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BnFrom2B(bnE, digest);
    BnFrom2B(bnN, curveData->n);
    BnFrom2B(bnD, &dIn->b);
}
#endif
```

### Notes

- **A3:** Use random number generator to generate random number 1 <= k <= n-1;
- **A4:** numbers are from the SM2 standard

```c
loop:
{  // Get a random number
    _cpri__GenerateRandom(k.t.size, k.t.buffer);
}
```

### Definitions

- **BN_hex2bn:**
  - `"B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76"`;
  - `"128B2FA8BD43368C688BD803DFF79792A519A55171B1B650C23661D15897263"`
BN_hex2bn(&bnK, "6CB2BD9385C175C94F94E934817663FC176D925DD72B727260DBAAE1FB2F96F");
BNTo2B(&k.b, bnK, 32);
k.t.size = 32;
#endif
#endif
// make sure that the number is 0 < k < n
BNFrom2B(bnK, &k.b);
if (BN_ucmp(bnK, bnN) >= 0 || BN_is_zero(bnK))
   goto loop;
// A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according to details specified in 4.2.7 in Part 1 of this document, transform the data type of x1 into an integer;
if (_cpri__EccPointMultiply(&p2Br, curveId, &k, NULL, NULL) == CRYPT_NO_RESULT)
goto loop;
BNFrom2B(bnX1, &p2Br.x.b);
// A5: Figure out r = (e + x1) mod n,
if(!BN_mod_add(bnR, bnE, bnX1, bnN, context))
   FAIL(FATAL_ERROR_INTERNAL);
if(BN_is_zero(bnR) || BN_ucmp(bnR, bnN) == 0)
   goto loop;
// A6: Figure out s = ((1 + dA)^-1 * (k - r * dA)) mod n, if s=0, return to A3;
// compute t = (1+d)^-1
BN_copy(bnT, bnD);
if(!BN_add_word(bnT, 1) || !BN_mod_inverse(bnT, bnT, bnN, context) // (1 + dA)^-1 mod n)
   FAIL(FATAL_ERROR_INTERNAL);
pAssert(cmp_bn2hex(bnT, "79BF3052C80DA7B9396C96914A18CBB2D96D8555256E831273A7D4F5F956") == 0);
#endif
// if r=0 or r+k=n, return to A3;
if(!BN_add(bnT, bnK, bnR))
   FAIL(FATAL_ERROR_INTERNAL);
if(BN_is_zero(bnR) || BN_ucmp(bnR, bnN) == 0)
   goto loop;
// A7: According to details specified in 4.2.1 in Part 1 of this document, transform the data type of r, s into bit strings, signature of message M is (r, s).
BnTo2B(&rOut->b, bnR, curveData->n->size);
Trusted Platform Module Library

Part 4: Supporting Routines

B.10.4.1.17. _cpri__SignEcc()

This function is the dispatch function for the various ECC-based signing schemes.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SCHEME</td>
<td>scheme is not supported</td>
</tr>
</tbody>
</table>

1246  BnTo2B(&sOut->b, bnS, curveData->n->size);
1247  #ifdef _SM2_SIGN_DEBUG
1248    pAssert(cmp_2B2hex(&rOut->b,
1249         "40F1EC59F793D9F49E09DCEF49130D4194F79FB1ED2CAA55BACDB49C4E755D1")
1250      == 0);
1251    pAssert(cmp_2B2hex(&sOut->b,
1252         "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEBEB7")
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SCHEME</td>
<td>scheme is not supported</td>
</tr>
</tbody>
</table>

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 needed
1266     TPM_ECC_CURVE        curveId, // IN: the curve used in the signature process
1267     TPM2B_ECC_PARAMETER *dIn, // IN: the private key
1268     TPM2B_ECC_PARAMETER *kIn // IN: k for input
1269     )
1270  {
1271    switch (scheme)
1272    {
1273    case TPM_ALG_ECDSA:
1274      // SignEcdsa always works
1275      return SignEcdsa(rOut, sOut, curveId, dIn, digest);
1276      break;
1277    #ifdef TPM_ALG_ECDAA
1278      case TPM_ALG_ECDAA:
1279      if(rOut != NULL)
1280         rOut->b.size = 0;
1281      return EcDaa(rOut, sOut, curveId, dIn, digest, kIn);
1282      break;
1283    #endif
1284    #ifdef TPM_ALG_ECSCHNORR
1285      case TPM_ALG_ECSCHNORR:
1286      return SchnorrEcc(rOut, sOut, hashAlg, curveId, dIn, digest, kIn);
1287      break;
1288    #endif
1289    #ifdef TPM_ALG_SM2
1290      case TPM_ALG_SM2:
1291      return SignSM2(rOut, sOut, curveId, dIn, digest);
1292      break;
1293    #endif
1294    default:
1295      return CRYPT_SCHEME;
1296    }
1297  }
B.10.4.1.18. ValidateSignatureEcdsa()

This function validates an ECDSA signature.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>signature valid</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>signature not valid</td>
</tr>
</tbody>
</table>

```
static CRYPT_RESULT ValidateSignatureEcdsa(
    TPM2B_ECC_PARAMETER     *rIn, // IN: r component of the signature
    TPM2B_ECC_PARAMETER     *sIn, // IN: s component of the signature
    TPM_ECC_CURVE            curveId, // IN: the curve used in the signature
    TPM2B_ECC_POINT          *Qin, // IN: the public point of the key
    TPM2B                   *digest // IN: the digest that was signed
) {
    TPM2B_ECC_PARAMETER      U1;
    TPM2B_ECC_PARAMETER      U2;
    TPM_ECC_POINT            R;
    const TPM2B             *n;
    const BN_CTX             *context;
    const EC_POINT           *pQ = NULL;
    const EC_GROUP           *group = NULL;
    const BIGNUM             *bnU1;
    const BIGNUM             *bnU2;
    const BIGNUM             *bnR;
    const BIGNUM             *bnS;
    const BIGNUM             *bnW;
    const BIGNUM             *bnV;
    const BIGNUM             *bnN;
    const BIGNUM             *bnE;
    const BIGNUM             *bnGx;
    const BIGNUM             *bnGy;
    const BIGNUM             *bnQx;
    const BIGNUM             *bnQy;
    CRYPT_RESULT             retVal = CRYPT_FAIL;
    int t;
    const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
    // The curve selector should have been filtered by the unmarshaling process
    pAssert (curveData != NULL);
    n = curveData->n;
    // 1. If r and s are not both integers in the interval [1, n - 1], output
    //    INVALID.
    if(( _math__uComp(rIn->t.size, rIn->t.buffer, n->size, n->buffer) >= 0
        || _math__uComp(sIn->t.size, sIn->t.buffer, n->size, n->buffer) >= 0
    )
        return CRYPT_FAIL;
    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    bnR = BN_CTX_get(context);
    bnS = BN_CTX_get(context);
    bnN = BN_CTX_get(context);
    bnE = BN_CTX_get(context);
    bnV = BN_CTX_get(context);
    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  // initialize the group parameters
1372  || (group = EccCurveInit(curveId, context)) == NULL
1373  // allocate a local point
1374  || (pQ = EC_POINT_new(group)) == NULL
1375  // use the public key values (QxIn and QyIn) to initialize Q
1376  || BN_bin2bn(Qin->x.t.buffer, Qin->x.t.size, bnQx) == NULL
1377  || BN_bin2bn(Qin->x.t.buffer, Qin->x.t.size, bnQy) == NULL
1378  || !EC_POINT_set_affine_coordinates_GFp(group, pQ, bnQx, bnQy, context)
1379  // convert the signature values
1380  || BN_bin2bn(rIn->t.buffer, rIn->t.size, bnR) == NULL
1381  || BN_bin2bn(sIn->t.buffer, sIn->t.size, bnS) == NULL
1382  // convert the curve order
1383  || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
1384  FAIL(FATAL_ERROR_INTERNAL);
1385
1386  // 2. Use the selected hash function to compute H0 = Hash(M0).
1387  // This is an input parameter
1388  // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
1389  t = (digest->size > rIn->t.size) ? rIn->t.size : digest->size;
1390  if(BN_bin2bn(digest->buffer, t, bnE) == NULL)
1391  FAIL(FATAL_ERROR_INTERNAL);
1392
1393  // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
1394  if (BN_mod_inverse(bnW, bnS, bnN, context) == NULL)
1395  FAIL(FATAL_ERROR_INTERNAL);
1396
1397  // 5. Compute u1 = (e' * w) mod n, and compute u2 = (r' * w) mod n.
1398  if( !BN_mod_mul(bnU1, bnE, bnW, bnN, context)
1399     || !BN_mod_mul(bnU2, bnR, bnW, bnN, context))
1400  FAIL(FATAL_ERROR_INTERNAL);
1401
1402  BnTo2B(&U1.b, bnU1, (UINT16) BN_num_bytes(bnU1));
1403  BnTo2B(&U2.b, bnU2, (UINT16) BN_num_bytes(bnU2));
1404
1405  // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
1406  // scalar multiplication and EC addition (see [Routines]). If R is equal to
1407  // the point at infinity O, output INVALID.
1408  if(_cpri__EccPointMultiply(&R, curveId, &U1, Qin, &U2) == CRYPT_SUCCESS)
1409  {
1410     // 7. Compute v = Rx mod n.
1411     if( BN_bin2bn(R.x.t.buffer, R.x.t.size, bnV) == NULL
1412         || !BN_mod(bnV, bnV, bnN, context))
1413      FAIL(FATAL_ERROR_INTERNAL);
8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
   if (BN_cmp(bnV, bnR) == 0)
   
   retVal = CRYPT_SUCCESS;

   return retVal;

// TPM_ALG_ECDSA

B.10.4.1.19. ValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>signature valid</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>signature not valid</td>
</tr>
<tr>
<td>CRYPT_SCHEME</td>
<td>hashAlg is not supported</td>
</tr>
</tbody>
</table>

ValidateSignatureEcSchnorr(

tpm2b_ECC_PARAMETER *rIn,   // IN: r component of the signature
  tpm2b_ECC_PARAMETER *sIn,   // IN: s component of the signature
  TPM_ALG_ID hashAlg,        // IN: hash algorithm of the signature
  TPM_ECC_CURVE curveId,     // IN: the curve used in the signature
  // process
  TPMS_ECC_POINT *Qin,        // IN: the public point of the key
  TPM2B *digest              // IN: the digest that was signed
)

  TPM2B_ECC_PARAMETER *n;
  const TPMS_ECC_POINT pE;
  const TPM2B *n;
  CPRI_HASH_STATE hashState;
  TPM2B_DIGEST rPrime;
  TPM2B_ECC_PARAMETER minusR;
  UINT16 digestSize = _cpri__GetDigestSize(hashAlg);
  const ECC_CURVE_DATA *curveData = GetCurveData(curveId);

  // The curve parameter should have been filtered by unmarshaling code
  if (digestSize == 0)
      return CRYPT_SCHEME;

  // Input parameter validation
  pAssert(rIn != NULL && sIn != NULL && Qin != NULL && digest != NULL);
  n = curveData->n;

  if (sIn or rIn are not between 1 and N-1, signature check fails
     if (   _math__uComp(sIn->b.size, sIn->b.buffer, n->size, n->buffer) >= 0
         || _math__uComp(rIn->b.size, rIn->b.buffer, n->size, n->buffer) >= 0
          )
      return CRYPT_FAIL;

  //E = [s]InG - [r]nQ
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     _math__Normalize2B(&pE.x.b);
1483
1484     // rPrime = h(digest || pE.x) mod n;
1485     _cpri__StartHash(hashAlg, FALSE, &hashState);
1486     _cpri__UpdateHash(&hashState, digest->size, digest->buffer);
1487     _cpri__UpdateHash(&hashState, pE.x.t.size, pE.x.t.buffer);
1488     if(_cpri__CompleteHash(&hashState, digestSize, rPrime.t.buffer) != digestSize)
1489         FAIL(FATAL_ERROR_INTERNAL);
1490
1491     rPrime.t.size = digestSize;
1492
1493     // rPrime = rPrime (mod n)
1494     if(Mod2B(&rPrime.b, n) != CRYPT_SUCCESS)
1495         FAIL(FATAL_ERROR_INTERNAL);
1496
1497     // If rIn and rPrime are not the same size, denormalize rPrime.
1498     if(rIn->t.size > rPrime.t.size)
1499         _math__Denormalize2B(&rPrime.b, rIn->t.size);
1500
1501     // see if the values match
1502     if ( rIn->t.size == rPrime.t.size
1503         && (memcmp(rIn->t.buffer, rPrime.t.buffer, rIn->t.size) == 0))
1504         return CRYPT_SUCCESS;
1505     else
1506         return CRYPT_FAIL;
1507 #endif  //% TPM_ALG_ECSCHNORR
1508 #ifdef  TPM_ALG_SM2  //%

B.10.4.1.20. ValidateSignatureSM2Dsa()

This function is used to validate an SM2 signature.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>signature valid</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>signature not valid</td>
</tr>
</tbody>
</table>

static CRYPT_RESULT
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     TPM_ECC_POINT           *Qin,    // IN: the public point of the key
1517     TPM2B                  *digest   // IN: the digest that was signed
1518 )
1519 {
1520     BIGNUM          *bnR;
1521     BIGNUM          *bnRp;
1522     BIGNUM          *bnT;
1523     BIGNUM          *bsS;
1524     BIGNUM          *bnE;
EC_POINT *pQ;
BN_CTX *context;
EC_GROUP *group = NULL;
const ECC_CURVE_DATA *curveData = GetCurveData(curveId);

const

if((context = BN_CTX_new()) == NULL || curveData == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

bnR = BN_CTX_get(context);
bnRp= BN_CTX_get(context);
bnE = BN_CTX_get(context);
bnT = BN_CTX_get(context);
bnS = BN_CTX_get(context);

if(bnS == NULL || (group = EccCurveInit(curveId, context)) == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

#ifdef _SM2_SIGN_DEBUG
    cpy_hexTo2B(&Qin->x.b, "0AE4C7798AA0F119471BEE11825BE46202BB79E2A5844495E97C04FF4DF2548A");
    cpy_hexTo2B(&Qin->y.b, "7C0240F88F1CD4E16352A73C17B7F16F07353E53A176D684A9FE0C6BB798E857");
    cpy_hexTo2B(digest, "B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76");
#endif

pQ = EccInitPoint2B(group, Qin, context);

#ifdef _SM2_SIGN_DEBUG
    pAssert(EC_POINT_get_affine_coordinates_GFp(group, pQ, bnT, bnS, context));
    pAssert(cmp_bn2hex(bnT, "0AE4C7798AA0F119471BEE11825BE46202BB79E2A5844495E97C04FF4DF2548A") == 0);
#endif

BnFrom2B(bnR, &rIn->b);
BnFrom2B(bnS, &sIn->b);
BnFrom2B(bnE, digest);

#ifdef _SM2_SIGN_DEBUG
    // Make sure that the input signature is the test signature
    pAssert(cmp_2B2hex(&rIn->b, "40F1EC59F793D9F49E09DCEF49130D4194F79FB1ED2CAA55BACDB49C4E755D1") == 0);
    pAssert(cmp_2B2hex(&sIn->b, "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7") == 0);
#endif

// a) verify that r and s are in the inclusive interval 1 to (n - 1)
    fail = BN_is_zero(bnR) || (BN_ucmp(bnR, &group->order) >= 0);
    fail = BN_is_zero(bnS) || (BN_ucmp(bnS, &group->order) >= 0) || fail;
    if(fail)
        // There is no reason to continue. Since r and s are inputs from the caller,
        // they can know that the values are not in the proper range. So, exiting here
        // does not disclose any information.
        goto Cleanup;

// b) compute t := (r + s) mod n
    if(!BN_mod_add(bnT, bnR, bnS, &group->order, context))
        FAIL(FATAL_ERROR_INTERNAL);

#ifdef _SM2_SIGN_DEBUG
    pAssert(cmp_bn2hex(bnT, "2B75F07ED7EC7CC1C8986B991F441AD324D6D619FE06DD63ED32E0C997C801")
#endif
== 0);
#pragma endregion

// c) verify that t > 0
if (BN_is_zero(bnT)) {
    fail = TRUE;
    // set to a value that should allow rest of the computations to run without
    // trouble
    BN_copy(bnT, bnS);
}

// d) compute (x, y) := [s]G + [t]Q
if (!EC_POINT_mul(group, pQ, bnS, pQ, bnT, context))
    FAIL(FATAL_ERROR_INTERNAL);

// Get the x coordinate of the point
if (!EC_POINT_get_affine_coordinates_GFp(group, pQ, bnT, NULL, context))
    FAIL(FATAL_ERROR_INTERNAL);

#ifdef _SM2_SIGN_DEBUG
    pAssert(cmp_bn2hex(bnT, "110FCDA57615705D5E7B9324AC4B856D23E6D9188B2AE47759514657CE25D112") == 0);
#endif

// e) compute r' := (e + x) mod n (the x coordinate is in bnT)
if (!BN_mod_add(bnRp, bnE, bnT, &group->order, context))
    FAIL(FATAL_ERROR_INTERNAL);

// f) verify that r' = r
fail = BN_ucmp(bnR, bnRp) != 0 || fail;

Cleanup:
if (pQ) EC_POINT_free(pQ);
if (group) EC_GROUP_free(group);
BN_CTX_end(context);
BN_CTX_free(context);
if (fail)
    return CRYPT_FAIL;
else
    return CRYPT_SUCCESS;
#endif

B.10.4.1.21. _cpri__ValidateSignatureEcc()

This function validates

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>signature is valid</td>
</tr>
<tr>
<td>CRYPT_FAIL</td>
<td>not a valid signature</td>
</tr>
<tr>
<td>CRYPT_SCHEME</td>
<td>unsupported scheme or hash algorithm</td>
</tr>
</tbody>
</table>

BOOL _cpri__ValidateSignatureEcc(
    TPM2B_ECC_PARAMETER *rIn, // IN: r component of the signature
    TPM2B_ECC_PARAMETER *sIn, // IN: s component of the signature
    TPM_ALG_ID scheme,      // IN: the scheme selector
    TPM_ALG_ID hashAlg,     // IN: the hash algorithm used (not used in all schemes)
    TPM_ECC_CURVE curveId,  // IN: the curve used in the signature process
    TPM_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         #ifdef TPM_ALG_ECSCHNORR
1653             case TPM_ALG_ECSCHNORR:
1654                 return ValidateSignatureEcSchnorr(rIn, sIn, hashAlg, curveId, Qin,
1655                                 digest);
1656                 break;
1657             #endif
1658         #ifdef TPM_ALG_SM2
1659             case TPM_ALG_SM2:
1660                 return ValidateSignatureSM2Dsa(rIn, sIn, curveId, Qin, digest);
1661             #endif
1662         default:
1663             break;
1664       }
1665       return CRYPT_SCHEME;
1666   }
1667   #if CC_ZGen_2Phase == YES //
1668   #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 xqi using the convention specified in Appendix C. 3.
i) Calculate xqm = xqi mod 2^ceil(f/2) (where f = ceil(log2(n)).
j) Calculate the associate value function avf(Q) = xqm + 2ceil(f / 2)

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. 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

```c
static CRYPT_RESULT 
C_2_2_MQV(
  TPMS_ECC_POINT *outZ,   // OUT: the computed point
  TPM_ECC_CURVE  curveId, // IN: the curve for the computations
  TPM2B_ECC_PARAMETER *dsA, // IN: static private TPM key
  TPM2B_ECC_PARAMETER *deA,  // IN: ephemeral private TPM key
  TPMS_ECC_POINT *QsB,  // IN: static public party B key
  TPMS_ECC_POINT *QeB  // IN: ephemeral public party B key
) {
  BN_CTX                  *context;
  EC_POINT                *pQeA = NULL;
  EC_POINT                *pQeB = NULL;
  EC_POINT                *pQsB = NULL;
  EC_GROUP                *group = NULL;
  BIGNUM                  *bnTa;
  BIGNUM                  *bnDeA;
  BIGNUM                  *bnDsA;
  BIGNUM                  *bnXeA;       // x coordinate of ephemeral party A key
  BIGNUM                  *bnH;
  BIGNUM                  *bnN;
  BIGNUM                  *bnXeB;
  const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
  CRYPT_RESULT            retVal;

  pAssert(    curveData != NULL && outZ != NULL && dsA != NULL
             &&        deA != NULL &&  QsB != NULL && QeB != NULL);

  context = BN_CTX_new();
  if(context == NULL || curveData == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);
  BN_CTX_start(context);
  bnTa = BN_CTX_get(context);
  bnDeA = BN_CTX_get(context);
  bnDsA = BN_CTX_get(context);
  bnXeA = BN_CTX_get(context);
  bnH = BN_CTX_get(context);
  bnN = BN_CTX_get(context);
  bnXeB = BN_CTX_get(context);
  if(bnXeB == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);

  // Process:
  //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
  //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
  //  3. If P = O, output an error indicator.
  //  4. Z=xP, where xP is the x-coordinate of P.

  // Initialize group parameters and local values of input
  if((group = EccCurveInit(curveId, context)) == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

  if((pQeA = EC_POINT_new(group)) == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);

  BnFrom2B(bnDeA, &deA->b);
  BnFrom2B(bnDsA, &dsA->b);
  BnFrom2B(bnH, curveData->h);
  BnFrom2B(bnN, curveData->n);
```

---

Family "02" Published Page 465
Level 00 Revision 00.96 Copyright © TCG 2006-2013 March 15, 2013
BnFrom2B(bnXeB, &QeB->x.b);
pQeB = EccInitPoint2B(group, QeB, context);
pQsB = EccInitPoint2B(group, QsB, context);

// Compute the public ephemeral key pQeA = [de,A]G
if (retVal = PointMul(group, pQeA, bnDeA, NULL, NULL, context))
    != CRYPT_SUCCESS)
    goto Cleanup;

if(EC_POINT_get_affine_coordinates_GFp(group, pQeA, bnXeA, NULL, context) != 1)
    FAIL(FATAL_ERROR_INTERNAL);

// 1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
// tA := (ds,A + de,A • avf(Xe,A)) mod n (3)
// Compute 'tA' = ('deA' + 'dsA' * avf('XeA')) mod n
// Ta = avf(XeA);
BN_copy(bnTa, bnXeA);
avf1(bnTa, bnN);
if(!BN_mod_mul(bnTa, bnDsA, bnTa, bnN, context)
|| !BN_mod_add(bnTa, bnDeA, bnTa, bnN, context))
    FAIL(FATAL_ERROR_INTERNAL);

// 2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
// Put this in because almost every case of h is == 1 so skip the call when
// not necessary.
if(!BN_is_one(bnH))
{
    // Cofactor is not 1 so compute Ta := Ta * h mod n
    if(!BN_mul(bnTa, bnTa, bnH, context))
        FAIL(FATAL_ERROR_INTERNAL);

    // Now that 'tA' is (h * 'tA' mod n)
    // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
    // first, compute XeB = avf(XeB)
    avf1(bnXeB, bnN);

    // QsB := [XeB]QsB
    if(!EC_POINT_mul(group, pQsB, NULL, pQsB, bnXeB, context)
        || !EC_POINT_add(group, pQeB, pQeB, pQsB, context))
        FAIL(FATAL_ERROR_INTERNAL);

    // QeB := QsB + QeB

    if(PointMull(group, pQeB, NULL, pQeB, bnTa, context) == CRYPT_SUCCESS)
        // Convert BIGNUM E to TPM2B E
        Point2B(group, outZ, pQeB, (UINT16)BN_num_bytes(bnN), context);

Cleanup:
if(pQeA != NULL) EC_POINT_free(pQeA);
if(pQeB != NULL) EC_POINT_free(pQeB);
if(pQsB != NULL) EC_POINT_free(pQsB);
if(group != NULL) EC_GROUP_free(group);
BN_CTX_end(context);
BN_CTX_free(context);
return retVal;
B.10.4.1.24. avfSm2()

This function does the associated value computation required by SM2 key exchange. This is different form the avf() in the international standards because it returns a value that is half the size of the value returned by the standard avf. For example, if n is 15, Ws (w in the standard) is 2 but the W here is 1. This means that an input value of 14 (1110b) would return a value of 110b with the standard but 10b with the scheme in SM2.

```c
static BOOL
avfSm2(
    BIGNUM                  *bnX, // IN/OUT: the reduced value
    BIGNUM                  *bnN, // IN: the order of the curve
)
{
    // a) set w := ceil(ceil(log2(n)) / 2) - 1
    int w = ((BN_num_bits(bnN) + 1) / 2) - 1;
    // b) set x' := 2^w + ( x & (2^w - 1))
    // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
    BN_mask_bits(bnX, w); // as with avf1, this is too big by a factor of 2 but
    // it doesn't matter because we SET the extra bit anyway
    BN_set_bit(bnX, w);
    return TRUE;
}
```

SM2KeyExchange() This function performs the key exchange defined in SM2. The first step is to compute $tA = (dsA + deA \cdot avf(Xe,A)) \mod n$ Then, compute the Z value from $outZ = (h \cdot tA \mod n) (QsA + [avf(QeB(). x)](QeB()))$. The function will compute the ephemeral public key from the ephemeral private key. All points are required to be on the curve of $inQsA$. The function will fail catastrophically if this is not the case.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SUCCESS</td>
<td>results is valid</td>
</tr>
<tr>
<td>CRYPT_NO_RESULT</td>
<td>the value for $dsA$ does not give a valid point on the curve</td>
</tr>
</tbody>
</table>

```c
static CRYPT_RESULT
SM2KeyExchange(
    TPMS_ECC_POINT        *outZ, // OUT: the computed point
    TPM_ECC_CURVE          curveId, // IN: the curve for the computations
    TPM2B_ECC_PARAMETER   *dsA, // IN: static private TPM key
    TPM2B_ECC_PARAMETER   *deA, // IN: ephemeral private TPM key
    TPMS_ECC_POINT        *QsB, // IN: static public party B key
    TPMS_ECC_POINT        *QeB // IN: ephemeral public party B key
)
{
    BN_CTX                  *context;
    EC_POINT                *pQeA = NULL;
    EC_POINT                *pQeB = NULL;
    EC_POINT                *pQsB = NULL;
    EC_GROUP                *group = NULL;
    BIGNUM                  *bnTa;
    BIGNUM                  *bnDeA;
    BIGNUM                  *bnDsA;
    BIGNUM                  *bnXeA; // x coordinate of ephemeral party A key
    BIGNUM                  *bnH;
    BIGNUM                  *bnN;
```
BIGNUM          *bnXeB;
const ECC_CURVE_DATA *curveData = GetCurveData(curveId);
CRYPT_RESULT    retVal;
pAssert(    curveData != NULL && outZ != NULL && dsA != NULL
&& deA != NULL && QsB != NULL && QeB != NULL);

context = BN_CTX_new();
if( context == NULL || curveData == NULL)
FAIL(FATAL_ERROR_ALLOCATION);
BN_CTX_start(context);
bnTa = BN_CTX_get(context);
bnDeA = BN_CTX_get(context);
bnDsA = BN_CTX_get(context);
bnXeA = BN_CTX_get(context);
bnH = BN_CTX_get(context);
bnN = BN_CTX_get(context);
bnXeB = BN_CTX_get(context);
if( bnXeB == NULL)
FAIL(FATAL_ERROR_ALLOCATION);

context = BN_CTX_new();
if( context == NULL || curveData == NULL)
FAIL(FATAL_ERROR_ALLOCATION);

BN_CTX_start(context);
bnTa = BN_CTX_get(context);
bnDeA = BN_CTX_get(context);
bnDsA = BN_CTX_get(context);
bnXeA = BN_CTX_get(context);
bnH = BN_CTX_get(context);
bnN = BN_CTX_get(context);
bnXeB = BN_CTX_get(context);
if( bnXeB == NULL)
FAIL(FATAL_ERROR_ALLOCATION);

if( (group = EccCurveInit(curveId, context)) == NULL)
FAIL(FATAL_ERROR_INTERNAL);

if( (pQeA = EC_POINT_new(group)) == NULL)
FAIL(FATAL_ERROR_ALLOCATION);

BnFrom2B(bnDeA, &deA->b);
BnFrom2B(bnDsA, &dsA->b);
BnFrom2B(bnH, curveData->h);
BnFrom2B(bnN, curveData->n);
BnFrom2B(bnXeB, &QeB->x.b);
pQeB = EccInitPoint2B(group, QeB, context);
pQsB = EccInitPoint2B(group, QsB, context);

// Compute the public ephemeral key pQeA = [de,A]G
if( (retVal = PointMul(group, pQeA, bnDeA, NULL, NULL, context))
!=  CRYPT_SUCCESS)
goto Cleanup;

if( !BN_is_one(bnH))
{
// Cofactor is not 1 so compute Ta := Ta * h mod n
if( !BN_mul(bnTa, bnTa, bnH, context))
FAIL(FATAL_ERROR_INTERNAL);
}

// tA := (dsA + deA * avf(XeA)) mod n  (3)
// Compute 'tA' = ('dsA' +  'deA' • avf('XeA')) mod n
// Ta = avf(XeA);
BN_copy(bnTa, bnXeA);
avfSm2(bnTa, bnN);
if(!!! do Ta = deA * Ta mod n = deA * avf(XeA) mod n
!BN_mod_mul(bnTa, bnDeA, bnTa, bnN, context)
Ta = avf(XeA);
BN_copy(bnTa, bnXeA);
avfSm2(bnTa, bnN);
if(!!! do Ta = deA * Ta mod n = deA * avf(XeA) mod n
!BN_mod_mul(bnTa, bnDeA, bnTa, bnN, context)
Ta = avf(XeA);
BN_copy(bnTa, bnXeA);
}
// Now that 'tA' is (h * 'tA' mod n)
// 'outZ' = ['tA'](QsB + [avf(QeB.x)](QeB)).
// first, compute XeB = avf(XeB)
avfSm2(bnXeB, bnN);

// QeB := [XeB]QeB
if (!EC_POINT_mul(group, pQeB, NULL, pQeB, bnXeB, context)
    // QeB := QsB + QeB
    || !EC_POINT_add(group, pQeB, pQeB, pQsB, context)
)
    FAIL(FATAL_ERROR_INTERNAL);

// QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
if (PointMul(group, pQeB, NULL, pQeB, bnTa, context) == CRYPT_SUCCESS)
    // Convert BIGNUM E to TPM2B E
    Point2B(group, outZ, pQeB, (UINT16)BN_num_bytes(bnN), context);

Cleanup:
if (pQeA != NULL) EC_POINT_free(pQeA);
if (pQeB != NULL) EC_POINT_free(pQeB);
if (pQsB != NULL) EC_POINT_free(pQsB);
if (group != NULL) EC_GROUP_free(group);
BN_CTX_end(context);
BN_CTX_free(context);
return retVal;
}

static CRYPT_RESULT
C_2_2_ECDH(
    TPMS_ECC_POINT *outZ1,    // OUT: Zs
    TPMS_ECC_POINT *outZ2,    // OUT: Ze
    TPM_ECC_CURVE curveId,    // IN: the curve for the computations
    TPM2B_ECC_PARAMETER *dsA,    // IN: static private TPM key
    TPM2B_ECC_PARAMETER *deA,    // IN: ephemeral private TPM key
    TPMS_ECC_POINT *QsB,        // IN: static public party B key
    TPMS_ECC_POINT *QeB    // IN: ephemeral public party B key
)
{
    BN_CTX *context;
    EC_POINT *pQ = NULL;
    EC_GROUP *group = NULL;
    BIGNUM *bnD;
    UINT16 size;
    const ECC_CURVE_DATA *curveData = GetCurveData(curveId);

    context = BN_CTX_new();
    if (context == NULL || curveData == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    if ((bnD = BN_CTX_get(context)) == NULL)
        FAIL(FATAL_ERROR_INTERNAL);
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        // Get the static private key of A
1979        BnFrom2B(bnD, &dsA->b);
1980        // Initialize the static public point from B
1981        pQ = EccInitPoint2B(group, QsB, context);
1982        // Do the point multiply for the Zs value
1983        if(PointMul(group, pQ, NULL, pQ, bnD, context) != CRYPT_NO_RESULT)
1984            // Convert the Zs value
1985            Point2B(group, outZ1, pQ, size, context);
1986        // Get the ephemeral private key of A
1987        BnFrom2B(bnD, &deA->b);
1988        // Initialize the ephemeral public point from B
1989        PointFrom2B(group, pQ, QeB, context);
1990        // Do the point multiply for the Ze value
1991        if(PointMul(group, pQ, NULL, pQ, bnD, context) != CRYPT_NO_RESULT)
1992            // Convert the Ze value.
1993            Point2B(group, outZ2, pQ, size, context);
1994        if(pQ != NULL) EC_POINT_free(pQ);
1995        if(group != NULL) EC_GROUP_free(group);
1996        BN_CTX_end(context);
1997        BN_CTX_free(context);
1998        return CRYPT_SUCCESS;
1999    }
2000
2001    B.10.4.1.26. _cpri__C_2_2_KeyExchange()
2002
2003    This function is the dispatch routine for the EC key exchange functions that use two ephemeral and two
2004    static keys.
2005
2006
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPT_SCHEME</td>
<td>scheme is not defined</td>
</tr>
</tbody>
</table>
2007
2008    CRYPT_RESULT
2009    _cpri__C_2_2_KeyExchange(  
2010    TPMS_ECC_POINT *outZ1,    // OUT: a computed computed point
2011    TPMS_ECC_POINT *outZ2,    // OUT: and optional second point
2012    TPM_ECC_CURVE curveId,   // IN: the curve for the computations
2013    TPM_ALG_ID scheme,       // IN: the key exchange scheme
2014    TPM2B_ECC_PARAMETER *dsA, // IN: static private TPM key
2015    TPM2B_ECC_PARAMETER *deA, // IN: ephemeral private TPM key
2016    TPM_ECC_POINT *QsB,      // IN: static public party B key
2017    TPM_ECC_POINT *QeB       // IN: ephemeral public party B key
2018    )
2019    {                  
2020    pAssert(outZ1 != NULL    
2021        && dsA != NULL && deA != NULL    
2022        && QsB != NULL && QeB != NULL);
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)
outZ2->x.b.size = 0;
outZ2->y.b.size = 0;
}

switch (scheme)
{
    case TPM_ALG_ECDH:
        return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
        break;
#ifdef TPM_ALG_ECMQV
    case TPM_ALG_ECMQV:
        return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
        break;
#endif
#ifdef TPM_ALG_SM2
    case TPM_ALG_SM2:
        return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
        break;
#endif
    default:
        return CRYPTO_SCHEME;
    
}

// 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.

CRYPT_RESULT
__cpri__C_2_2_KeyExchange()
{
    return CRYPTO_FAIL;
}
#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

```c
#include "PlatformData.h"
```

C.2.3. Functions

C.2.3.1. _plat__IsCanceled()

Check if the cancel flag is set

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if cancel flag is set</td>
</tr>
<tr>
<td>FALSE</td>
<td>if cancel flag is not set</td>
</tr>
</tbody>
</table>

```c
BOOL _plat__IsCanceled(void)
{
    // return cancel flag
    return s_isCanceled;
}
```

C.2.3.2. _plat__SetCancel()

Set cancel flag.

```c
void _plat__SetCancel(void)
{
    s_isCanceled = TRUE;
    return;
}
```

C.2.3.3. _plat__ClearCancel()

Clear cancel flag

```c
void _plat__ClearCancel(void)
{
    s_isCanceled = FALSE;
    return;
}
```
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

```c
#include <time.h>
#include <assert.h>
#include "PlatformData.h"
#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

```c
void _plat__ClockReset(void)
{
    // Implementation specific: Microsoft C set CLOCKS_PER_SEC to be 1/1000,
    // so here the measurement of clock() is in millisecond.
    s_initClock = clock();
    s_adjustRate = CLOCK_NOMINAL;
    return;
}
```

C.3.3.2. _plat__ClockTimeFromStart()

Function returns the compensated time from the start of the command when _plat__ClockTimeFromStart() was called.

```c
unsigned long long _plat__ClockTimeFromStart()
{
    unsigned long long currentClock = clock();
    return ((currentClock - s_initClock) * CLOCK_NOMINAL) / s_adjustRate;
}
```

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

```c
unsigned long long _plat__ClockTimeElapsed(void)
{
```
unsigned long long elapsed;
unsigned long long currentClock = clock();
elapsed = ((currentClock - s_initClock) * CLOCK_NOMINAL) / s_adjustRate;
s_initClock += (elapsed * s_adjustRate) / CLOCK_NOMINAL;

#ifdef DEBUGGING_TIME
    // Put this in so that TPM time will pass much faster than real time when
    // doing debug.
    // A value of 1000 for DEBUG_TIME_MULTIPLIER will make each ms into a second
    // A good value might be 100
    elapsed *= DEBUG_TIME_MULTIPLIER
#endif
    return elapsed;
}

C.3.3.4. _plat__ClockAdjustRate()

Adjust the clock rate

void _plat__ClockAdjustRate(
    int adjust  // IN: the adjust number. It could be
    // positive or negative
)
{
    // We expect the caller should only use a fixed set of constant values to
    // adjust the rate
    switch(adjust)
    {
        case CLOCK_ADJUST_COARSE:
            s_adjustRate += CLOCK_ADJUST_COARSE;
            break;
        case -CLOCK_ADJUST_COARSE:
            s_adjustRate -= CLOCK_ADJUST_COARSE;
            break;
        case CLOCK_ADJUST_MEDIUM:
            s_adjustRate += CLOCK_ADJUST_MEDIUM;
            break;
        case -CLOCK_ADJUST_MEDIUM:
            s_adjustRate -= CLOCK_ADJUST_MEDIUM;
            break;
        case CLOCK_ADJUST_FINE:
            s_adjustRate += CLOCK_ADJUST_FINE;
            break;
        case -CLOCK_ADJUST_FINE:
            s_adjustRate -= CLOCK_ADJUST_FINE;
            break;
        default:
            assert(FALSE);
            break;
    }

    if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
        s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
    if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
        s_adjustRate = CLOCK_NOMINAL-CLOCK_ADJUST_LIMIT;
    return;
C.4  Entropy.c

C.4.1. Includes

```c
#define  _CRT_RAND_S
#include  <stdlib.h>
#include  <assert.h>
#include  <memory.h>
#include  "bool.h"
#include  "Platform.h"
```

C.4.2. _plat__GetEntropy()

```c
unsigned int  
_plat__GetEntropy ( 
    unsigned char*  EntropyBuffer,  
    unsigned int  EntropySize  // Assumption: EntropyBuffer is big enough to  
    // receive it, we don't do any checks for that  
)
{
    unsigned int  rndNum;
    unsigned int  i;
    errno_t  err;
    // Use h/w random number generator to build entropy for crypto PRNG.
    for ( i = 0; i < EntropySize/<K>sizeof(unsigned int); i++)
    {
        err = rand_s(&rndNum);
        assert(err == 0);
        memcpy((char*)EntropyBuffer+i*sizeof(unsigned int),
               (char*)&rndNum,
               sizeof(unsigned int));
    }
    return  EntropySize;
    }
```
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

```
#include <assert.h>
#include <memory.h>
#include <string.h>
#include "PlatformData.h"
```

C.6.3. Functions

C.6.3.1. _plat__NVEnable()

Enable NV memory

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if success</td>
</tr>
<tr>
<td>non-0</td>
<td>if fail</td>
</tr>
</tbody>
</table>

```
int _plat__NVEnable(
    void *platParameter  
    // IN: platform specific parameters
) {
    platParameter = 0;  // to try to satisfy the compiler and remove warning

    #ifdef FILE_BACKED_NV
        if(s_NVFile != NULL) return 0;
        // Try to open an exist NVChip file for read/write
        if(0 != fopen_s(&s_NVFile, "NVChip", "r+b"))
            s_NVFile = NULL;

        if(NULL != s_NVFile)
            { // See if the NVChip file is empty
                fseek(s_NVFile, 0, SEEK_END);
                if(0 == ftell(s_NVFile))
                    s_NVFile = NULL;
                }
        if(s_NVFile == NULL)
            { // Initialize all the byte in the new file to 0
                memset(s_NV, 0, NV_MEMORY_SIZE);
                // If NVChip file does not exist, try to create it for read/write
                fopen_s(&s_NVFile, "NVChip", "w+b");
                // Start initialize at the end of new file
                fseek(s_NVFile, 0, SEEK_END);
                // Write 0s to NVChip file
                fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);
```

39     }  
40   } else { 
41     // If NVChip file exist, assume the size is correct 
42     fseek(s_NVFile, 0, SEEK_END); 
43     assert(ftell(s_NVFile) == NV_MEMORY_SIZE); 
44     // read NV file data to memory 
45     fseek(s_NVFile, 0, SEEK_SET); 
46     fread(s_NV, NV_MEMORY_SIZE, 1, s_NVFile); 
47   } } 
49 #endif 
50 
51 return 0; 
52 } 

C.6.3.2. _plat__NVDisable()

Disable NV memory

53 void _plat__NVDisable(void) 
54 { 
55   #ifdef FILE_BACKED_NV 
56     assert(s_NVFile != NULL); 
57     // Close NV file 
58     fclose(s_NVFile); 
59     // Set file handle to NULL 
60     s_NVFile = NULL; 
61 #endif 
62   
63   return; 
64 } 

C.6.3.3. _plat__IsNvAvailable()

Check if NV is available

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NV is available</td>
</tr>
<tr>
<td>1</td>
<td>NV is not available due to write failure</td>
</tr>
<tr>
<td>2</td>
<td>NV is not available due to rate limit</td>
</tr>
</tbody>
</table>

68 int _plat__IsNvAvailable(void) 
69 { 
70   if(s_NvIsAvailable == FALSE) 
71     return 1; 
72 
73 #ifdef FILE_BACKED_NV 
74   if(s_NVFile == NULL) 
75     return 1; 
76 #endif 
77 
78 return 0; 
79 }
C.6.3.4. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```c
void
_plat__NvMemoryRead(
    unsigned int startOffset, // IN: read start
    unsigned int size, // IN: size of bytes to read
    void *data // OUT: data buffer
)
{
    assert(startOffset + size <= NV_MEMORY_SIZE);
    // Copy data from RAM
    memcpy(data, &s_NV[startOffset], size);
    return;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>the NV location is different from the test value</td>
</tr>
<tr>
<td>FALSE</td>
<td>the NV location is the same as the test value</td>
</tr>
</tbody>
</table>

```c
BOOL
_plat__NvIsDifferent(
    unsigned int startOffset, // IN: read start
    unsigned int size, // IN: size of bytes to read
    void *data // IN: data buffer
)
{
    return (memcmp(&s_NV[startOffset], data, size) != 0);
}
```

C.6.3.6. _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.

```c
void
_plat__NvMemoryWrite(
    unsigned int startOffset, // IN: write start
    unsigned int size, // IN: size of bytes to write
    void *data // OUT: data buffer
)
{
    assert(startOffset + size <= NV_MEMORY_SIZE);
    // Copy the data to the NV image
    memcpy(&s_NV[startOffset], data, size);
}
```
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

```c
void _plat__NvMemoryMove(
    unsigned int sourceOffset, // IN: source offset
    unsigned int destOffset,   // IN: destination offset
    unsigned int size         // IN: size of data being moved
)
{
    assert(sourceOffset + size <= NV_MEMORY_SIZE);
    assert(destOffset + size <= NV_MEMORY_SIZE);
    // Move data in RAM
    memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);
    return;
}
```

C.6.3.8. _plat__NvCommit()

Update NV chip

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NV write success</td>
</tr>
<tr>
<td>non-0</td>
<td>NV write fail</td>
</tr>
</tbody>
</table>

```c
int _plat__NvCommit(void)
{
    #ifdef FILE_BACKED_NV
        // If NV file is not available, return failure
        if(s_NVFile == NULL || s_NvIsAvailable == FALSE)
            return 1;
    #else
        // Write RAM data to NV
        fseek(s_NVFile, 0, SEEK_SET);
        fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);
        return 0;
    #endif
        return 0;
}
```

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

```c
void _plat__SetNvAvail(void)
{
    s_NvIsAvailable = TRUE;
    return;
}
```
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.

```c
void _plat__ClearNvAvail(void)
{
    s_NvIsAvailable = FALSE;
    return;
}
```
C.7  PowerPlat.c

C.7.1.  Includes and Function Prototypes

```c
#include <assert.h>
#include "PlatformData.h"
#include "Platform.h"
```

Platform power on and off functions

C.7.2.  Functions

C.7.2.1.  _plat__Signal_PowerOn()

Signal platform power on

```c
void _plat__Signal_PowerOn(void)
{
    // Start clock
    _plat__ClockReset();
    // Prepare NV memory for power on
    _plat__NVEnable(0);
    return;
}
```

C.7.2.2.  _plat__Signal_PowerOff()

Signal platform power off

```c
void _plat__Signal_PowerOff(void)
{
    // Prepare NV memory for power off
    _plat__NVDisable();
    return;
}
```
C.8 Platform.h

```c
#ifndef PLATFORM_H
#define PLATFORM_H

C.8.1. Includes

#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

void _plat__Signal_PowerOn (void);

C.8.2.2. _plat__Signal_PowerOff()

Signal power off This signal is simulate by a RPC call

void _plat__Signal_PowerOff (void);

C.8.3. Physical Presence Functions

C.8.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if physical presence is signaled</td>
</tr>
<tr>
<td>FALSE</td>
<td>if physical presence is not signaled</td>
</tr>
</tbody>
</table>

BOOL _plat__PhysicalPresenceAsserted (void);

C.8.3.2. _plat__Signal_PhysicalPresenceOn

Signal physical presence on This signal is simulate by a RPC call

void _plat__Signal_PhysicalPresenceOn (void);

C.8.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off This signal is simulate by a RPC call

void _plat__Signal_PhysicalPresenceOff (void);
C.8.4. Command Canceling Functions

C.8.4.1. _plat__IsCanceled()

Check if the cancel flag is set

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if cancel flag is set</td>
</tr>
<tr>
<td>FALSE</td>
<td>if cancel flag is not set</td>
</tr>
</tbody>
</table>

    BOOL _plat__IsCanceled(void);

C.8.4.2. _plat__SetCancel()

Set cancel flag.

    void _plat__SetCancel(void);

C.8.4.3. _plat__ClearCancel()

Clear cancel flag

    void _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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if success</td>
</tr>
<tr>
<td>non-0</td>
<td>if fail</td>
</tr>
</tbody>
</table>

    int _plat__NVEnable(  // IN: platform specific parameters
        void *platParameter
    );

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

    void _plat__NVDisable(void);
C.8.5.3. _plat__IsNvAvailable()

Check if NV is available

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NV is available</td>
</tr>
<tr>
<td>1</td>
<td>NV is not available due to write failure</td>
</tr>
<tr>
<td>2</td>
<td>NV is not available due to rate limit</td>
</tr>
</tbody>
</table>

```
26 int _plat__IsNvAvailable(void);
```

C.8.5.4. _plat__NvCommit()

Update NV chip

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NV write success</td>
</tr>
<tr>
<td>non-0</td>
<td>NV write fail</td>
</tr>
</tbody>
</table>

```
28 int _plat__NvCommit(void);
```

C.8.5.5. _plat__NvMemoryRead()

Read a chunk of NV memory

```
30 void _plat__NvMemoryRead(
31     unsigned int startOffset,  // IN: read start
32     unsigned int size,        // IN: size of bytes to read
33     void *data                // OUT: data buffer
34 );
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>the NV location is different from the test value</td>
</tr>
<tr>
<td>FALSE</td>
<td>the NV location is the same as the test value</td>
</tr>
</tbody>
</table>

```
36 BOOL _plat__NvIsDifferent(
37     unsigned int startOffset,  // IN: read start
38     unsigned int size,        // IN: size of bytes to compare
39     void *data                // IN: data buffer
40 );
```

C.8.5.7. _plat__NvMemoryWrite()

Write a chunk of NV memory
void __plat__NvMemoryWrite(
    unsigned int startOffset, // IN: read start
    unsigned int size, // IN: size of bytes to read
    void *data // OUT: data buffer
);

C.8.5.8. __plat__NvMemoryMove()

Move a chunk of NV memory from source to destination. This function should ensure that if there is overlap, the original data is copied before it is written.

void __plat__NvMemoryMove(
    unsigned int sourceOffset, // IN: source offset
    unsigned int destOffset, // IN: destination offset
    unsigned int size // IN: size of data being moved
);

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.

void __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.

void __plat__ClearNvAvail(void);

C.8.6. Locality Functions

C.8.6.1. __plat__LocalityGet()

Get the most recent command locality in locality value form

unsigned char __plat__LocalityGet(void);

C.8.6.2. __plat__LocalitySet()

Set the most recent command locality in locality value form

void __plat__LocalitySet(
    unsigned char locality
);
C.8.7. Clock Constants and Functions

Assume that the nominal divisor is 30000

```c
#define CLOCK_NOMINAL 30000
```

A 1% change in rate is 300 counts

```c
#define CLOCK_ADJUST_COARSE 300
```

A .1 change in rate is 30 counts

```c
#define CLOCK_ADJUST_MEDIUM 30
```

A minimum change in rate is 1 count

```c
#define CLOCK_ADJUST_FINE 1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

```c
#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

```c
void _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.

```c
unsigned long long _plat__ClockTimeFromStart(void);
```

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

```c
unsigned long long _plat__ClockTimeElapsed(void);
```

C.8.7.4. _plat__ClockAdjustRate()

Adjust the clock rate

```c
void _plat__ClockAdjustRate(int adjust // IN: the adjust number. It could be
```

Family "02" Published Page 489
Level 00 Revision 00.96 Copyright © TCG 2006-2013 March 15, 2013
C.8.8. Entropy Constants and Functions

C.8.8.1. _plat_GetEntropy

Returns the number of bytes of entropy generated

```c
unsigned int
_plat_GetEntropy ( unsigned char* EntropyBuffer, // IN/OUT: Buffer to receive the entropy.
                    unsigned int EntropySize    // IN: amount of entropy to generate. We
                    );                          //     assume that EntropyBuffer is big enough
                                                            //     to receive it.
```

C.8.9. Failure Mode

C.8.9.1. _plat_TpmFail

Put TPM to failure mode

```c
int
_plat_TpmFail(const char *function, int line, int code);
```
```c
#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.

```c
#ifndef PLATFORM_DATA_H
#define PLATFORM_DATA_H
#include "Implementation.h"
#include "bool.h"

extern BOOL s_isCanceled;

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

extern unsigned long long s_initClock;
extern unsigned int s_adjustRate;

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

extern unsigned char s_locality;

From LocalityPlat.c Locality of current command

#define FILE_BACKED_NV
#if defined FILE_BACKED_NV
#include <stdio.h>

A file to emulate NV storage

extern FILE* s_NVFile;
#endif

extern unsigned char s_NV[NV_MEMORY_SIZE];

extern BOOL s_NvIsAvailable;

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.

From PPPlat.c Physical presence. It is initialized to FALSE

extern BOOL s_physicalPresence;
#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

```c
#include "PlatformData.h"
```

C.11.3. Functions

C.11.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>if physical presence is signaled</td>
</tr>
<tr>
<td>FALSE</td>
<td>if physical presence is not signaled</td>
</tr>
</tbody>
</table>

```c
BOOL _plat__PhysicalPresenceAsserted(void)
{
    // Do not know how to check physical presence without real hardware.
    // so always return TRUE;
    return s_physicalPresence;
}
```

C.11.3.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```c
void _plat__Signal_PhysicalPresenceOn(void)
{
    s_physicalPresence = TRUE;
    return;
}
```

C.11.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```c
void _plat__Signal_PhysicalPresenceOff( void)
{
    s_physicalPresence = FALSE;
    return;
}
```
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.

```c
#include "assert.h"

C.12.2. _plat__TpmFail()

int _plat__TpmFail(const char *function, int line, int code)
{
    // These lines are added to keep the compiler from complaining about no reference to
    // the formal parameter
    char a = *function;
    int l = line;
    int c = code;
    a += 1;
    l += 1;
    c += 1;
    // LAST JUNK LINE
    assert(0);
    return 0;
}
```
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

```
#ifndef TCP_TPM_PROTOCOL_H
#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 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 UINT32=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

```
#define TPM_SIGNAL_POWER_ON         1
#define TPM_SIGNAL_POWER_OFF        2
#define TPM_SIGNAL_PHYS_PRES_ON     3
#define TPM_SIGNAL_PHYS_PRES_OFF    4
#define TPM_SIGNAL_HASH_START       5
#define TPM_SIGNAL_HASH_DATA        6
// [UINT32 BufferSize, BYTE[BufferSize] Buffer]
#define TPM_SIGNAL_HASH_END         7
#define TPM_SEND_COMMAND            8
// [BYTE Locality, UINT32 InBufferSize, BYTE[InBufferSize] InBuffer] ->
#define TPM_SIGNAL_CANCEL_ON        9
#define TPM_SIGNAL_CANCEL_OFF       10
#define TPM_SIGNAL_NV_ON            11
#define TPM_SIGNAL_NV_OFF           12
#define TPM_REMOTE_HANDSHAKE        15
#define TPM_SET_ALTERNATIVE_RESULT  16
#define TPM_SHUTDOWN                20

enum TpmEndPointInfo
{
    tpmPlatformAvailable = 0x01,
    tpmUsesTbs = 0x02,
    tpmInRawMode = 0x04,
    tpmSupportsPP = 0x08
};

// Existing RPC interface type definitions retained so that the implementation can be re-used
typedef struct
{
    unsigned long BufferSize;
    unsigned char *Buffer;
} _IN_BUFFER;
typedef unsigned char *_OUTPUT_BUFFER;
typedef struct
{
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.
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

```c
#include <stdio.h>
#include <windows.h>
#include <winsock.h>
#include "string.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "TpmTcpProtocol.h"

BOOL ReadBytes(SOCKET s, char* buffer, int NumBytes);
BOOL WriteBytes(SOCKET s, char* buffer, int NumBytes);
BOOL WriteUINT32(SOCKET s, UINT32 val);
static UINT32 ServerVersion = 1;

#define MAX_BUFFER 1048576
char InputBuffer[MAX_BUFFER];
char OutputBuffer[MAX_BUFFER];
```

The input and output data buffers for the simulator.

D.3.3. Functions

D.3.3.1. CreateSocket()

Function creates a socket listening on PortNumber.

```c
static int CreateSocket(int PortNumber, SOCKET *listenSocket)
{
    WSADATA wsaData;
    struct sockaddr_in MyAddress;
    int res;
    // Initialize Winsock
    res = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (res != 0)
    {
        printf("WSAStartup failed with error: %d\n", res);
        return -1;
    }
    // create listening socket
    *listenSocket = socket(PF_INET, SOCK_STREAM, 0);
    if(INVALID_SOCKET == *listenSocket)
    {
        printf("Cannot create server listen socket.  Error is 0x%x\n",
                WSAGetLastError());
        return -1;
    }
    // bind the listening socket to the specified port
    ZeroMemory(&MyAddress, sizeof(MyAddress));
    MyAddress.sin_port=htons((short)PortNumber);
```

D.3.3.2. PlatformServer()

This function processes incoming platform requests.

```c
BOOL PlatformServer(SOCKET s)
{
    UINT32 Command;
    BOOL ok;
    for(;;)
    {
        ok = ReadBytes(s, (char*) &Command, 4);
        // client disconnected (or other error). We stop processing this client
        // and return to our caller who can stop the server or listen for another
        // connection.
        if(!ok) return TRUE;
        Command = ntohl(Command);
        switch(Command)
        {
            case TPM_SIGNAL_POWER_ON:
                _rpc__Signal_PowerOn();
                break;
            case TPM_SIGNAL_POWER_OFF:
                _rpc__Signal_PowerOff();
                break;
            case TPM_SIGNAL_PHYS_PRES_ON:
                _rpc__Signal_PhysicalPresenceOn();
                break;
            case TPM_SIGNAL_PHYS_PRES_OFF:
                _rpc__Signal_PhysicalPresenceOff();
                break;
            case TPM_SIGNAL_CANCEL_ON:
                _rpc__Signal_CanCelOn();
                break;
            case TPM_SIGNAL_CANCEL_OFF:
                _rpc__Signal_CanCelOff();
                break;
            case TPM_SIGNAL_NV_ON:
                break;
            case TPM_SIGNAL_NV_OFF:
                break;
            default:
                break;
        }
    }
}
```
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    return FALSE;
119
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);
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.

```c
int PlatformSignalService(int PortNumber)
{
    HANDLE hPlatformSvc;
    int ThreadId;
    int port = PortNumber;
    // Create service thread for platform signals
    hPlatformSvc = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
        (LPVOID) (INT_PTR) port, 0, (LPDWORD)&ThreadId);
    if(hPlatformSvc == NULL)
    {
        printf("Thread Creation failed\n");
        return -1;
    }
    return 0;
}
```

D.3.3.5. RegularCommandService()

```c
int RegularCommandService(int PortNumber)
{
    SOCKET listenSocket;
    SOCKET serverSocket;
    struct sockaddr_in HerAddress;
    int res, length;
    BOOL continueServing;
    res = CreateSocket(PortNumber, &listenSocket);
    if(res != 0)
    {
        printf("Create platform service socket fail\n");
        return res;
    }
    // Loop accepting connections one-by-one until we are killed or asked to stop
    // Note the TPM command service is single-threaded so we don't listen for
    // a new connection until the prior connection drops.
    do
    {
        printf("TPM command server listening on port %d\n", PortNumber);
        // blocking accept
        length = sizeof(HerAddress);
        serverSocket = accept(listenSocket,
            (struct sockaddr*) &HerAddress,
            &length);
        if(serverSocket == SOCKET_ERROR)
        {
            printf("Accept error. Error is 0x%x\n", WSAGetLastError());
            return -1;
        }
    } while(continueServing);
```
printf("Client accepted\n");

// normal behavior on client disconnection is to wait for a new client to connect
continueServing = TpmServer(serverSocket);
closesocket(serverSocket);
}
}
while(continueServing);
return 0;
}

D.3.3.6. StartTcpServer()

Main entry-point. The server listens on port specified. Note that there is no way to specify the network interface in this implementation.

int StartTcpServer(int PortNumber)
{
    int res;
    // Start Platform Signal Processing Service
    res = PlatformSignalService(PortNumber+1);
    if (res != 0)
    {
        printf("PlatformSignalService failed\n");
        return res;
    }
    // Start Regular/DRTM TPM command service
    res = RegularCommandService(PortNumber);
    if (res != 0)
    {
        printf("RegularCommandService failed\n");
        return res;
    }
    return 0;
}

D.3.3.7. ReadBytes()

Read NumBytes() into buffer on indicated socket.

BOOL ReadBytes(SOCKET s, char* buffer, int NumBytes)
{
    int res;
    int numGot = 0;
    while(numGot<NumBytes)
    {
        res = recv(s, buffer+numGot, NumBytes-numGot, 0);
        if(res == -1)
        {
            printf("Receive error. Error is 0x%x\n", WSAGetLastError());
            return FALSE;
        }
        if(res==0)
        {
            return FALSE;
        }
        numGot+=res;
    }
    return TRUE;
D.3.3.8. WriteBytes()

Send NumBytes() on indicated socket

```c
BOOL WriteBytes(SOCKET s, char* buffer, int NumBytes)
{
    int res;
    int numSent = 0;
    while(numSent<NumBytes)
    {
        res = send(s, buffer+numSent, NumBytes-numSent, 0);
        if(res == -1)
        {
            if(WSAGetLastError() == 0x2745)
            {
                printf("Client disconnected\n");
            }
            else
            {
                printf("Send error.  Error is 0x%x\n", WSAGetLastError());
            }
            return FALSE;
        }
        numSent+=res;
    }
    return TRUE;
}
```

D.3.3.9. WriteUINT32()

Send one byte containing htonl(1)

```c
BOOL WriteUINT32(SOCKET s, UINT32 val)
{
    UINT32 netVal = htonl(val);
    return WriteBytes(s, (char*) &netVal, 4);
}
```

D.3.3.10. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order

```c
BOOL ReadVarBytes(SOCKET s, char* buffer, UINT32* BytesReceived, int MaxLen)
{
    int length;
    BOOL res;
    res = ReadBytes(s, (char*) &length, 4);
    if(!res) return res;
    length = ntohl(length);
    *BytesReceived = length;
    if(length>MaxLen)
    {
        printf("Buffer too big.  Client says %d\n", length);
        return FALSE;
    }
    if(length==0) return TRUE;
    res = ReadBytes(s, buffer, length);
    if(!res) return res;
    return TRUE;
}
D.3.3.11. WriteVarBytes()

Send a UINT32-length-prefixed binary array. Note that the 4-byte length is in network byte order.

```c
BOOL WriteVarBytes(SOCKET s, char* buffer, int BytesToSend)
{
    UINT32 netLength = htonl(BytesToSend);
    BOOL res;
    res = WriteBytes(s, (char*) &netLength, 4);
    if(!res) return res;
    res = WriteBytes(s, buffer, BytesToSend);
    if(!res) return res;
    return TRUE;
}
```

D.3.3.12. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```c
BOOL TpmServer(SOCKET s)
{
    UINT32 length;
    UINT32 Command;
    BYTE locality;
    BOOL ok;
    BOOL WasDebugCommand=FALSE;
    int result;
    int clientVersion;
    _IN_BUFFER InBuffer;
    _OUT_BUFFER OutBuffer;
    for(;;)
    {
        ok = ReadBytes(s, (char*) &Command, 4);
        // client disconnected (or other error). We stop processing this client
        // and return to our caller who can stop the server or listen for another
        // connection.
        if(!ok) return TRUE;
        Command = ntohl(Command);
        switch(Command)
        {
        case TPM_SIGNAL_HASH_START:
            _rpc__Signal_Hash_Start();
            WriteUINT32(s,0);
            break;
        case TPM_SIGNAL_HASH_END:
            _rpc__Signal_HashEnd();
            WriteUINT32(s,0);
            break;
        case TPM_SIGNAL_HASH_DATA:
            ok = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
            if('!ok) return TRUE;
            InBuffer.Buffer = (BYTE*) InputBuffer;
            InBuffer.BufferSize = length;
            _rpc__Signal_Hash_Data(InBuffer);
            WriteUINT32(s,0);
            break;
        ```
case TPM_SEND_COMMAND:
    ok = ReadBytes(s, (char*) &locality, 1);
    if(!ok) return TRUE;

    ok = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
    if(!ok) return TRUE;
    InBuffer.Buffer = (BYTE*) InputBuffer;
    InBuffer.BufferSize = length;
    OutBuffer.BufferSize = MAX_BUFFER;
    OutBuffer.Buffer = (_OUTPUT_BUFFER) OutputBuffer;
    _rpc__Send_Command(locality, InBuffer, &OutBuffer);
    ok = WriteVarBytes(s, (char*) OutBuffer.Buffer, OutBuffer.BufferSize);
    if(!ok) return TRUE;
    ok = WriteUINT32(s,0);
    if(!ok) return TRUE;
    break;

case TPM_REMOTE_HANDSHAKE:
    ok = ReadBytes(s, (char*) &clientVersion, 4);
    if(!ok) return TRUE;
    if( clientVersion == 0 )
    {
        printf("Unsupported client version (0).\n");
        return TRUE;
    }
    ok &= WriteUINT32(s, ServerVersion);
    ok &= WriteUINT32(s, tpmInRawMode | tpmPlatformAvailable | tpmSupportsPP);
    if(!ok) return TRUE;
    break;

case TPM_SET_ALTERNATIVE_RESULT:
    ok = ReadBytes(s, (char*) &result, 4);
    if(!ok) return TRUE;
    ok = WriteUINT32(s,0);
    if(!ok) return TRUE;
    // Alternative result is not applicable to the simulator.
    break;

case TPM_SHUTDOWN:
    // Client signaled end-of-session
    return TRUE;

default:
    printf("Unrecognized TPM interface command. Client says %d\n", Command);
    return TRUE;
    }
    return FALSE;
D.4 TPMCmdp.c

D.4.1. Description

This file contains the functions that process the commands received on the control port or the command port of the simulator. The control port is used to allow simulation of hardware events (such as, _TPM_Hash_Start()) to test the simulated TPM's reaction to those events. This improves code coverage of the testing.

D.4.2. Includes and Data Definitions

```c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include<stdio.h>
#include<TPMLib.h>
#include<platform.h>
#include<windows.h>
#include "TpmTcpProtocol.h"
static BOOL     s_isPowerOn = FALSE;
```

D.4.3. Functions

D.4.3.1. Signal_PowerOn()

Signal a power on event.

```c
void _rpc__Signal_PowerOn()
{
    if(s_isPowerOn) return;
    // Pass power on signal to platform
    _plat__Signal_PowerOn();
    // Pass power on signal to TPM
    _TPM_Init();
    // Set state as power on
    s_isPowerOn = TRUE;
}
```

D.4.3.2. Signal_PowerOff()

Signal a power off event.

```c
void _rpc__Signal_PowerOff()
{
    if(!s_isPowerOn) return;
    // Pass power off signal to platform
    _plat__Signal_PowerOff();
    s_isPowerOn = FALSE;
    return;
}
```
D.4.3.3.  _rpc__Signal_PhysicalPresenceOn()

Function to simulate activation of the physical presence pin.

```c
void _rpc__Signal_PhysicalPresenceOn()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;

    // Pass physical presence on to platform
    _plat__Signal_PhysicalPresenceOn();
    return;
}
```

D.4.3.4.  _rpc__Signal_PhysicalPresenceOff()

Function to simulate deactivation of the physical presence pin.

```c
void _rpc__Signal_PhysicalPresenceOff()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;

    // Pass physical presence off to platform
    _plat__Signal_PhysicalPresenceOff();
    return;
}
```

D.4.3.5.  _rpc__Signal_Hash_Start()

Function to simulate a _TPM_Hash_Start() event.

```c
void _rpc__Signal_Hash_Start()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;

    // Pass _TPM_Hash_Start signal to TPM
    Signal_Hash_Start();
    return;
}
```

D.4.3.6.  _rpc__Signal_Hash_Data()

Function to simulate a _TPM_Hash_Data() event.

```c
void _rpc__Signal_Hash_Data(
    _IN_BUFFER input)
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;

    // Pass _TPM_Hash_Data signal to TPM
    Signal_Hash_Data(input.BufferSize, input.Buffer);
}
D.4.3.7. _rpc__Signal_HashEnd()

Function to simulate a _TPM_Hash_End() event.

```c
void _rpc__Signal_HashEnd()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;

    // Pass TPM HashEnd signal to TPM
    Signal_Hash_End();

    return;
}
```

D.4.3.8. _rpc__Send_Command()

This is the TPM command interface.

```c
void _rpc__Send_Command( unsigned char locality,
            _IN_BUFFER request,
            _OUT_BUFFER *response )
{
    // If TPM is power off, reject any commands.
    if(!s_isPowerOn)
    {
        response->BufferSize = 0;
        return;
    }

    // Set command locality. Command locality is a signal rather than a part
    // of TPM internal state. So we always set the locality information even
    // the command may fail
    _plat__LocalitySet(locality);

    // Call command execution
    // response buffer space is provided by the called function.
    ExecuteCommand(request.BufferSize, request.Buffer,
                    &response->BufferSize, &response->Buffer);

    if(response->BufferSize == 10 && response->Buffer[9] != 0)
        response->Buffer[6] = 0;

    return;
}
```

D.4.3.9. _rpc__Signal_CancelOn()

Function to turn on the indication to cancel a command in process.

```c
void _rpc__Signal_CancelOn()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;
}
D.4.3.10. _rpc__Signal_CancelOff()

Function to turn off the indication to cancel a command in process.

```c
void _rpc__Signal_CancelOff()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;
    // Set the platform canceling flag.
    _plat__SetCancel();
    return;
}
```

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.

```c
void _rpc__Signal_NvOn()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;
    _plat__SetNvAvail();
    return;
}
```

D.4.3.12. _rpc__Signal_NvOff()

This function set the indication that NV memory is no longer available.

```c
void _rpc__Signal_NvOff()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;
    _plat__ClearNvAvail();
    return;
}
```

D.4.3.13. _rpc__Shutdown()

This function is used to stop the TPM simulator.

```c
void _rpc__Shutdown()
{
    RPC_STATUS status;
```
// Stop TPM
TPM_TearDown();

status = RpcMgmtStopServerListening(NULL);
if (status != RPC_S_OK)
{
    printf_s("RpcMgmtStopServerListening returned: 0x%x\n", status);
    exit(status);
}

status = RpcServerUnregisterIf(NULL, NULL, FALSE);
if (status != RPC_S_OK)
{
    printf_s("RpcServerUnregisterIf returned 0x%x\n", status);
    exit(status);
}
D.5 TPM_cmds.c

D.5.1 Description

This file contains the entry point for the simulator.

D.5.2 Includes, Defines, Data Definitions, and Function Prototypes

```c
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <windows.h>
#include <strsafe.h>
#include "string.h"
#include "TpmTcpProtocol.h"
#define PURPOSE "TPM Reference Simulator. Copyright Microsoft 2010, 2011."
define DEFAULT_TPM_PORT 2321
def MainPointer;
def TPM_Manufacture();
def _plat__NVEnable(void* platParameters);
def _plat__NVDisable();
def StartTcpServer(int PortNumber);
```

D.5.3 Functions

D.5.3.1 Usage()

This function prints the proper calling sequence for the simulator.

```c
void Usage(char * pszProgramName)
{
    fprintf_s(stderr, "%s", PURPOSE);
    fprintf_s(stderr, "Usage:
    
    %s PortNum - Starts the TPM server listening on port PortNum\n", pszProgramName, DEFAULT_TPM_PORT);
    fprintf_s(stderr, "%s PortNum - Starts the TPM server listening on port PortNum\n", pszProgramName);
    fprintf_s(stderr, "%s ? - This message\n", pszProgramName);
    exit(1);
}
```

D.5.3.2 main()

Entry point for the simulator.

```c
void __cdecl main(int argc, char * argv[])
{
    RPC_STATUS status;
    int portNum = DEFAULT_TPM_PORT;
    if(argc>2)
    {
        Usage(argv[0]);
    }
    if(argc==2)
```
```c
{   
    if(strcmp(argv[1], "?") ==0)  
    {   
        Usage(argv[0]);  
    }  
    portNum = atoi(argv[1]);  
    if(portNum <=0 || portNum>65535)  
    {   
        Usage(argv[0]);  
    }  
    _plat__NVEnable(NULL);  
    if(TPM_Manufacture() != 0)  
    {   
        status = RPC_S_INTERNAL_ERROR;  
        exit(status);  
    }  
    // Disable NV memory  
    _plat__NVDisable();  
    StartTcpServer(portNum);  
    return;  
}  
```