Trusted Platform Module Library

Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 01.16

October 30, 2014

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

TCG

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 owner.

CONTENTS

Trusted Platform Module Library
Part 4: Supporting Routines

1 Scope

This part contains C code that describes the algorithms and methods used by the command code in TPM 2.0 Part 3. The code in this document augments TPM 2.0 Part 2 and TPM 2.0 Part 3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

Any TPM 2.0 Part 4 code may be replaced by code that provides similar results when interfacing to the action code in TPM 2.0 Part 3. The behavior of code in this document that is not included in an annex is *normative*, as observed at the interfaces with TPM 2.0 Part 3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of the specification from the provided code.

The code in parts 3 and 4 is written to define the behavior of a compliant TPM. In some cases (e.g., firmware update), it is not possible to provide a compliant implementation. In those cases, any implementation provided by the vendor that meets the general description of the function provided in TPM 2.0 Part 3 would be compliant.

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

2 Terms and definitions

For the purposes of this document, the terms and definitions given in TPM 2.0 Part 1 apply.

3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in TPM 2.0 Part 1 apply.

4 Automation

TPM 2.0 Part 2 and 3 are constructed so that they can be processed by an automated parser. For example, TPM 2.0 Part 2 can be processed to generate header file contents such as structures, typedefs, and enums. TPM 2.0 Part 3 can be processed to generate command and response marshaling and unmarshaling code.

The automated processor is not provided to the TCG. It was used to generate the Microsoft Visual Studio TPM simulator files. These files are not specification reference code, but rather design examples.

4.1 Configuration Parser

The tables in the TPM 2.0 Part 2 Annexes are constructed so that they can be processed by a program. The program that processes these tables in the TPM 2.0 Part 2 Annexes is called "The TPM 2.0 Part 2 Configuration Parser."

The tables in the TPM 2.0 Part 2 Annexes determine the configuration of a TPM implementation. These tables may be modified by an implementer to describe the algorithms and commands to be executed in by a specific implementation as well as to set implementation limits such as the number of PCR, sizes of buffers, etc.

The TPM 2.0 Part 2 Configuration Parser produces a set of structures and definitions that are used by the TPM 2.0 Part 2 Structure Parser.

4.2 Structure Parser

4.2.1 Introduction

The program that processes the tables in TPM 2.0 Part 2 (other than the table in the annexes) is called "The TPM 2.0 Part 2 Structure Parser."

NOTE A Perl script was used to parse the tables in TPM 2.0 Part 2 to produce the header files and unmarshaling code in for the reference implementation.

The TPM 2.0 Part 2 Structure Parser takes as input the files produced by the TPM 2.0 Part 2 Configuration Parser and the same TPM 2.0 Part 2 specification that was used as input to the TPM 2.0 Part 2 Configuration Parser. The TPM 2.0 Part 2 Structure Parser will generate all of the C structure constant definitions that are required by the TPM interface. Additionally, the parser will generate unmarshaling code for all structures passed to the TPM, and marshaling code for structures passed from the TPM.

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

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

The sections below describe the function prototypes for the marshaling and unmarshaling code that is automatically generated by the TPM 2.0 Part 2 Structure Parser. These prototypes are described here as the unmarshaling and marshaling of various types occurs in places other than when the command is being parsed or the response is being built. The prototypes and the description of the interface are intended to aid in the comprehension of the code that uses these auto-generated routines.

4.2.2 Unmarshaling Code Prototype

4.2.2.1 Simple Types and Structures

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

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

Where:

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

When the data is successfully unmarshaled, the called routine will return TPM_RC_SUCCESS. Otherwise, it will return a Format-One response code (see TPM 2.0 Part 2).

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

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

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

4.2.2.2 Union Types

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

The function prototype for a union has the form:

```
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:

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

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

If the data is successfully marshaled, ***buffer** is advanced point to the first octet of the next location in the output buffer and ***size** is reduced by the number of octets placed in the buffer.

When the data type is a simple type, the parser will generate code that will marshal the underlying type. The presumption is that the TPM internal structures are consistent and correct so the marshaling code does not validate that the data placed in the buffer has a permissible value.

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

4.2.3.2 Union Types

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

The function prototype for a union has the form:

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

The parameters have a similar meaning as those in 5.2.2.2 but the data movement is from **source** to **buffer**.

4.2.3.3 Arrays

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

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

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

4.3 Command Parser

The program that processes the tables in TPM 2.0 Part 3 is called "The TPM 2.0 Part 3 Command Parser."

The TPM 2.0 Part 3 Command Parser takes as input a TPM 2.0 Part 3 of the TPM specification and some configuration files produced by the TPM 2.0 Part 2 Configuration Parser. This parser uses the contents of the command and response tables in TPM 2.0 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 TPM 2.0 Part 4.

4.4 Portability

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

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

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

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

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

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

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

5 Header Files

5.1 Introduction

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

5.2 BaseTypes.h

```
#ifndef _BASETYPES_H
#define _BASETYPES_H
#include "stdint.h"
```

NULL definition

```
#ifndef         NULL
#define         NULL        (0)
#endif
typedef  uint8_t            UINT8;
typedef  uint8_t            BYTE;
typedef  int8_t             INT8;
typedef  int                BOOL;
typedef  uint16_t           UINT16;
typedef  int16_t            INT16;
typedef  uint32_t           UINT32;
typedef  int32_t            INT32;
typedef  uint64_t           UINT64;
typedef  int64_t            INT64;
typedef struct {
    UINT16        size;
    BYTE          buffer[1];
} TPM2B;
#endif
```

5.3 bits.h

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

5.4 bool.h

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

5.5 Capabilities.h

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

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

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

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

5.6 TPMB.h

This file contains extra TPM2B structures

```
#ifndef _TPMB_H
#define _TPMB_H
```

This macro helps avoid having to type in the structure in order to create a new TPM2B type that is used in a function.

```
#define TPM2B_TYPE(name, bytes)             \
    typedef union {                         \
        struct  {                           \
            UINT16  size;                   \
            BYTE    buffer[(bytes)];        \
        } t;                                \
        TPM2B   b;                          \
    } TPM2B_##name
```

Macro to instance and initialize a TPM2B value

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

5.7 TpmError.h

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

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

```
void
TpmFail(const char *function, int line, int code);
typedef void     (*FAIL_FUNCTION)(const char *, int, int);
#define FAIL(a) (TpmFail(__FUNCTION__, __LINE__, a))
#if defined(EMPTY_ASSERT)
#   define pAssert(a)  ((void)0)
#else
#   define pAssert(a) (!!(a) ? 1 : (FAIL(FATAL_ERROR_PARAMETER), 0))
#endif
#endif // _TPM_ERROR_H
```

5.8 Global.h

5.8.1 Description

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

The first part of this file has the typedefs for structures and other defines used in many portions of the code. After the typedef 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.

5.8.2 Includes

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

5.8.3 Defines and Types

5.8.3.1 Unreferenced Parameter

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

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

5.8.3.2 Crypto Self-Test Values

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

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

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

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

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

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

5.8.3.3 Hash and HMAC State Structures

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

```
typedef BYTE    HASH_STATE_TYPE;
#define HASH_STATE_EMPTY        ((HASH_STATE_TYPE) 0)
#define HASH_STATE_HASH         ((HASH_STATE_TYPE) 1)
#define HASH_STATE_HMAC         ((HASH_STATE_TYPE) 2)
```

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

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

An HMAC_STATE structure contains an opaque HMAC stack state. A caller would use this structure when performing incremental HMAC operations. This structure contains a hash state and an HMAC key and allows slightly better stack optimization than adding an HMAC key to each hash state.

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

5.8.3.4 Other Types

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

```
39  typedef BYTE    AUTH_VALUE[sizeof(TPMU_HA)];
```

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

```
40  typedef BYTE    TIME_INFO[sizeof(TPMS_TIME_INFO)];
```

A NAME is a BYTE array that can contain a TPMU_NAME

```
41  typedef BYTE    NAME[sizeof(TPMU_NAME)];
```

5.8.4 Loaded Object Structures

5.8.4.1 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.

5.8.4.2 OBJECT_ATTRIBUTES

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

```
42  typedef struct
43  {
44      unsigned            publicOnly    : 1;   //0) SET if only the public portion of
45                                                //   an object is loaded
46      unsigned            epsHierarchy  : 1;   //1) SET if the object belongs to EPS
47                                                //   Hierarchy
48      unsigned            ppsHierarchy  : 1;   //2) SET if the object belongs to PPS
49                                                //   Hierarchy
50      unsigned            spsHierarchy  : 1;   //3) SET f the object belongs to SPS
51                                                //   Hierarchy
52      unsigned            evict         : 1;   //4) SET if the object is a platform or
53                                                //   owner evict object.  Platform-
54                                                //   evict object belongs to PPS
55                                                //   hierarchy, owner-evict object
56                                                //   belongs to SPS or EPS hierarchy.
57                                                //   This bit is also used to mark a
58                                                //   completed sequence object so it
59                                                //   will be flush when the
60                                                //   SequenceComplete command succeeds.
61      unsigned            primary       : 1;   //5) SET for a primary object
```

```
62      unsigned            temporary   : 1;    //6) SET for a temporary object
63      unsigned            stClear     : 1;    //7) SET for an stClear object
64      unsigned            hmacSeq     : 1;    //8) SET for an HMAC sequence object
65      unsigned            hashSeq     : 1;    //9) SET for a hash sequence object
66      unsigned            eventSeq    : 1;    //10) SET for an event sequence object
67      unsigned            ticketSafe  : 1;    //11) SET if a ticket is safe to create
68                                              //    for hash sequence object
69      unsigned            firstBlock  : 1;    //12) SET if the first block of hash
70                                              //    data has been received.  It
71                                              //    works with ticketSafe bit
72      unsigned            isParent    : 1;    //13) SET if the key has the proper
73                                              //    attributes to be a parent key
74      unsigned            privateExp  : 1;    //14) SET when the private exponent
75                                              //    of an RSA key has been validated.
76      unsigned        reserved    : 1;    //15) reserved bits. unused.
77  } OBJECT_ATTRIBUTES;
```

5.8.4.3 OBJECT Structure

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

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

5.8.4.4 HASH_OBJECT Structure

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

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

```
99  typedef struct
100 {
101     OBJECT_ATTRIBUTES   attributes;         // The attributes of the HASH object
102     TPMI_ALG_PUBLIC     type;               // algorithm
103     TPMI_ALG_HASH       nameAlg;            // name algorithm
104     TPMA_OBJECT         objectAttributes;   // object attributes
```



```
105
106         // The data below is unique to a sequence object
107         TPM2B_AUTH          auth;               // auth for use of sequence
108         union
109         {
110             HASH_STATE      hashState[HASH_COUNT];
111             HMAC_STATE      hmacState;
112         }                   state;
113     } HASH_OBJECT;
```

5.8.4.5 ANY_OBJECT

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

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

5.8.5 AUTH_DUP Types

These values are used in the authorization processing.

```
119     typedef UINT32          AUTH_ROLE;
120     #define AUTH_NONE       ((AUTH_ROLE)(0))
121     #define AUTH_USER       ((AUTH_ROLE)(1))
122     #define AUTH_ADMIN      ((AUTH_ROLE)(2))
123     #define AUTH_DUP        ((AUTH_ROLE)(3))
```

5.8.6 Active Session Context

5.8.6.1 Description

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

5.8.6.2 SESSION_ATTRIBUTES

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

```
124     typedef struct
125     {
126         unsigned            isPolicy : 1;       //1) SET if the session may only
127                                                 //   be used for policy
128         unsigned            isAudit : 1;        //2) SET if the session is used
129                                                 //   for audit
130         unsigned            isBound : 1;        //3) SET if the session is bound to
131                                                 //   with an entity.
132                                                 //   This attribute will be CLEAR if
133                                                 //   either isPolicy or isAudit is SET.
134         unsigned            iscpHashDefined : 1;//4) SET if the cpHash has been defined
135                                                 //   This attribute is not SET unless
136                                                 //   'isPolicy' is SET.
137         unsigned            isAuthValueNeeded : 1;
138                                                 //5) SET if the authValue is required
139                                                 //   for computing the session HMAC.
140                                                 //   This attribute is not SET unless
```

```
                                        //   isPolicy is SET.
    unsigned            isPasswordNeeded : 1;
                                        //6) SET if a password authValue is
                                        //   required for authorization
                                        //   This attribute is not SET unless
                                        //   isPolicy is SET.
    unsigned            isPPRequired : 1;   //7) SET if physical presence is
                                        //   required to be asserted when the
                                        //   authorization is checked.
                                        //   This attribute is not SET unless
                                        //   isPolicy is SET.
    unsigned            isTrialPolicy : 1;  //8) SET if the policy session is
                                        //   created for trial of the policy's
                                        //   policyHash generation.
                                        //   This attribute is not SET unless
                                        //   isPolicy is SET.
    unsigned            isDaBound : 1;      //9) SET if the bind entity had noDA
                                        //   CLEAR. If this is SET, then an
                                        //   auth failure using this session
                                        //   will count against lockout even
                                        //   if the object being authorized is
                                        //   exempt from DA.
    unsigned            isLockoutBound : 1; //10)SET if the session is bound to
                                        //   lockoutAuth.
    unsigned            requestWasBound : 1;//11) SET if the session is being used
                                        //    with the bind entity. If SET
                                        //    the authValue will not be use
                                        //    in the response HMAC computation.
    unsigned            checkNvWritten : 1; //12) SET if the TPMA_NV_WRITTEN
                                        //    attribute needs to be checked
                                        //    when the policy is used for
                                        //    authorization for NV access.
                                        //    If this is SET for any other
                                        //    type, the policy will fail.
    unsigned            nvWrittenState : 1; //13) SET if TPMA_NV_WRITTEN is
                                        //    required to be SET.
} SESSION_ATTRIBUTES;
```

5.8.6.3 SESSION Structure

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

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

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

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

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

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

```
198                                             // session)
199
200         UINT64              timeOut;        // timeout relative to
201                                             // TPMS_CLOCK_INFO.clock
202                                             // There is no timeout if this value
203                                             // is 0.
204         union
205         {
206             TPM2B_NAME      boundEntity;     // value used to track the entity to
207                                              // which the session is bound
208
209             TPM2B_DIGEST    cpHash;          // the required cpHash value for the
210                                              // command being authorized
211
212         } u1;                                // 'boundEntity' and 'cpHash' may
213                                              // share the same space to save memory
214
215         union
216         {
217             TPM2B_DIGEST    auditDigest;    // audit session digest
218             TPM2B_DIGEST    policyDigest;     // policyHash
219
220         } u2;                               // audit log and policyHash may
221                                             // share space to save memory
222     } SESSION;
```

5.8.7 PCR

5.8.7.1 PCR_SAVE Structure

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

```
223     typedef struct
224     {
225     #ifdef TPM_ALG_SHA1
226         BYTE                sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
227     #endif
228     #ifdef TPM_ALG_SHA256
229         BYTE                sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
230     #endif
231     #ifdef TPM_ALG_SHA384
232         BYTE                sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
233     #endif
234     #ifdef TPM_ALG_SHA512
235         BYTE                sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
236     #endif
237     #ifdef TPM_ALG_SM3_256
238         BYTE                sm3_256[NUM_STATIC_PCR][SM3_256_DIGEST_SIZE];
239     #endif
240
241         // This counter increments whenever the PCR are updated.
242         // NOTE: A platform-specific specification may designate
243         //       certain PCR changes as not causing this counter
244         //       to increment.
245         UINT32              pcrCounter;
246
247     } PCR_SAVE;
```

5.8.7.2 PCR_POLICY

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

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

5.8.7.3 PCR_AUTHVALUE

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

```
typedef struct
{
    TPM2B_DIGEST        auth[NUM_AUTHVALUE_PCR_GROUP];
} PCR_AUTHVALUE;
```

5.8.8 Startup

5.8.8.1 SHUTDOWN_NONE

Part 2 defines the two shutdown/startup types that may be used in TPM2_Shutdown() and TPM2_Starup(). This additional define is used by the TPM to indicate that no shutdown was received.

NOTE: This is a reserved value.

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

5.8.8.2 STARTUP_TYPE

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

```
typedef enum
{
    SU_RESET,
    SU_RESTART,
    SU_RESUME
} STARTUP_TYPE;
```

5.8.9 NV

5.8.9.1 NV_RESERVE

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

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

```
270         NV_DISABLE_CLEAR,
271         NV_OWNER_ALG,
272         NV_ENDORSEMENT_ALG,
273         NV_LOCKOUT_ALG,
274         NV_OWNER_POLICY,
275         NV_ENDORSEMENT_POLICY,
276         NV_LOCKOUT_POLICY,
277         NV_OWNER_AUTH,
278         NV_ENDORSEMENT_AUTH,
279         NV_LOCKOUT_AUTH,
280
281         NV_EP_SEED,
282         NV_SP_SEED,
283         NV_PP_SEED,
284
285         NV_PH_PROOF,
286         NV_SH_PROOF,
287         NV_EH_PROOF,
288
289         // Time
290         NV_TOTAL_RESET_COUNT,
291         NV_RESET_COUNT,
292
293         // PCR
294         NV_PCR_POLICIES,
295         NV_PCR_ALLOCATED,
296
297         // Physical Presence
298         NV_PP_LIST,
299
300         // Dictionary Attack
301         NV_FAILED_TRIES,
302         NV_MAX_TRIES,
303         NV_RECOVERY_TIME,
304         NV_LOCKOUT_RECOVERY,
305         NV_LOCKOUT_AUTH_ENABLED,
306
307         // Orderly State flag
308         NV_ORDERLY,
309
310         // Command Audit
311         NV_AUDIT_COMMANDS,
312         NV_AUDIT_HASH_ALG,
313         NV_AUDIT_COUNTER,
314
315         // Algorithm Set
316         NV_ALGORITHM_SET,
317
318         NV_FIRMWARE_V1,
319         NV_FIRMWARE_V2,
320
321     // The entries above are in PERSISTENT_DATA. The entries below represent
322     // structures that are read and written as a unit.
323
324     // ORDERLY_DATA data structure written on each orderly shutdown
325         NV_ORDERLY_DATA,
326
327     // STATE_CLEAR_DATA structure written on each Shutdown(STATE)
328         NV_STATE_CLEAR,
329
330     // STATE_RESET_DATA structure written on each Shutdown(STATE)
331         NV_STATE_RESET,
332
333         NV_RESERVE_LAST             // end of NV reserved data list
334     } NV_RESERVE;
```

5.8.9.2 NV_INDEX

The NV_INDEX structure defines the internal format for an NV index. The *indexData* size varies according to the type of the index. In this implementation, all of the index is manipulated as a unit.

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

5.8.10 COMMIT_INDEX_MASK

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

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

5.8.11 RAM Global Values

5.8.11.1 Description

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

5.8.11.2 g_rcIndex

This array is used to contain the array of values that are added to a return code when it is a parameter-, handle-, or session-related error. This is an implementation choice and the same result can be achieved by using a macro.

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

5.8.11.3 g_exclusiveAuditSession

This location holds the session handle for the current exclusive audit session. If there is no exclusive audit session, the location is set to TPM_RH_UNASSIGNED.

```
344  extern TPM_HANDLE       g_exclusiveAuditSession;
```

5.8.11.4 g_time

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

```
345  extern  UINT64          g_time;
```

5.8.11.5 g_phEnable

This is the platform hierarchy control and determines if the platform hierarchy is available. This value is SET on each TPM2_Startup(). The default value is SET.

```
346  extern BOOL             g_phEnable;
```


5.8.11.6 g_pceReConfig

This value is SET if a TPM2_PCR_Allocate() command successfully executed since the last TPM2_Startup(). If so, then the next shutdown is required to be Shutdown(CLEAR).

```
347   extern BOOL             g_pcrReConfig;
```

5.8.11.7 g_DRTMHandle

This location indicates the sequence object handle that holds the DRTM sequence data. When not used, it is set to TPM_RH_UNASSIGNED. A sequence DRTM sequence is started on either _TPM_Init() or _TPM_Hash_Start().

```
348   extern TPMI_DH_OBJECT   g_DRTMHandle;
```

5.8.11.8 g_DrtmPreStartup

This value indicates that an H-CRTM occurred after _TPM_Init() but before TPM2_Startup(). The define for PRE_STARTUP_FLAG is used to add the *g_DrtmPreStartup* value to *gp_orderlyState* at shutdown. This hack is to avoid adding another NV variable.

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

5.8.11.9 g_StartupLocality3

This value indicates that a TPM2_Startup() occured at locality 3. Otherwise, it at locality 0. The define for STARTUP_LOCALITY_3 is to indicate that the startup was not at locality 0. This hack is to avoid adding another NV variable.

```
351   extern  BOOL            g_StartupLocality3;
352   #define STARTUP_LOCALITY_3       0x4000
```

5.8.11.10 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.

```
353   extern BOOL             g_updateNV;
```

5.8.11.11 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.

```
354   extern BOOL             g_clearOrderly;
```


5.8.11.12 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.

```
355  extern TPM_SU           g_prevOrderlyState;
```

5.8.11.13 g_nvOk

This value indicates if the NV integrity check was successful or not. If not and the failure was severe, then the TPM would have been put into failure mode after it had been re-manufactured. If the NV failure was in the area where the state-save data is kept, then this variable will have a value of FALSE indicating that a TPM2_Startup(CLEAR) is required.

```
356  extern BOOL             g_nvOk;
```

5.8.11.14 g_platformUnique

This location contains the unique value(s) used to identify the TPM. It is loaded on every _TPM2_Startup() The first value is used to seed the RNG. The second value is used as a vendor *authValue*. The value used by the RNG would be the value derived from the chip unique value (such as fused) with a dependency on the authorities of the code in the TPM boot path. The second would be derived from the chip unique value with a dependency on the details of the code in the boot path. That is, the first value depends on the various signers of the code and the second depends on what was signed. The TPM vendor should not be able to know the first value but they are expected to know the second.

```
357  extern TPM2B_AUTH       g_platformUniqueAuthorities; // Reserved for RNG
358  extern TPM2B_AUTH       g_platformUniqueDetails;   // referenced by VENDOR_PERMANENT
```

5.8.12 Persistent Global Values

5.8.12.1 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.

5.8.12.2 PERSISTENT_DATA

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

```
359  typedef struct
360  {
361  //*****************************************************************************
362  //          Hierarchy
363  //*****************************************************************************
364  // The values in this section are related to the hierarchies.
365
366      BOOL                disableClear;       // TRUE if TPM2_Clear() using
367                                              // lockoutAuth is disabled
368
369      // Hierarchy authPolicies
370      TPMI_ALG_HASH       ownerAlg;
371      TPMI_ALG_HASH       endorsementAlg;
372      TPMI_ALG_HASH       lockoutAlg;
373      TPM2B_DIGEST        ownerPolicy;
```

```
374         TPM2B_DIGEST        endorsementPolicy;
375         TPM2B_DIGEST        lockoutPolicy;
376
377         // Hierarchy authValues
378         TPM2B_AUTH          ownerAuth;
379         TPM2B_AUTH          endorsementAuth;
380         TPM2B_AUTH          lockoutAuth;
381
382         // Primary Seeds
383         TPM2B_SEED          EPSeed;
384         TPM2B_SEED          SPSeed;
385         TPM2B_SEED          PPSeed;
386         // Note there is a nullSeed in the state_reset memory.
387
388         // Hierarchy proofs
389         TPM2B_AUTH          phProof;
390         TPM2B_AUTH          shProof;
391         TPM2B_AUTH          ehProof;
392         // Note there is a nullProof in the state_reset memory.
393
394     //*****************************************************************************
395     //          Reset Events
396     //*****************************************************************************
397     // A count that increments at each TPM reset and never get reset during the life
398     // time of TPM.  The value of this counter is initialized to 1 during TPM
399     // manufacture process.
400         UINT64              totalResetCount;
401
402     // This counter increments on each TPM Reset. The counter is reset by
403     // TPM2_Clear().
404         UINT32              resetCount;
405
406     //*****************************************************************************
407     //          PCR
408     //*****************************************************************************
409     // This structure hold the policies for those PCR that have an update policy.
410     // This implementation only supports a single group of PCR controlled by
411     // policy. If more are required, then this structure would be changed to
412     // an array.
413         PCR_POLICY          pcrPolicies;
414
415     // This structure indicates the allocation of PCR. The structure contains a
416     // list of PCR allocations for each implemented algorithm. If no PCR are
417     // allocated for an algorithm, a list entry still exists but the bit map
418     // will contain no SET bits.
419         TPML_PCR_SELECTION  pcrAllocated;
420
421     //*****************************************************************************
422     //          Physical Presence
423     //*****************************************************************************
424     // The PP_LIST type contains a bit map of the commands that require physical
425     // to be asserted when the authorization is evaluated. Physical presence will be
426     // checked if the corresponding bit in the array is SET and if the authorization
427     // handle is TPM_RH_PLATFORM.
428     //
429     // These bits may be changed with TPM2_PP_Commands().
430         BYTE                ppList[((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7)/8];
431
432     //*****************************************************************************
433     //          Dictionary attack values
434     //*****************************************************************************
435     // These values are used for dictionary attack tracking and control.
436         UINT32              failedTries;        // the current count of unexpired
437                                                 // authorization failures
438
439         UINT32              maxTries;           // number of unexpired authorization
```

```
                                        // failures before the TPM is in
                                        // lockout

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

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

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

//*****************************************************************************
//            Orderly State
//*****************************************************************************
// The orderly state for current cycle
    TPM_SU              orderlyState;

//*****************************************************************************
//           Command audit values.
//*****************************************************************************
    BYTE                auditComands[((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8];
    TPMI_ALG_HASH       auditHashAlg;
    UINT64              auditCounter;

//*****************************************************************************
//           Algorithm selection
//*****************************************************************************
//
// The 'algorithmSet' value indicates the collection of algorithms that are
// currently in used on the TPM.  The interpretation of value is vendor dependent.
    UINT32              algorithmSet;

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

// g_firmwareV1 contains the more significant 32-bits of the vendor version number.
// In the reference implementation, if this value is printed as a hex
// value, it will have the format of yyyymmdd
    UINT32              firmwareV1;

// g_firmwareV1 contains the less significant 32-bits of the vendor version number.
// In the reference implementation, if this value is printed as a hex
// value, it will have the format of 00 hh mm ss
    UINT32              firmwareV2;

} PERSISTENT_DATA;
extern PERSISTENT_DATA  gp;
```

5.8.12.3 ORDERLY_DATA

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

```
typedef struct orderly_data
{

```

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

5.8.12.4 STATE_CLEAR_DATA

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

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

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

5.8.12.5 State Reset Data

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

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

```
551  typedef struct state_reset_data
552  {
553  //*****************************************************************************
554  //          Hierarchy Control
555  //*****************************************************************************
556      TPM2B_AUTH          nullProof;          // The proof value associated with
557                                              // the TPM_RH_NULL hierarchy. The
558                                              // default reset value is from the RNG.
559
560      TPM2B_SEED          nullSeed;           // The seed value for the TPM_RN_NULL
561                                              // hierarchy. The default reset value
562                                              // is from the RNG.
563
564  //*****************************************************************************
565  //           Context
566  //*****************************************************************************
567  // The 'clearCount' counter is incremented each time the TPM successfully executes
568  // a TPM Resume. The counter is included in each saved context that has 'stClear'
569  // SET (including descendants of keys that have 'stClear' SET). This prevents these
570  // objects from being loaded after a TPM Resume.
571  // If 'clearCount' at its maximum value when the TPM receives a Shutdown(STATE),
572  // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
573      UINT32              clearCount;         // The default reset value is 0.
574
575      UINT64              objectContextID;    // This is the context ID for a saved
576                                              //  object context. The default reset
577                                              //  value is 0.
578
579      CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];
580                                              // This is the value from which the
581                                              // 'contextID' is derived. The
582                                              // default reset value is {0}.
583
584      CONTEXT_COUNTER     contextCounter;     // This array contains contains the
585                                              // values used to track the version
586                                              // numbers of saved contexts (see
587                                              // Session.c in for details). The
588                                              // default reset value is 0.
589
590  //*****************************************************************************
591  //           Command Audit
592  //*****************************************************************************
593  // When an audited command completes, ExecuteCommand() checks the return
594  // value.  If it is TPM_RC_SUCCESS, and the command is an audited command, the
595  // TPM will extend the cpHash and rpHash for the command to this value. If this
596  // digest was the Zero Digest before the cpHash was extended, the audit counter
597  // is incremented.
598
599      TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest
600                                              // by TPM2_GetCommandAuditDigest() or a
601                                              // TPM Reset.
602
603  //*****************************************************************************
604  //           Boot counter
605  //*****************************************************************************
606
607      UINT32              restartCount;       // This counter counts TPM Restarts.
608                                              // The default reset value is 0.
```

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

5.8.13 Global Macro Definitions

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

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

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

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

5.8.14 Private data

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

From SessionProcess.c

The following arrays are used to save command sessions information so that the command handle/session buffer does not have to be preserved for the duration of the command. These arrays are indexed by the session index in accordance with the order of sessions in the session area of the command.

Array of the authorization session handles

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

Array of authorization session attributes

```
extern TPMA_SESSION     s_attributes[MAX_SESSION_NUM];
```

Array of handles authorized by the corresponding authorization sessions; and if none, then TPM_RH_UNASSIGNED value is used

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

Array of nonces provided by the caller for the corresponding sessions

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

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

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

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter

```
extern UINT32           s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
extern UINT32           s_decryptSessionIndex;
```

Index of a session used for audit

```
extern UINT32           s_auditSessionIndex;
```

The *cpHash* for an audit session

```
extern TPM2B_DIGEST     s_cpHashForAudit;
```

The *cpHash* for command audit

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

Number of authorization sessions present in the command

```
extern UINT32           s_sessionNum;
```

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

```
extern BOOL             s_DAPendingOnNV;
#endif // SESSION_PROCESS_C
#if defined DA_C || defined GLOBAL_C || defined MANUFACTURE_C
```

From DA.c

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

```
665   extern UINT64       s_selfHealTimer;
```

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

```
666   extern UINT64       s_lockoutTimer;
667   #endif // DA_C
668   #if defined NV_C || defined GLOBAL_C
```

From NV.c

List of pre-defined address of reserved data

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

List of pre-defined reserved data size in byte

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

Size of data in RAM index buffer

```
671   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

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

Address of size of RAM index space in NV

```
673   extern UINT32   s_ramIndexSizeAddr;
```

Address of NV copy of RAM index space

```
674   extern UINT32   s_ramIndexAddr;
```

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

```
675   extern UINT32   s_maxCountAddr;
```

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

```
676   extern UINT32   s_evictNvStart;
```

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

```
677   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

```
678   extern TPM_RC   s_NvStatus;
679   #endif
680   #if defined OBJECT_C || defined GLOBAL_C
```

From Object.c

This type is the container for an object.

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

This is the memory that holds the loaded objects.

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

From PCR.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;
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;
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

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

```
730   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)

```
731   extern int               s_freeSessionSlots;
732   #endif // SESSION_C
```

From Manufacture.c

```
733   extern BOOL              g_manufactured;
734   #if defined POWER_C || defined GLOBAL_C
```

From Power.c

This value indicates if a TPM2_Startup() commands has been receive since the power on event. This flag is maintained in power simulation module because this is the only place that may reliably set this flag to FALSE.

```
735   extern BOOL              s_initialized;
736   #endif // POWER_C
737   #if defined MEMORY_LIB_C || defined GLOBAL_C
```

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

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

From TPMFail.c

This value holds the address of the string containing the name of the function in which the failure occurred. This address value isn't useful for anything other than helping the vendor to know in which file the failure occurred.

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

5.9 Tpm.h

Root header file for building any TPM.lib code

```
1   #ifndef     _TPM_H_
2   #define     _TPM_H_
3   #include    "bool.h"
```

```
#include    "Implementation.h"
#include    "TPM_Types.h"
#include    "swap.h"
#endif // _TPM_H_
```

5.10 swap.h

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

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

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

Disaggregate a UINT into a byte array

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

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

```
#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

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

5.11 InternalRoutines.h

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

NULL definition

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

UNUSED_PARAMETER

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

Internal data definition

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

Error Reporting

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

DRTM functions

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

Internal subsystem functions

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

Internal support functions

```
23  #include "CommandCodeAttributes_fp.h"
24  #include "MemoryLib_fp.h"
25  #include "marshal_fp.h"
26  #include "Time_fp.h"
27  #include "Locality_fp.h"
28  #include "PP_fp.h"
29  #include "CommandAudit_fp.h"
30  #include "Manufacture_fp.h"
31  #include "Power_fp.h"
```

```
32  #include "Handle_fp.h"
33  #include "Commands_fp.h"
34  #include "AlgorithmCap_fp.h"
35  #include "PropertyCap_fp.h"
36  #include "Bits_fp.h"
```

Internal crypto functions

```
37  #include "Ticket_fp.h"
38  #include "CryptUtil_fp.h"
39  #include "CryptSelfTest_fp.h"
40  #endif
```

5.12 TpmBuildSwitches.h

This file contains the build switches. This contains switches for multiple versions of the crypto-library so some may not apply to your environment.

```
1   #ifndef _TPM_BUILD_SWITCHES_H
2   #define _TPM_BUILD_SWITCHES_H
3   #define SIMULATION
4   #define FIPS_COMPLIANT
```

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

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

For ISO 9899:2011

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

This switch enables the RNG state save and restore

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

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

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

```
9   #ifdef CRYPTO_ALIGN_16
10  #   define CRYPTO_ALIGNMENT     16
11  #elif defined CRYPTO_ALIGN_8
12  #   define CRYPTO_ALIGNMENT     8
13  #eliF defined CRYPTO_ALIGN_2
14  #   define  CRYPTO_ALIGNMENT    2
15  #elif defined CRTYPO_ALIGN_1
16  #   define  CRYPTO_ALIGNMENT    1
17  #else
18  #   define CRYPTO_ALIGNMENT     4    // For 32-bit builds
19  #endif
20  #define CRYPTO_ALIGNED  ALIGN_TO(CRYPTO_ALIGNMENT)
```

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

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

For import of a variable

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

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

```
25  #define _No_Return_  __declspec(noreturn)
26  //#define _No_Return_
27  #ifdef SELF_TEST
28  #pragma comment(lib, "algorithmtests.lib")
29  #endif
```

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

```
30  #ifdef SIMULATION
31  #   define RSA_KEY_CACHE
32  #   define TPM_RNG_FOR_DEBUG
33  #else
34  #   undef RSA_KEY_CACHE
35  #   undef TPM_RNG_FOR_DEBUG
36  #endif  // SIMULATION
37  #define INLINE  __inline
38  #endif // _TPM_BUILD_SWITCHES_H
```

5.13 VendorString.h

```
1   #ifndef     _VENDOR_STRING_H
2   #define     _VENDOR_STRING_H
```

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

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

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

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

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

```
9   #define       VENDOR_STRING_1       "xCG "
10  #define       VENDOR_STRING_2       "fTPM"
11  // #define       VENDOR_STRING_3
12  // #define       VENDOR_STRING_4
```

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

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

```
17  #endif
```

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

```
18  #define   FIRMWARE_V1         (0x20140504)
```

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

```
19  #define   FIRMWARE_V2         (0x00200136)
```

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

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

6 Main

6.1 CommandDispatcher()

In the reference implementation, a program that uses TPM 2.0 Part 3 as input automatically generates the command dispatch code. The function prototype header file (CommandDispatcher_fp.h) is shown here.

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;
- invokes the function that performs the command actions;
- marshals the returned handles, if any; and
- marshals the returned parameters, if any, into the output buffer putting in the *parameterSize* field if authorization sessions are present.

```
1   #ifndef    _COMMANDDISPATCHER_FP_H_
2   #define    _COMMANDDISPATCHER_FP_H_
3   TPM_RC
4   CommandDispatcher(
5       TPMI_ST_COMMAND_TAG     tag,    // IN: Input command tag
6       TPM_CC      commandCode,        // IN: Command code
7       INT32       *parmBufferSize,    // IN: size of parameter buffer
8       BYTE        *parmBufferStart,   // IN: pointer to start of parameter buffer
9       TPM_HANDLE  handles[],          // IN: handle array
10      UINT32      *responseHandleSize,// OUT: size of handle buffer in response
11      UINT32      *respParmSize       // OUT: size of parameter buffer in response
12      );
13  #endif  // _COMMANDDISPATCHER_FP_H_
```

6.2 ExecCommand.c

6.2.1 Introduction

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

6.2.2 Includes

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

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

```
5   // #include "CommandResponseSizes_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.

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

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

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

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

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

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

    TPM_ST               resTag;            // tag for the response

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

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

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

```
                                        // response

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

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

    UINT32              i;               // local temp

// This next function call is used in development to size the command and response
// buffers. The values printed are the sizes of the internal structures and
// not the sizes of the canonical forms of the command response structures. Also,
// the sizes do not include the tag, commandCode, requestSize, or the authorization
// fields.
//CommandResponseSizes();

    // Set flags for NV access state. This should happen before any other
    // operation that may require a NV write. Note, that this needs to be done
    // even when in failure mode. Otherwise, g_updateNV would stay SET while in
    // Failure mode and the NB would be written on each call.
    g_updateNV = FALSE;
    g_clearOrderly = FALSE;

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

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

    // Query platform to get the NV state.  The result state is saved internally
    // and will be reported by NvIsAvailable(). The reference code requires that
    // accessibility of NV does not change during the execution of a command.
    // Specifically, if NV is available when the command execution starts and then
    // is not available later when it is necessary to write to NV, then the TPM
    // will go into failure mode.
    NvCheckState();

    // Due to the limitations of the simulation, TPM clock must be explicitly
    // synchronized with the system clock whenever a command is received.
    // This function call is not necessary in a hardware TPM. However, taking
    // a snapshot of the hardware timer at the beginning of the command allows
    // the time value to be consistent for the duration of the command execution.
    TimeUpdateToCurrent();

    // Any command through this function will unceremoniously end the
    // _TPM_Hash_Data/_TPM_Hash_End sequence.
    if(g_DRTMHandle != TPM_RH_UNASSIGNED)
        ObjectTerminateEvent();
```

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



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

```
244             if(result != TPM_RC_SUCCESS)
245                 goto Cleanup;
246         }
247         else
248         {
249             // Whatever remains in the input buffer is used for the parameters of the
250             // command.
251             parmBufferStart = buffer;
252             parmBufferSize = size;
253 
254             // The command has no authorization sessions.
255             // If the command requires authorizations, then CheckAuthNoSession() will
256             // return an error.
257             result = CheckAuthNoSession(commandCode, handleNum, handles,
258                                         parmBufferStart, parmBufferSize);
259             if(result != TPM_RC_SUCCESS)
260                 goto Cleanup;
261         }
262 
263         // CommandDispatcher returns a response handle buffer and a response parameter
264         // buffer if it succeeds. It will also set the parameterSize field in the
265         // buffer if the tag is TPM_RC_SESSIONS.
266         result = CommandDispatcher(tag,
267                                    commandCode,
268                                    (INT32 *) &parmBufferSize,
269                                    parmBufferStart,
270                                    handles,
271                                    &resHandleSize,
272                                    &resParmSize);
273         if(result != TPM_RC_SUCCESS)
274             goto Cleanup;
275 
276         // Build the session area at the end of the parameter area.
277         BuildResponseSession(tag,
278                              commandCode,
279                              resHandleSize,
280                              resParmSize,
281                              &resAuthSize);
282 
283     Cleanup:
284         // This implementation loads an "evict" object to a transient object slot in
285         // RAM whenever an "evict" object handle is used in a command so that the
286         // access to any object is the same. These temporary objects need to be
287         // cleared from RAM whether the command succeeds or fails.
288         ObjectCleanupEvict();
289 
290     Fail:
291         // The response will contain at least a response header.
292         *responseSize = sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC);
293 
294         // If the command completed successfully, then build the rest of the response.
295         if(result == TPM_RC_SUCCESS)
296         {
297             // Outgoing tag will be the same as the incoming tag.
298             resTag = tag;
299             // The overall response will include the handles, parameters,
300             // and authorizations.
301             *responseSize += resHandleSize + resParmSize + resAuthSize;
302 
303             // Adding parameter size field.
304             if(tag == TPM_ST_SESSIONS)
305                 *responseSize += sizeof(UINT32);
306 
307             if(   g_clearOrderly == TRUE
308                && gp.orderlyState != SHUTDOWN_NONE)
309             {
```

```
310             gp.orderlyState = SHUTDOWN_NONE;
311             NvWriteReserved(NV_ORDERLY, &gp.orderlyState);
312             g_updateNV = TRUE;
313         }
314     }
315     else
316     {
317         // The command failed.
318         // If this was a failure due to a bad command tag, then need to return
319         // a TPM 1.2 compatible response
320         if(result == TPM_RC_BAD_TAG)
321             resTag = TPM_ST_RSP_COMMAND;
322         else
323             // return 2.0 compatible response
324             resTag = TPM_ST_NO_SESSIONS;
325     }
326     // Try to commit all the writes to NV if any NV write happened during this
327     // command execution. This check should be made for both succeeded and failed
328     // commands, because a failed one may trigger a NV write in DA logic as well.
329     // This is the only place in the command execution path that may call the NV
330     // commit. If the NV commit fails, the TPM should be put in failure mode.
331     if(g_updateNV && !g_inFailureMode)
332     {
333         g_updateNV = FALSE;
334         if(!NvCommit())
335             FAIL(FATAL_ERROR_INTERNAL);
336     }
337
338     // Marshal the response header.
339     buffer = MemoryGetResponseBuffer(commandCode);
340     TPM_ST_Marshal(&resTag, &buffer, NULL);
341     UINT32_Marshal((UINT32 *)responseSize, &buffer, NULL);
342     pAssert(*responseSize <= MAX_RESPONSE_SIZE);
343     TPM_RC_Marshal(&result, &buffer, NULL);
344
345     *response = MemoryGetResponseBuffer(commandCode);
346
347     // Clear unused bit in response buffer.
348     MemorySet(*response + *responseSize, 0, MAX_RESPONSE_SIZE - *responseSize);
349
350     return;
351 }
```

6.3 ParseHandleBuffer()

In the reference implementation, the routine for unmarshaling the command handles is automatically generated from TPM 2.0 Part 3 command tables. The prototype header file (HandleProcess_fp.h) is shown here.

```
1   #ifndef    _HANDLEPROCESS_FP_H_
2   #define    _HANDLEPROCESS_FP_H_
3   TPM_RC
4   ParseHandleBuffer(
5       TPM_CC      commandCode,                // IN: Command being processed
6       BYTE        **handleBufferStart,        // IN/OUT: command buffer where handles
7                                               //   are located. Updated as handles
8                                               //   are unmarshaled
9       INT32       *bufferRemainingSize,       // IN/OUT: indicates the amount of data
10                                              //   left in the command buffer.
11                                              //   Updated as handles are unmarshaled
12      TPM_HANDLE  handles[],                  // OUT: Array that receives the handles
13      UINT32      *handleCount                // OUT: Receives the count of handles
14      );
15  #endif  // _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

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

6.4.3 Authorization Support Functions

6.4.3.1 IsDAExempted()

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

a) a primary seed handle,

b) an object with *noDA* bit SET,

c) an NV Index with TPMA_NV_NO_DA bit SET, or

d) a PCR handle.

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

```
BOOL
IsDAExempted(
    TPM_HANDLE       handle          // IN: entity handle
    )
{
    BOOL        result = FALSE;

    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            // All permanent handles, other than TPM_RH_LOCKOUT, are exempt from
            // DA protection.
            result =  (handle != TPM_RH_LOCKOUT);
            break;

        // When this function is called, a persistent object will have been loaded
        // into an object slot and assigned a transient handle.
        case TPM_HT_TRANSIENT:
        {
            OBJECT      *object;
            object = ObjectGet(handle);
            result = (object->publicArea.objectAttributes.noDA == SET);
            break;
        }
        case TPM_HT_NV_INDEX:
        {
            NV_INDEX        nvIndex;
```

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

6.4.3.2 IncrementLockout()

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

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

```
45  static TPM_RC
46  IncrementLockout(
47      UINT32           sessionIndex
48      )
49  {
50      TPM_HANDLE       handle = s_associatedHandles[sessionIndex];
51      TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
52      TPM_RC           result;
53      SESSION         *session = NULL;
54
55      // Don't increment lockout unless the handle associated with the session
56      // is DA protected or the session is bound to a DA protected entity.
57      if(sessionHandle == TPM_RS_PW)
58      {
59          if(IsDAExempted(handle))
60              return TPM_RC_BAD_AUTH;
61
62      }
63      else
64      {
65          session = SessionGet(sessionHandle);
66          // If the session is bound to lockout, then use that as the relevant
67          // handle. This means that an auth failure with a bound session
68          // bound to lockoutAuth will take precedence over any other
69          // lockout check
70          if(session->attributes.isLockoutBound == SET)
71              handle = TPM_RH_LOCKOUT;
72
73          if(   session->attributes.isDaBound == CLEAR
74             && IsDAExempted(handle)
75            )
76              // If the handle was changed to TPM_RH_LOCKOUT, this will not return
77              // TPM_RC_BAD_AUTH
78              return TPM_RC_BAD_AUTH;
79
80      }
81
82      if(handle == TPM_RH_LOCKOUT)
```

```
83          {
84              pAssert(gp.lockOutAuthEnabled);
85              gp.lockOutAuthEnabled = FALSE;
86              // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
87              // the lockout auth will be reset at startup.
88              if(gp.lockoutRecovery != 0)
89              {
90                  result = NvIsAvailable();
91                  if(result != TPM_RC_SUCCESS)
92                  {
93                      // No NV access for now. Put the TPM in pending mode.
94                      s_DAPendingOnNV = TRUE;
95                  }
96                  else
97                  {
98                      // Update NV.
99                      NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
100                     g_updateNV = TRUE;
101                 }
102             }
103         }
104         else
105         {
106             if(gp.recoveryTime != 0)
107             {
108                 gp.failedTries++;
109                 result = NvIsAvailable();
110                 if(result != TPM_RC_SUCCESS)
111                 {
112                     // No NV access for now.  Put the TPM in pending mode.
113                     s_DAPendingOnNV = TRUE;
114                 }
115                 else
116                 {
117                     // Record changes to NV.
118                     NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
119                     g_updateNV = TRUE;
120                 }
121             }
122         }
123
124         // Register a DA failure and reset the timers.
125         DARegisterFailure(handle);
126
127         return TPM_RC_AUTH_FAIL;
128     }
```

6.4.3.3 IsSessionBindEntity()

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

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

```
129 static BOOL
130 IsSessionBindEntity(
131     TPM_HANDLE       associatedHandle,  // IN: handle to be authorized
132     SESSION         *session            // IN: associated session
```

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

6.4.3.4 IsPolicySessionRequired()

Checks if a policy session is required for a command. If a command requires DUP or ADMIN role authorization, then the handle that requires that role is the first handle in the command. This simplifies this checking. If a new command is created that requires multiple ADMIN role authorizations, then it will have to be special-cased in this function. A policy session is required if:

a) the command requires the DUP role,

b) the command requires the ADMIN role and the authorized entity is an object and its *adminWithPolicy* bit is SET, or

c) the command requires the ADMIN role and the authorized entity is a permanent handle or an NV Index.

d) The authorized entity is a PCR belonging to a policy group, and has its policy initialized

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

```
149 static BOOL
150 IsPolicySessionRequired(
151     TPM_CC           commandCode,   // IN: command code
152     UINT32           sessionIndex   // IN: session index
153     )
154 {
155     AUTH_ROLE       role = CommandAuthRole(commandCode, sessionIndex);
156     TPM_HT          type = HandleGetType(s_associatedHandles[sessionIndex]);
157
158     if(role == AUTH_DUP)
159         return TRUE;
160
161     if(role == AUTH_ADMIN)
162     {
163         if(type == TPM_HT_TRANSIENT)
164         {
165             OBJECT      *object = ObjectGet(s_associatedHandles[sessionIndex]);
166
167             if(object->publicArea.objectAttributes.adminWithPolicy == CLEAR)
168                 return FALSE;
169         }
170         return TRUE;
171     }
172
```

```
173         if(type == TPM_HT_PCR)
174         {
175             if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
176             {
177                 TPM2B_DIGEST        policy;
178                 TPMI_ALG_HASH       policyAlg;
179                 policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
180                                              &policy);
181                 if(policyAlg != TPM_ALG_NULL)
182                     return TRUE;
183             }
184         }
185         return FALSE;
186     }
```

6.4.3.5 IsAuthValueAvailable()

This function indicates if *authValue* is available and allowed for USER role authorization of an entity.

This function is similar to IsAuthPolicyAvailable() except that it does not check the size of the *authValue* as IsAuthPolicyAvailable() does (a null *authValue* is a valid auth, but a null policy is not a valid policy).

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

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

```
187     static BOOL
188     IsAuthValueAvailable(
189         TPM_HANDLE       handle,        // IN: handle of entity
190         TPM_CC           commandCode,   // IN: commandCode
191         UINT32           sessionIndex   // IN: session index
192         )
193     {
194         BOOL             result = FALSE;
195         // If a policy session is required, the entity can not be authorized by
196         // authValue. However, at this point, the policy session requirement should
197         // already have been checked.
198         pAssert(!IsPolicySessionRequired(commandCode, sessionIndex));
199
200         switch(HandleGetType(handle))
201         {
202             case TPM_HT_PERMANENT:
203                 switch(handle)
204                 {
205                         // At this point hierarchy availability has already been
206                         // checked so primary seed handles are always available here
207                     case TPM_RH_OWNER:
208                     case TPM_RH_ENDORSEMENT:
209                     case TPM_RH_PLATFORM:
210     #ifdef VENDOR_PERMANENT
211                         // This vendor defined handle associated with the
212                         // manufacturer's shared secret
213                     case VENDOR_PERMANENT:
214     #endif
215                         // NullAuth is always available.
216                     case TPM_RH_NULL:
217                          // At the point when authValue availability is checked, control
218                         // path has already passed the DA check so LockOut auth is
219                         // always available here
220                     case TPM_RH_LOCKOUT:
```

```
221
222                     result = TRUE;
223                     break;
224                 default:
225                     // Otherwise authValue is not available.
226                     break;
227             }
228             break;
229         case TPM_HT_TRANSIENT:
230             // A persistent object has already been loaded and the internal
231             // handle changed.
232             {
233                 OBJECT          *object;
234                 object = ObjectGet(handle);
235
236                 // authValue is always available for a sequence object.
237                 if(ObjectIsSequence(object))
238                 {
239                     result =  TRUE;
240                     break;
241                 }
242                 // authValue is available for an object if it has its sensitive
243                 // portion loaded and
244                 //  1. userWithAuth bit is SET, or
245                 //  2. ADMIN role is required
246                 if(   object->attributes.publicOnly == CLEAR
247                    &&     (object->publicArea.objectAttributes.userWithAuth == SET
248                        || (CommandAuthRole(commandCode, sessionIndex) == AUTH_ADMIN
249                            &&  object->publicArea.objectAttributes.adminWithPolicy
250                                == CLEAR)))
251                     result = TRUE;
252             }
253             break;
254         case TPM_HT_NV_INDEX:
255             // NV Index.
256             {
257                 NV_INDEX        nvIndex;
258                 NvGetIndexInfo(handle, &nvIndex);
259                 if(IsWriteOperation(commandCode))
260                 {
261                     if (nvIndex.publicArea.attributes.TPMA_NV_AUTHWRITE == SET)
262                         result = TRUE;
263
264                 }
265                 else
266                 {
267                     if (nvIndex.publicArea.attributes.TPMA_NV_AUTHREAD == SET)
268                         result = TRUE;
269                 }
270             }
271             break;
272         case TPM_HT_PCR:
273             // PCR handle.
274             // authValue is always allowed for PCR
275             result =  TRUE;
276             break;
277         default:
278             // Otherwise, authValue is not available
279             break;
280     }
281     return result;
282 }
```

6.4.3.6 IsAuthPolicyAvailable()

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

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

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

```
283 static BOOL
284 IsAuthPolicyAvailable(
285     TPM_HANDLE       handle,        // IN: handle of entity
286     TPM_CC           commandCode,   // IN: commandCode
287     UINT32           sessionIndex   // IN: session index
288     )
289 {
290     BOOL             result = FALSE;
291     switch(HandleGetType(handle))
292     {
293         case TPM_HT_PERMANENT:
294             switch(handle)
295             {
296                 // At this point hierarchy availability has already been checked.
297                 case TPM_RH_OWNER:
298                     if (gp.ownerPolicy.t.size != 0)
299                         result = TRUE;
300                     break;
301 
302                 case TPM_RH_ENDORSEMENT:
303                     if (gp.endorsementPolicy.t.size != 0)
304                         result = TRUE;
305                     break;
306 
307                 case TPM_RH_PLATFORM:
308                     if (gc.platformPolicy.t.size != 0)
309                         result = TRUE;
310                     break;
311                 case TPM_RH_LOCKOUT:
312                     if(gp.lockoutPolicy.t.size != 0)
313                         result = TRUE;
314                     break;
315                 default:
316                     break;
317             }
318             break;
319         case TPM_HT_TRANSIENT:
320             {
321                 // Object handle.
322                 // An evict object would already have been loaded and given a
323                 // transient object handle by this point.
324                 OBJECT  *object = ObjectGet(handle);
325                 // Policy authorization is not available for an object with only
326                 // public portion loaded.
327                 if(object->attributes.publicOnly == CLEAR)
328                 {
329                     // Policy authorization is always available for an object but
330                     // is never available for a sequence.
331                     if(!ObjectIsSequence(object))
332                         result = TRUE;
333                 }
334                 break;
```

```
335             }
336         case TPM_HT_NV_INDEX:
337             // An NV Index.
338             {
339                 NV_INDEX         nvIndex;
340                 NvGetIndexInfo(handle, &nvIndex);
341                 // If the policy size is not zero, check if policy can be used.
342                 if(nvIndex.publicArea.authPolicy.t.size != 0)
343                 {
344                     // If policy session is required for this handle, always
345                     // uses policy regardless of the attributes bit setting
346                     if(IsPolicySessionRequired(commandCode, sessionIndex))
347                         result = TRUE;
348                     // Otherwise, the presence of the policy depends on the NV
349                     // attributes.
350                     else if(IsWriteOperation(commandCode))
351                     {
352                         if (   nvIndex.publicArea.attributes.TPMA_NV_POLICYWRITE
353                             == SET)
354                             result = TRUE;
355                     }
356                     else
357                     {
358                         if (    nvIndex.publicArea.attributes.TPMA_NV_POLICYREAD
359                             ==  SET)
360                             result = TRUE;
361                     }
362                 }
363             }
364             break;
365         case TPM_HT_PCR:
366             // PCR handle.
367             if(PCRPolicyIsAvailable(handle))
368                 result = TRUE;
369             break;
370         default:
371             break;
372     }
373     return result;
374 }
```

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.

```
375 static void
376 ComputeCpHash(
377     TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
378     TPM_CC           commandCode,       // IN: command code
379     UINT32           handleNum,         // IN: number of handle
380     TPM_HANDLE       handles[],         // IN: array of handle
381     UINT32           parmBufferSize,    // IN: size of input parameter area
382     BYTE            *parmBuffer,        // IN: input parameter area
383     TPM2B_DIGEST    *cpHash,            // OUT: cpHash
384     TPM2B_DIGEST    *nameHash           // OUT: name hash of command
385     )
386 {
387     UINT32           i;
388     HASH_STATE       hashState;
389     TPM2B_NAME       name;
390
```

```
391         // cpHash = hash(commandCode [ || authName1
392         //                           [ || authName2
393         //                           [ || authName 3 ]]]
394         //                           [ || parameters])
395         // A cpHash can contain just a commandCode only if the lone session is
396         // an audit session.
397
398         // Start cpHash.
399         cpHash->t.size = CryptStartHash(hashAlg, &hashState);
400
401         //  Add commandCode.
402         CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
403
404         //  Add authNames for each of the handles.
405         for(i = 0; i < handleNum; i++)
406         {
407             name.t.size = EntityGetName(handles[i], &name.t.name);
408             CryptUpdateDigest2B(&hashState, &name.b);
409         }
410
411         //  Add the parameters.
412         CryptUpdateDigest(&hashState, parmBufferSize, parmBuffer);
413
414         //  Complete the hash.
415         CryptCompleteHash2B(&hashState, &cpHash->b);
416
417         // If the nameHash is needed, compute it here.
418         if(nameHash != NULL)
419         {
420             // Start name hash. hashState may be reused.
421             nameHash->t.size = CryptStartHash(hashAlg, &hashState);
422
423             //  Adding names.
424             for(i = 0; i < handleNum; i++)
425             {
426                 name.t.size = EntityGetName(handles[i], &name.t.name);
427                 CryptUpdateDigest2B(&hashState, &name.b);
428             }
429             //  Complete hash.
430             CryptCompleteHash2B(&hashState, &nameHash->b);
431         }
432         return;
433     }
```

6.4.4.2 CheckPWAuthSession()

This function validates the authorization provided in a PWAP session. It compares the input value to *authValue* of the authorized entity. Argument *sessionIndex* is used to get handles handle of the referenced entities from *s_inputAuthValues*[] and *s_associatedHandles*[].

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

```
434     static TPM_RC
435     CheckPWAuthSession(
436         UINT32           sessionIndex   // IN: index of session to be processed
437         )
438     {
439         TPM2B_AUTH      authValue;
440         TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
441
```

```
442         // Strip trailing zeros from the password.
443         MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
444 
445         // Get the auth value and size.
446         authValue.t.size = EntityGetAuthValue(associatedHandle, &authValue.t.buffer);
447 
448         // Success if the digests are identical.
449         if(Memory2BEqual(&s_inputAuthValues[sessionIndex].b, &authValue.b))
450         {
451             return TPM_RC_SUCCESS;
452         }
453         else                    // if the digests are not identical
454         {
455             // Invoke DA protection if applicable.
456             return IncrementLockout(sessionIndex);
457         }
458     }
```

6.4.4.3 ComputeCommandHMAC()

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

```
459     static void
460     ComputeCommandHMAC(
461         UINT32           sessionIndex,  // IN: index of session to be processed
462         TPM2B_DIGEST    *cpHash,        // IN: cpHash
463         TPM2B_DIGEST    *hmac           // OUT: authorization HMAC
464         )
465     {
466         TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
467         TPM2B_KEY        key;
468         BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
469         BYTE            *buffer;
470         UINT32           marshalSize;
471         HMAC_STATE       hmacState;
472         TPM2B_NONCE     *nonceDecrypt;
473         TPM2B_NONCE     *nonceEncrypt;
474         SESSION         *session;
475         TPM_HT           sessionHandleType =
476                                 HandleGetType(s_sessionHandles[sessionIndex]);
477 
478         nonceDecrypt = NULL;
479         nonceEncrypt = NULL;
480 
481         // Determine if extra nonceTPM values are going to be required.
482         // If this is the first session (sessionIndex = 0) and it is an authorization
483         // session that uses an HMAC, then check if additional session nonces are to be
484         // included.
485         if(   sessionIndex == 0
486            && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
487         {
488             // If there is a decrypt session and if this is not the decrypt session,
489             // then an extra nonce may be needed.
490             if(   s_decryptSessionIndex != UNDEFINED_INDEX
491                && s_decryptSessionIndex != sessionIndex)
492             {
493                 // Will add the nonce for the decrypt session.
494                 SESSION *decryptSession
495                             = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
496                 nonceDecrypt = &decryptSession->nonceTPM;
497             }
498             // Now repeat for the encrypt session.
499             if(   s_encryptSessionIndex != UNDEFINED_INDEX
500                && s_encryptSessionIndex != sessionIndex
```

```
501             && s_encryptSessionIndex != s_decryptSessionIndex)
502         {
503             // Have to have the nonce for the encrypt session.
504             SESSION *encryptSession
505                         = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
506             nonceEncrypt = &encryptSession->nonceTPM;
507         }
508     }
509
510     // Continue with the HMAC processing.
511     session = SessionGet(s_sessionHandles[sessionIndex]);
512
513     // Generate HMAC key.
514     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
515
516     // Check if the session has an associated handle and if the associated entity
517     // is the one to which the session is bound. If not, add the authValue of
518     // this entity to the HMAC key.
519     // If the session is bound to the object or the session is a policy session
520     // with no authValue required, do not include the authValue in the HMAC key.
521     // Note: For a policy session, its isBound attribute is CLEARED.
522
523     // If the session isn't used for authorization, then there is no auth value
524     // to add
525     if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
526     {
527         // used for auth so see if this is a policy session with authValue needed
528         // or an hmac session that is not bound
529         if(         sessionHandleType == TPM_HT_POLICY_SESSION
530                 &&  session->attributes.isAuthValueNeeded == SET
531             ||      sessionHandleType == TPM_HT_HMAC_SESSION
532                 && !IsSessionBindEntity(s_associatedHandles[sessionIndex], session)
533           )
534         {
535             // add the authValue to the HMAC key
536             pAssert((sizeof(AUTH_VALUE) + key.t.size) <= sizeof(key.t.buffer));
537             key.t.size =   key.t.size
538                          + EntityGetAuthValue(s_associatedHandles[sessionIndex],
539                                               (AUTH_VALUE *)&(key.t.buffer[key.t.size]));
540         }
541     }
542
543      // if the HMAC key size is 0, a NULL string HMAC is allowed
544      if(    key.t.size == 0
545          && s_inputAuthValues[sessionIndex].t.size == 0)
546      {
547          hmac->t.size = 0;
548          return;
549      }
550
551     // Start HMAC
552     hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
553
554     //  Add cpHash
555     CryptUpdateDigest2B(&hmacState, &cpHash->b);
556
557     //  Add nonceCaller
558     CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
559
560     //  Add nonceTPM
561     CryptUpdateDigest2B(&hmacState, &session->nonceTPM.b);
562
563     //  If needed, add nonceTPM for decrypt session
564     if(nonceDecrypt != NULL)
565         CryptUpdateDigest2B(&hmacState, &nonceDecrypt->b);
566
```

```
567         //  If needed, add nonceTPM for encrypt session
568         if(nonceEncrypt != NULL)
569             CryptUpdateDigest2B(&hmacState, &nonceEncrypt->b);
570 
571         //  Add sessionAttributes
572         buffer = marshalBuffer;
573         marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
574                                            &buffer, NULL);
575         CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
576 
577         // Complete the HMAC computation
578         CryptCompleteHMAC2B(&hmacState, &hmac->b);
579 
580         return;
581     }
```

6.4.4.4 CheckSessionHMAC()

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

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

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

```
582     static TPM_RC
583     CheckSessionHMAC(
584         UINT32            sessionIndex,  // IN: index of session to be processed
585         TPM2B_DIGEST     *cpHash         // IN: cpHash of the command
586         )
587     {
588         TPM2B_DIGEST        hmac;           // authHMAC for comparing
589 
590         // Compute authHMAC
591         ComputeCommandHMAC(sessionIndex, cpHash, &hmac);
592 
593         // Compare the input HMAC with the authHMAC computed above.
594         if(!Memory2BEqual(&s_inputAuthValues[sessionIndex].b,  &hmac.b))
595         {
596             // If an HMAC session has a failure, invoke the anti-hammering
597             // if it applies to the authorized entity or the session.
598             // Otherwise, just indicate that the authorization is bad.
599             return IncrementLockout(sessionIndex);
600         }
601         return TPM_RC_SUCCESS;
602     }
```

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:

a) compare *policyDigest* in session with *authPolicy* associated with the entity to be authorized;

b) compare timeout if applicable;

c) compare *commandCode* if applicable;

d) compare *cpHash* if applicable; and

e) see if PCR values have changed since computed.

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

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

```
603  static TPM_RC
604  CheckPolicyAuthSession(
605      UINT32            sessionIndex,  // IN: index of session to be processed
606      TPM_CC            commandCode,   // IN: command code
607      TPM2B_DIGEST     *cpHash,        // IN: cpHash using the algorithm of this
608                                       //     session
609      TPM2B_DIGEST     *nameHash       // IN: nameHash using the session algorithm
610      )
611  {
612      TPM_RC            result = TPM_RC_SUCCESS;
613      SESSION          *session;
614      TPM2B_DIGEST      authPolicy;
615      TPMI_ALG_HASH     policyAlg;
616      UINT8             locality;
617
618      // Initialize pointer to the auth session.
619      session = SessionGet(s_sessionHandles[sessionIndex]);
620
621      // If the command is TPM_RC_PolicySecret(), make sure that
622      // either password or authValue is required
623      if(     commandCode == TPM_CC_PolicySecret
624          &&  session->attributes.isPasswordNeeded == CLEAR
625          &&  session->attributes.isAuthValueNeeded == CLEAR)
626          return TPM_RC_MODE;
627
628      // See if the PCR counter for the session is still valid.
629      if( !SessionPCRValueIsCurrent(s_sessionHandles[sessionIndex]) )
630          return TPM_RC_PCR_CHANGED;
631
632      // Get authPolicy.
633      policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
634                                      &authPolicy);
635      // Compare authPolicy.
636      if(!Memory2BEqual(&session->u2.policyDigest.b, &authPolicy.b))
637          return TPM_RC_POLICY_FAIL;
638
639      // Policy is OK so check if the other factors are correct
640
641      // Compare policy hash algorithm.
642      if(policyAlg != session->authHashAlg)
643          return TPM_RC_POLICY_FAIL;
644
645      // Compare timeout.
```

```
646         if(session->timeOut != 0)
647         {
648             // Cannot compare time if clock stop advancing.  An TPM_RC_NV_UNAVAILABLE
649             // or TPM_RC_NV_RATE error may be returned here.
650             result = NvIsAvailable();
651             if(result != TPM_RC_SUCCESS)
652                 return result;
653 
654             if(session->timeOut < go.clock)
655                 return TPM_RC_EXPIRED;
656         }
657 
658         // If command code is provided it must match
659         if(session->commandCode != 0)
660         {
661             if(session->commandCode != commandCode)
662                 return TPM_RC_POLICY_CC;
663         }
664         else
665         {
666             // If command requires a DUP or ADMIN authorization, the session must have
667             // command code set.
668             AUTH_ROLE   role = CommandAuthRole(commandCode, sessionIndex);
669             if(role == AUTH_ADMIN || role == AUTH_DUP)
670                 return TPM_RC_POLICY_FAIL;
671         }
672         // Check command locality.
673         {
674             BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
675             BYTE        *buffer = sessionLocality;
676 
677             // Get existing locality setting in canonical form
678             TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
679 
680             // See if the locality has been set
681             if(sessionLocality[0] != 0)
682             {
683                 // If so, get the current locality
684                 locality = _plat__LocalityGet();
685                 if (locality < 5)
686                 {
687                     if(    ((sessionLocality[0] & (1 << locality)) == 0)
688                         || sessionLocality[0] > 31)
689                         return TPM_RC_LOCALITY;
690                 }
691                 else if (locality > 31)
692                 {
693                     if(sessionLocality[0] != locality)
694                         return TPM_RC_LOCALITY;
695                 }
696                 else
697                 {
698                     // Could throw an assert here but a locality error is just
699                     // as good. It just means that, whatever the locality is, it isn't
700                     // the locality requested so...
701                     return TPM_RC_LOCALITY;
702                 }
703             }
704         } // end of locality check
705 
706         // Check physical presence.
707         if(   session->attributes.isPPRequired == SET
708            && !_plat__PhysicalPresenceAsserted())
709             return TPM_RC_PP;
710 
711         // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
```

```
712         // DUP role for this handle.
713         if(session->u1.cpHash.b.size != 0)
714         {
715             if(session->attributes.iscpHashDefined)
716             {
717                 // Compare cpHash.
718                 if(!Memory2BEqual(&session->u1.cpHash.b, &cpHash->b))
719                     return TPM_RC_POLICY_FAIL;
720             }
721             else
722             {
723                 // Compare nameHash.
724                 // When cpHash is not defined, nameHash is placed in its space.
725                 if(!Memory2BEqual(&session->u1.cpHash.b, &nameHash->b))
726                     return TPM_RC_POLICY_FAIL;
727             }
728         }
729         if(session->attributes.checkNvWritten)
730         {
731             NV_INDEX        nvIndex;
732 
733             // If this is not an NV index, the policy makes no sense so fail it.
734             if(HandleGetType(s_associatedHandles[sessionIndex])!= TPM_HT_NV_INDEX)
735                 return TPM_RC_POLICY_FAIL;
736 
737             // Get the index data
738             NvGetIndexInfo(s_associatedHandles[sessionIndex], &nvIndex);
739 
740             // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
741             if(    (nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
742                != (session->attributes.nvWrittenState == SET))
743                 return TPM_RC_POLICY_FAIL;
744         }
745 
746         return TPM_RC_SUCCESS;
747     }
```

6.4.4.6 RetrieveSessionData()

This function will unmarshal the sessions in the session area of a command. The values are placed in the arrays that are defined at the beginning of this file. The normal unmarshaling errors are possible.

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

```
748     static TPM_RC
749     RetrieveSessionData (
750         TPM_CC           commandCode,   // IN: command code
751         UINT32          *sessionCount,  // OUT: number of sessions found
752         BYTE            *sessionBuffer, // IN: pointer to the session buffer
753         INT32            bufferSize     // IN: size of the session buffer
754         )
755     {
756         int          sessionIndex;
757         int          i;
758         TPM_RC       result;
759         SESSION     *session;
760         TPM_HT       sessionType;
761 
762         s_decryptSessionIndex = UNDEFINED_INDEX;
```

```
763         s_encryptSessionIndex = UNDEFINED_INDEX;
764         s_auditSessionIndex = UNDEFINED_INDEX;
765 
766         for(sessionIndex = 0; bufferSize > 0; sessionIndex++)
767         {
768             // If maximum allowed number of sessions has been parsed, return a size
769             // error with a session number that is larger than the number of allowed
770             // sessions
771             if(sessionIndex == MAX_SESSION_NUM)
772                 return TPM_RC_SIZE + TPM_RC_S + g_rcIndex[sessionIndex+1];
773 
774             // make sure that the associated handle for each session starts out
775             // unassigned
776             s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
777 
778             // First parameter: Session handle.
779             result = TPMI_SH_AUTH_SESSION_Unmarshal(&s_sessionHandles[sessionIndex],
780                                                     &sessionBuffer, &bufferSize, TRUE);
781             if(result != TPM_RC_SUCCESS)
782                 return result + TPM_RC_S + g_rcIndex[sessionIndex];
783 
784             // Second parameter: Nonce.
785             result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
786                                            &sessionBuffer, &bufferSize);
787             if(result != TPM_RC_SUCCESS)
788                 return result + TPM_RC_S + g_rcIndex[sessionIndex];
789 
790             // Third parameter: sessionAttributes.
791             result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
792                                             &sessionBuffer, &bufferSize);
793             if(result != TPM_RC_SUCCESS)
794                 return result + TPM_RC_S + g_rcIndex[sessionIndex];
795 
796             // Fourth parameter: authValue (PW or HMAC).
797             result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
798                                           &sessionBuffer, &bufferSize);
799             if(result != TPM_RC_SUCCESS)
800                 return result + TPM_RC_S + g_rcIndex[sessionIndex];
801 
802             if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
803             {
804                 // A PWAP session needs additional processing.
805                 //     Can't have any attributes set other than continueSession bit
806                 if(   s_attributes[sessionIndex].encrypt
807                    || s_attributes[sessionIndex].decrypt
808                    || s_attributes[sessionIndex].audit
809                    || s_attributes[sessionIndex].auditExclusive
810                    || s_attributes[sessionIndex].auditReset
811                   )
812                     return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
813 
814                 //     The nonce size must be zero.
815                 if(s_nonceCaller[sessionIndex].t.size != 0)
816                     return TPM_RC_NONCE + TPM_RC_S + g_rcIndex[sessionIndex];
817 
818                 continue;
819             }
820             // For not password sessions...
821 
822             // Find out if the session is loaded.
823             if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
824                 return TPM_RC_REFERENCE_S0 + sessionIndex;
825 
826             sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
827             session = SessionGet(s_sessionHandles[sessionIndex]);
828             // Check if the session is an HMAC/policy session.
```

```
829             if(   (   session->attributes.isPolicy == SET
830                    && sessionType == TPM_HT_HMAC_SESSION
831                   )
832                || (   session->attributes.isPolicy == CLEAR
833                    &&  sessionType == TPM_HT_POLICY_SESSION
834                   )
835               )
836                 return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
837
838             // Check that this handle has not previously been used.
839             for(i = 0; i < sessionIndex; i++)
840             {
841                 if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
842                     return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
843             }
844
845             // If the session is used for parameter encryption or audit as well, set
846             // the corresponding indices.
847
848             // First process decrypt.
849             if(s_attributes[sessionIndex].decrypt)
850             {
851                 // Check if the commandCode allows command parameter encryption.
852                 if(DecryptSize(commandCode) == 0)
853                     return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
854
855                 // Encrypt attribute can only appear in one session
856                 if(s_decryptSessionIndex != UNDEFINED_INDEX)
857                     return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
858
859                 // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
860                 if(session->symmetric.algorithm == TPM_ALG_NULL)
861                     return TPM_RC_SYMMETRIC + TPM_RC_S + g_rcIndex[sessionIndex];
862
863                 // All checks passed, so set the index for the session used to decrypt
864                 // a command parameter.
865                 s_decryptSessionIndex = sessionIndex;
866             }
867
868             // Now process encrypt.
869             if(s_attributes[sessionIndex].encrypt)
870             {
871                 // Check if the commandCode allows response parameter encryption.
872                 if(EncryptSize(commandCode) == 0)
873                     return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
874
875                 // Encrypt attribute can only appear in one session.
876                 if(s_encryptSessionIndex != UNDEFINED_INDEX)
877                     return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
878
879                 // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
880                 if(session->symmetric.algorithm == TPM_ALG_NULL)
881                     return TPM_RC_SYMMETRIC + TPM_RC_S + g_rcIndex[sessionIndex];
882
883                 // All checks passed, so set the index for the session used to encrypt
884                 // a response parameter.
885                 s_encryptSessionIndex = sessionIndex;
886             }
887
888             // At last process audit.
889             if(s_attributes[sessionIndex].audit)
890             {
891                 // Audit attribute can only appear in one session.
892                 if(s_auditSessionIndex != UNDEFINED_INDEX)
893                     return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
894
```

```
895             // An audit session can not be policy session.
896             if(   HandleGetType(s_sessionHandles[sessionIndex])
897                == TPM_HT_POLICY_SESSION)
898                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
899
900             // If this is a reset of the audit session, or the first use
901             // of the session as an audit session, it doesn't matter what
902             // the exclusive state is. The session will become exclusive.
903             if(   s_attributes[sessionIndex].auditReset == CLEAR
904                && session->attributes.isAudit == SET)
905             {
906                 // Not first use or reset. If auditExlusive is SET, then this
907                 // session must be the current exclusive session.
908                 if(   s_attributes[sessionIndex].auditExclusive == SET
909                    && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
910                    return TPM_RC_EXCLUSIVE;
911             }
912
913             s_auditSessionIndex = sessionIndex;
914         }
915
916         // Initialize associated handle as undefined. This will be changed when
917         // the handles are processed.
918         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
919
920     }
921
922     // Set the number of sessions found.
923     *sessionCount = sessionIndex;
924     return TPM_RC_SUCCESS;
925 }
```

6.4.4.7 CheckLockedOut()

This function checks to see if the TPM is in lockout. This function should only be called if the entity being checked is subject to DA protection. The TPM is in lockout if the NV is not available and a DA write is pending. Otherwise the TPM is locked out if checking for *lockoutAuth (lockoutAuthCheck* == TRUE) and use of *lockoutAuth* is disabled, or *failedTries >= maxTries*

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

```
926 static TPM_RC
927 CheckLockedOut(
928     BOOL             lockoutAuthCheck   // IN: TRUE if checking is for lockoutAuth
929     )
930 {
931     TPM_RC      result;
932
933     // If NV is unavailable, and current cycle state recorded in NV is not
934     // SHUTDOWN_NONE, refuse to check any authorization because we would
935     // not be able to handle a DA failure.
936     result = NvIsAvailable();
937     if(result != TPM_RC_SUCCESS && gp.orderlyState != SHUTDOWN_NONE)
938         return result;
939
940     // Check if DA info needs to be updated in NV.
941     if(s_DAPendingOnNV)
942     {
```

```
943             // If NV is accessible, ...
944             if(result == TPM_RC_SUCCESS)
945             {
946                 // ... write the pending DA data and proceed.
947                 NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED,
948                                 &gp.lockOutAuthEnabled);
949                 NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
950                 g_updateNV = TRUE;
951                 s_DAPendingOnNV = FALSE;
952             }
953             else
954             {
955                 // Otherwise no authorization can be checked.
956                 return result;
957             }
958         }
959 
960         // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
961         // is disabled...
962         if(lockoutAuthCheck)
963         {
964             if(gp.lockOutAuthEnabled == FALSE)
965                 return TPM_RC_LOCKOUT;
966         }
967         else
968         {
969             // ... or if the number of failed tries has been maxed out.
970             if(gp.failedTries >= gp.maxTries)
971                 return TPM_RC_LOCKOUT;
972         }
973         return TPM_RC_SUCCESS;
974     }
```

6.4.4.8 CheckAuthSession()

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

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

```
975     static TPM_RC
976     CheckAuthSession(
977         TPM_CC            commandCode,   // IN: commandCode
978         UINT32            sessionIndex,  // IN: index of session to be processed
979         TPM2B_DIGEST     *cpHash,        // IN: cpHash
980         TPM2B_DIGEST     *nameHash       // IN: nameHash
```

```
981         )
982     {
983         TPM_RC            result;
984         SESSION          *session = NULL;
985         TPM_HANDLE        sessionHandle = s_sessionHandles[sessionIndex];
986         TPM_HANDLE        associatedHandle = s_associatedHandles[sessionIndex];
987         TPM_HT            sessionHandleType = HandleGetType(sessionHandle);
988 
989         pAssert(sessionHandle != TPM_RH_UNASSIGNED);
990 
991         if(sessionHandle != TPM_RS_PW)
992             session = SessionGet(sessionHandle);
993 
994         pAssert(sessionHandleType != TPM_HT_POLICY_SESSION || session != NULL);
995 
996         // If the authorization session is not a policy session, or if the policy
997         // session requires authorization, then check lockout.
998         if(    sessionHandleType != TPM_HT_POLICY_SESSION
999            || session->attributes.isAuthValueNeeded
1000           || session->attributes.isPasswordNeeded)
1001        {
1002            // See if entity is subject to lockout.
1003            if(!IsDAExempted(associatedHandle))
1004            {
1005                // If NV is unavailable, and current cycle state recorded in NV is not
1006                // SHUTDOWN_NONE, refuse to check any authorization because we would
1007                // not be able to handle a DA failure.
1008                result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1009                if(result != TPM_RC_SUCCESS)
1010                    return result;
1011            }
1012        }
1013 
1014        if(associatedHandle == TPM_RH_PLATFORM)
1015        {
1016            // If the physical presence is required for this command, check for PP
1017            // assertion. If it isn't asserted, no point going any further.
1018            if(   PhysicalPresenceIsRequired(commandCode)
1019               && !_plat__PhysicalPresenceAsserted()
1020              )
1021                return TPM_RC_PP;
1022        }
1023        // If a policy session is required, make sure that it is being used.
1024        if(   IsPolicySessionRequired(commandCode, sessionIndex)
1025           && sessionHandleType != TPM_HT_POLICY_SESSION)
1026            return TPM_RC_AUTH_TYPE;
1027 
1028        // If this is a PW authorization, check it and return.
1029        if(sessionHandle == TPM_RS_PW)
1030        {
1031            if(IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
1032                return CheckPWAuthSession(sessionIndex);
1033            else
1034                return TPM_RC_AUTH_UNAVAILABLE;
1035        }
1036        // If this is a policy session, ...
1037        if(sessionHandleType == TPM_HT_POLICY_SESSION)
1038        {
1039            // ... see if the entity has a policy, ...
1040            if( !IsAuthPolicyAvailable(associatedHandle, commandCode, sessionIndex))
1041                return TPM_RC_AUTH_UNAVAILABLE;
1042            // ... and check the policy session.
1043            result = CheckPolicyAuthSession(sessionIndex, commandCode,
1044                                            cpHash, nameHash);
1045            if (result != TPM_RC_SUCCESS)
1046                return result;
```

```
1047        }
1048        else
1049        {
1050            // For non policy, the entity being accessed must allow authorization
1051            // with an auth value. This is required even if the auth value is not
1052            // going to be used in an HMAC because it is bound.
1053            if(!IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
1054                return TPM_RC_AUTH_UNAVAILABLE;
1055        }
1056        // At this point, the session must be either a policy or an HMAC session.
1057        session = SessionGet(s_sessionHandles[sessionIndex]);
1058
1059        if(     sessionHandleType == TPM_HT_POLICY_SESSION
1060            &&  session->attributes.isPasswordNeeded == SET)
1061        {
1062            // For policy session that requires a password, check it as PWAP session.
1063            return CheckPWAuthSession(sessionIndex);
1064        }
1065        else
1066        {
1067            // For other policy or HMAC sessions, have its HMAC checked.
1068            return CheckSessionHMAC(sessionIndex, cpHash);
1069        }
1070    }
1071    #ifdef  TPM_CC_GetCommandAuditDigest
```

6.4.4.9 CheckCommandAudit()

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

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

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

```
1099                        handles, parmBufferSize, parmBufferStart,
1100                        &s_cpHashForCommandAudit, NULL);
1101        }
1102
1103        return TPM_RC_SUCCESS;
1104    }
1105    #endif
```

6.4.4.10 ParseSessionBuffer()

This function is the entry function for command session processing. It iterates sessions in session area and reports if the required authorization has been properly provided. It also processes audit session and passes the information of encryption sessions to parameter encryption module.

Error Returns	Meaning
various	parsing failure or authorization failure

```
1106    TPM_RC
1107    ParseSessionBuffer(
1108        TPM_CC            commandCode,           // IN: Command code
1109        UINT32            handleNum,             // IN: number of element in handle array
1110        TPM_HANDLE        handles[],             // IN: array of handle
1111        BYTE             *sessionBufferStart,    // IN: start of session buffer
1112        UINT32            sessionBufferSize,     // IN: size of session buffer
1113        BYTE             *parmBufferStart,       // IN: start of parameter buffer
1114        UINT32            parmBufferSize         // IN: size of parameter buffer
1115        )
1116    {
1117        TPM_RC            result;
1118        UINT32            i;
1119        INT32             size = 0;
1120        TPM2B_AUTH        extraKey;
1121        UINT32            sessionIndex;
1122        SESSION          *session;
1123        TPM2B_DIGEST      cpHash;
1124        TPM2B_DIGEST      nameHash;
1125        TPM_ALG_ID        cpHashAlg = TPM_ALG_NULL;  // algID for the last computed
1126                                                     // cpHash
1127
1128        // Check if a command allows any session in its session area.
1129        if(!IsSessionAllowed(commandCode))
1130            return TPM_RC_AUTH_CONTEXT;
1131
1132        // Default-initialization.
1133        s_sessionNum = 0;
1134        cpHash.t.size = 0;
1135
1136        result = RetrieveSessionData(commandCode, &s_sessionNum,
1137                                     sessionBufferStart, sessionBufferSize);
1138        if(result != TPM_RC_SUCCESS)
1139            return result;
1140
1141        // There is no command in the TPM spec that has more handles than
1142        // MAX_SESSION_NUM.
1143        pAssert(handleNum <= MAX_SESSION_NUM);
1144
1145        // Associate the session with an authorization handle.
1146        for(i = 0; i < handleNum; i++)
1147        {
1148            if(CommandAuthRole(commandCode, i) != AUTH_NONE)
1149            {
1150                // If the received session number is less than the number of handle
1151                // that requires authorization, an error should be returned.
```

```
1152                // Note: for all the TPM 2.0 commands, handles requiring
1153                // authorization come first in a command input.
1154                if(i > (s_sessionNum - 1))
1155                    return TPM_RC_AUTH_MISSING;
1156
1157                // Record the handle associated with the authorization session
1158                s_associatedHandles[i] = handles[i];
1159            }
1160        }
1161
1162        // Consistency checks are done first to avoid auth failure when the command
1163        // will not be executed anyway.
1164        for(sessionIndex = 0; sessionIndex < s_sessionNum; sessionIndex++)
1165        {
1166            // PW session must be an authorization session
1167            if(s_sessionHandles[sessionIndex] == TPM_RS_PW )
1168            {
1169                if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1170                    return TPM_RC_HANDLE + g_rcIndex[sessionIndex];
1171            }
1172            else
1173            {
1174                session = SessionGet(s_sessionHandles[sessionIndex]);
1175
1176                // A trial session can not appear in session area, because it cannot
1177                // be used for authorization, audit or encrypt/decrypt.
1178                if(session->attributes.isTrialPolicy == SET)
1179                    return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
1180
1181                // See if the session is bound to a DA protected entity
1182                // NOTE: Since a policy session is never bound, a policy is still
1183                // usable even if the object is DA protected and the TPM is in
1184                // lockout.
1185                if(session->attributes.isDaBound == SET)
1186                {
1187                    result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1188                    if(result != TPM_RC_SUCCESS)
1189                        return result;
1190                }
1191                // If the current cpHash is the right one, don't re-compute.
1192                if(cpHashAlg != session->authHashAlg)   // different so compute
1193                {
1194                    cpHashAlg = session->authHashAlg;   // save this new algID
1195                    ComputeCpHash(session->authHashAlg, commandCode, handleNum,
1196                                  handles, parmBufferSize, parmBufferStart,
1197                                  &cpHash, &nameHash);
1198                }
1199                // If this session is for auditing, save the cpHash.
1200                if(s_attributes[sessionIndex].audit)
1201                    s_cpHashForAudit = cpHash;
1202            }
1203
1204            // if the session has an associated handle, check the auth
1205            if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1206            {
1207                result = CheckAuthSession(commandCode, sessionIndex,
1208                                          &cpHash, &nameHash);
1209                if(result != TPM_RC_SUCCESS)
1210                    return RcSafeAddToResult(result,
1211                                             TPM_RC_S + g_rcIndex[sessionIndex]);
1212            }
1213            else
1214            {
1215                // a session that is not for authorization must either be encrypt,
1216                // decrypt, or audit
1217                if(     s_attributes[sessionIndex].audit == CLEAR
```

```
1218                &&  s_attributes[sessionIndex].encrypt == CLEAR
1219                &&  s_attributes[sessionIndex].decrypt == CLEAR)
1220                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
1221
1222            // check HMAC for encrypt/decrypt/audit only sessions
1223            result =  CheckSessionHMAC(sessionIndex, &cpHash);
1224            if(result != TPM_RC_SUCCESS)
1225                return RcSafeAddToResult(result,
1226                                         TPM_RC_S + g_rcIndex[sessionIndex]);
1227        }
1228    }
1229
1230  #ifdef  TPM_CC_GetCommandAuditDigest
1231    // Check if the command should be audited.
1232    result = CheckCommandAudit(commandCode, handleNum, handles,
1233                               parmBufferStart, parmBufferSize);
1234    if(result != TPM_RC_SUCCESS)
1235        return result;              // No session number to reference
1236  #endif
1237
1238    // Decrypt the first parameter if applicable. This should be the last operation
1239    // in session processing.
1240    // If the encrypt session is associated with a handle and the handle's
1241    // authValue is available, then authValue is concatenated with sessionAuth to
1242    // generate encryption key, no matter if the handle is the session bound entity
1243    // or not.
1244    if(s_decryptSessionIndex != UNDEFINED_INDEX)
1245    {
1246        // Get size of the leading size field in decrypt parameter
1247        if(   s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED
1248           && IsAuthValueAvailable(s_associatedHandles[s_decryptSessionIndex],
1249                                   commandCode,
1250                                   s_decryptSessionIndex)
1251          )
1252        {
1253            extraKey.b.size=
1254                EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1255                                   &extraKey.t.buffer);
1256        }
1257        else
1258        {
1259            extraKey.b.size = 0;
1260        }
1261        size = DecryptSize(commandCode);
1262        result = CryptParameterDecryption(
1263                     s_sessionHandles[s_decryptSessionIndex],
1264                     &s_nonceCaller[s_decryptSessionIndex].b,
1265                     parmBufferSize, (UINT16)size,
1266                     &extraKey,
1267                     parmBufferStart);
1268        if(result != TPM_RC_SUCCESS)
1269            return RcSafeAddToResult(result,
1270                                     TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1271    }
1272
1273    return TPM_RC_SUCCESS;
1274  }
```

6.4.4.11 CheckAuthNoSession()

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

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

```
1275    TPM_RC
1276    CheckAuthNoSession(
1277        TPM_CC            commandCode,        // IN: Command Code
1278        UINT32            handleNum,          // IN: number of handles in command
1279        TPM_HANDLE        handles[],          // IN: array of handle
1280        BYTE             *parmBufferStart,    // IN: start of parameter buffer
1281        UINT32            parmBufferSize      // IN: size of parameter buffer
1282        )
1283    {
1284        UINT32 i;
1285        TPM_RC            result = TPM_RC_SUCCESS;
1286
1287        // Check if the commandCode requires authorization
1288        for(i = 0; i < handleNum; i++)
1289        {
1290            if(CommandAuthRole(commandCode, i) != AUTH_NONE)
1291                return TPM_RC_AUTH_MISSING;
1292        }
1293
1294    #ifdef  TPM_CC_GetCommandAuditDigest
1295        // Check if the command should be audited.
1296        result = CheckCommandAudit(commandCode, handleNum, handles,
1297                                   parmBufferStart, parmBufferSize);
1298        if(result != TPM_RC_SUCCESS) return result;
1299    #endif
1300
1301        // Initialize number of sessions to be 0
1302        s_sessionNum = 0;
1303
1304        return TPM_RC_SUCCESS;
1305    }
```

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.

```
1306    static void
1307    ComputeRpHash(
1308        TPM_ALG_ID        hashAlg,            // IN: hash algorithm to compute rpHash
1309        TPM_CC            commandCode,        // IN: commandCode
1310        UINT32            resParmBufferSize,  // IN: size of response parameter buffer
1311        BYTE             *resParmBuffer,      // IN: response parameter buffer
1312        TPM2B_DIGEST     *rpHash              // OUT: rpHash
1313        )
1314    {
1315        // The command result in rpHash is always TPM_RC_SUCCESS.
1316        TPM_RC       responseCode = TPM_RC_SUCCESS;
1317        HASH_STATE   hashState;
1318
1319        //   rpHash := hash(responseCode || commandCode || parameters)
```

```
1320
1321        // Initiate hash creation.
1322        rpHash->t.size = CryptStartHash(hashAlg, &hashState);
1323
1324        // Add hash constituents.
1325        CryptUpdateDigestInt(&hashState, sizeof(TPM_RC), &responseCode);
1326        CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
1327        CryptUpdateDigest(&hashState, resParmBufferSize, resParmBuffer);
1328
1329        // Complete hash computation.
1330        CryptCompleteHash2B(&hashState, &rpHash->b);
1331
1332        return;
1333    }
```

6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```
1334    static void
1335    InitAuditSession(
1336        SESSION         *session        // session to be initialized
1337        )
1338    {
1339        // Mark session as an audit session.
1340        session->attributes.isAudit = SET;
1341
1342        // Audit session can not be bound.
1343        session->attributes.isBound = CLEAR;
1344
1345        // Size of the audit log is the size of session hash algorithm digest.
1346        session->u2.auditDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
1347
1348        // Set the original digest value to be 0.
1349        MemorySet(&session->u2.auditDigest.t.buffer,
1350                  0,
1351                  session->u2.auditDigest.t.size);
1352
1353        return;
1354    }
```

6.4.5.4 Audit()

This function updates the audit digest in an audit session.

```
1355    static void
1356    Audit(
1357        SESSION         *auditSession,      // IN: loaded audit session
1358        TPM_CC           commandCode,       // IN: commandCode
1359        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1360        BYTE            *resParmBuffer      // IN: response parameter buffer
1361        )
1362    {
1363        TPM2B_DIGEST     rpHash;            // rpHash for response
1364        HASH_STATE       hashState;
1365
1366        // Compute rpHash
1367        ComputeRpHash(auditSession->authHashAlg,
1368                      commandCode,
1369                      resParmBufferSize,
1370                      resParmBuffer,
1371                      &rpHash);
1372
```

```
1373        // auditDigestnew :=  hash (auditDigestold || cpHash || rpHash)
1374
1375        // Start hash computation.
1376        CryptStartHash(auditSession->authHashAlg, &hashState);
1377
1378        // Add old digest.
1379        CryptUpdateDigest2B(&hashState, &auditSession->u2.auditDigest.b);
1380
1381        // Add cpHash and rpHash.
1382        CryptUpdateDigest2B(&hashState, &s_cpHashForAudit.b);
1383        CryptUpdateDigest2B(&hashState, &rpHash.b);
1384
1385        // Finalize the hash.
1386        CryptCompleteHash2B(&hashState, &auditSession->u2.auditDigest.b);
1387
1388        return;
1389    }
1390    #ifdef  TPM_CC_GetCommandAuditDigest
```

6.4.5.5 CommandAudit()

This function updates the command audit digest.

```
1391    static void
1392    CommandAudit(
1393        TPM_CC            commandCode,       // IN: commandCode
1394        UINT32            resParmBufferSize, // IN: size of response parameter buffer
1395        BYTE             *resParmBuffer      // IN: response parameter buffer
1396        )
1397    {
1398        if(CommandAuditIsRequired(commandCode))
1399        {
1400            TPM2B_DIGEST    rpHash;        // rpHash for response
1401            HASH_STATE      hashState;
1402
1403            // Compute rpHash.
1404            ComputeRpHash(gp.auditHashAlg, commandCode, resParmBufferSize,
1405                          resParmBuffer, &rpHash);
1406
1407            // If the digest.size is one, it indicates the special case of changing
1408            // the audit hash algorithm. For this case, no audit is done on exit.
1409            // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1410            // force an update to the NV on exit so that the change in digest will
1411            // be recorded. So, it is safe to exit here without setting any flags
1412            // because the digest change will be written to NV when this code exits.
1413            if(gr.commandAuditDigest.t.size == 1)
1414            {
1415                gr.commandAuditDigest.t.size = 0;
1416                return;
1417            }
1418
1419            // If the digest size is zero, need to start a new digest and increment
1420            // the audit counter.
1421            if(gr.commandAuditDigest.t.size == 0)
1422            {
1423                gr.commandAuditDigest.t.size = CryptGetHashDigestSize(gp.auditHashAlg);
1424                MemorySet(gr.commandAuditDigest.t.buffer,
1425                          0,
1426                          gr.commandAuditDigest.t.size);
1427
1428                // Bump the counter and save its value to NV.
1429                gp.auditCounter++;
1430                NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
1431                g_updateNV = TRUE;
```

```
1432            }
1433
1434            // auditDigestnew :=  hash (auditDigestold || cpHash || rpHash)
1435
1436            //  Start hash computation.
1437            CryptStartHash(gp.auditHashAlg, &hashState);
1438
1439            //  Add old digest.
1440            CryptUpdateDigest2B(&hashState, &gr.commandAuditDigest.b);
1441
1442            //  Add cpHash
1443            CryptUpdateDigest2B(&hashState, &s_cpHashForCommandAudit.b);
1444
1445            //  Add rpHash
1446            CryptUpdateDigest2B(&hashState, &rpHash.b);
1447
1448            //  Finalize the hash.
1449            CryptCompleteHash2B(&hashState, &gr.commandAuditDigest.b);
1450        }
1451        return;
1452    }
1453    #endif
```

6.4.5.6 UpdateAuditSessionStatus()

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

a) initializes the session as audit session and sets it to be exclusive if this is the first time it is used for audit or audit reset was requested;

b) reports exclusive audit session;

c) extends audit log; and

d) clears exclusive audit session if no audit session found in the command.

```
1454    static void
1455    UpdateAuditSessionStatus(
1456        TPM_CC           commandCode,       // IN: commandCode
1457        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1458        BYTE            *resParmBuffer      // IN: response parameter buffer
1459        )
1460    {
1461        UINT32           i;
1462        TPM_HANDLE       auditSession = TPM_RH_UNASSIGNED;
1463
1464        // Iterate through sessions
1465        for (i = 0; i < s_sessionNum; i++)
1466        {
1467            SESSION     *session;
1468
1469            // PW session do not have a loaded session and can not be an audit
1470            // session either.  Skip it.
1471            if(s_sessionHandles[i] == TPM_RS_PW) continue;
1472
1473            session = SessionGet(s_sessionHandles[i]);
1474
1475            // If a session is used for audit
1476            if(s_attributes[i].audit == SET)
1477            {
1478                // An audit session has been found
1479                auditSession = s_sessionHandles[i];
1480
1481                // If the session has not been an audit session yet, or
1482                // the auditSetting bits indicate a reset, initialize it and set
```

```
1483                // it to be the exclusive session
1484                if(   session->attributes.isAudit == CLEAR
1485                   || s_attributes[i].auditReset == SET
1486                  )
1487                {
1488                    InitAuditSession(session);
1489                    g_exclusiveAuditSession = auditSession;
1490                }
1491                else
1492                {
1493                    // Check if the audit session is the current exclusive audit
1494                    // session and, if not, clear previous exclusive audit session.
1495                    if(g_exclusiveAuditSession != auditSession)
1496                        g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1497                }
1498
1499                // Report audit session exclusivity.
1500                if(g_exclusiveAuditSession == auditSession)
1501                {
1502                    s_attributes[i].auditExclusive = SET;
1503                }
1504                else
1505                {
1506                    s_attributes[i].auditExclusive = CLEAR;
1507                }
1508
1509                // Extend audit log.
1510                Audit(session, commandCode, resParmBufferSize, resParmBuffer);
1511            }
1512        }
1513
1514        // If no audit session is found in the command, and the command allows
1515        // a session then, clear the current exclusive
1516        // audit session.
1517        if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(commandCode))
1518        {
1519            g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1520        }
1521
1522        return;
1523    }
```

6.4.5.7 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```
1524    static void
1525    ComputeResponseHMAC(
1526        UINT32           sessionIndex,      // IN: session index to be processed
1527        SESSION         *session,           // IN: loaded session
1528        TPM_CC           commandCode,       // IN: commandCode
1529        TPM2B_NONCE     *nonceTPM,          // IN: nonceTPM
1530        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1531        BYTE            *resParmBuffer,     // IN: response parameter buffer
1532        TPM2B_DIGEST    *hmac               // OUT: authHMAC
1533        )
1534    {
1535        TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1536        TPM2B_KEY        key;       // HMAC key
1537        BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
1538        BYTE            *buffer;
1539        UINT32           marshalSize;
1540        HMAC_STATE       hmacState;
1541        TPM2B_DIGEST     rp_hash;
```

```
1542
1543        // Compute rpHash.
1544        ComputeRpHash(session->authHashAlg, commandCode, resParmBufferSize,
1545                      resParmBuffer, &rp_hash);
1546
1547        // Generate HMAC key
1548        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1549
1550        // Check if the session has an associated handle and the associated entity is
1551        // the one that the session is bound to.
1552        // If not bound, add the authValue of this entity to the HMAC key.
1553        if(   s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED
1554           &&    !( HandleGetType(s_sessionHandles[sessionIndex])
1555                    == TPM_HT_POLICY_SESSION
1556              &&   session->attributes.isAuthValueNeeded == CLEAR)
1557           && !session->attributes.requestWasBound)
1558        {
1559            pAssert((sizeof(AUTH_VALUE) + key.t.size) <= sizeof(key.t.buffer));
1560            key.t.size = key.t.size +
1561                         EntityGetAuthValue(s_associatedHandles[sessionIndex],
1562                                            (AUTH_VALUE *)&key.t.buffer[key.t.size]);
1563        }
1564
1565        // if the HMAC key size for a policy session is 0, the response HMAC is
1566        // computed according to the input HMAC
1567        if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1568            && key.t.size == 0
1569            && s_inputAuthValues[sessionIndex].t.size == 0)
1570        {
1571            hmac->t.size = 0;
1572            return;
1573        }
1574
1575        // Start HMAC computation.
1576        hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
1577
1578        // Add hash components.
1579        CryptUpdateDigest2B(&hmacState, &rp_hash.b);
1580        CryptUpdateDigest2B(&hmacState, &nonceTPM->b);
1581        CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
1582
1583        // Add session attributes.
1584        buffer = marshalBuffer;
1585        marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1586        CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
1587
1588        // Finalize HMAC.
1589        CryptCompleteHMAC2B(&hmacState, &hmac->b);
1590
1591        return;
1592    }
```

6.4.5.8 BuildSingleResponseAuth()

Function to compute response for an authorization session.

```
1593    static void
1594    BuildSingleResponseAuth(
1595        UINT32           sessionIndex,      // IN: session index to be processed
1596        TPM_CC           commandCode,       // IN: commandCode
1597        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1598        BYTE            *resParmBuffer,     // IN: response parameter buffer
1599        TPM2B_AUTH      *auth               // OUT: authHMAC
1600        )
```

```
{
    // For password authorization, field is empty.
    if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
    {
        auth->t.size = 0;
    }
    else
    {
        // Fill in policy/HMAC based session response.
        SESSION     *session = SessionGet(s_sessionHandles[sessionIndex]);

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

6.4.5.10 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time, and

b) Clears a policy session since nonce is rolling.

```
1654  static void
1655  UpdateInternalSession(
1656      void
1657      )
1658  {
1659      UINT32      i;
1660      for(i = 0; i < s_sessionNum; i++)
1661      {
1662          // For PW session, no update.
1663          if(s_sessionHandles[i] == TPM_RS_PW) continue;
1664
1665          if(s_attributes[i].continueSession == CLEAR)
1666          {
1667              // Close internal session.
1668              SessionFlush(s_sessionHandles[i]);
1669          }
1670          else
1671          {
1672              // If nonce is rolling in a policy session, the policy related data
1673              // will be re-initialized.
1674              if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION)
1675              {
1676                  SESSION     *session = SessionGet(s_sessionHandles[i]);
1677
1678                  // When the nonce rolls it starts a new timing interval for the
1679                  // policy session.
1680                  SessionResetPolicyData(session);
1681                  session->startTime = go.clock;
1682              }
1683          }
1684      }
1685      return;
1686  }
```

6.4.5.11 BuildResponseSession()

Function to build Session buffer in a response.

```
1687  void
1688  BuildResponseSession(
1689      TPM_ST           tag,           // IN: tag
1690      TPM_CC           commandCode,   // IN: commandCode
1691      UINT32           resHandleSize, // IN: size of response handle buffer
1692      UINT32           resParmSize,   // IN: size of response parameter buffer
1693      UINT32          *resSessionSize // OUT: response session area
1694      )
1695  {
1696      BYTE            *resParmBuffer;
1697      TPM2B_NONCE  responseNonces[MAX_SESSION_NUM];
1698
1699      // Compute response parameter buffer start.
1700      resParmBuffer = MemoryGetResponseBuffer(commandCode) + sizeof(TPM_ST) +
1701                      sizeof(UINT32) + sizeof(TPM_RC) + resHandleSize;
1702
1703      // For TPM_ST_SESSIONS, there is parameterSize field.
1704      if(tag == TPM_ST_SESSIONS)
1705          resParmBuffer += sizeof(UINT32);
1706
1707      // Session nonce should be updated before parameter encryption
1708      if(tag == TPM_ST_SESSIONS)
```

```
1709        {
1710            UpdateTPMNonce(MAX_SESSION_NUM, responseNonces);
1711
1712            // Encrypt first parameter if applicable. Parameter encryption should
1713            // happen after nonce update and before any rpHash is computed.
1714            // If the encrypt session is associated with a handle, the authValue of
1715            // this handle will be concatenated with sessionAuth to generate
1716            // encryption key, no matter if the handle is the session bound entity
1717            // or not. The authValue is added to sessionAuth only when the authValue
1718            // is available.
1719            if(s_encryptSessionIndex != UNDEFINED_INDEX)
1720            {
1721                UINT32          size;
1722                TPM2B_AUTH      extraKey;
1723
1724                // Get size of the leading size field
1725                if(   s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED
1726                   && IsAuthValueAvailable(s_associatedHandles[s_encryptSessionIndex],
1727                                           commandCode, s_encryptSessionIndex)
1728                  )
1729                {
1730                    extraKey.b.size =
1731                        EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1732                                           &extraKey.t.buffer);
1733                }
1734                else
1735                {
1736                    extraKey.b.size = 0;
1737                }
1738                size = EncryptSize(commandCode);
1739                CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1740                                         &s_nonceCaller[s_encryptSessionIndex].b,
1741                                         (UINT16)size,
1742                                         &extraKey,
1743                                         resParmBuffer);
1744
1745            }
1746
1747        }
1748        // Audit session should be updated first regardless of the tag.
1749        // A command with no session may trigger a change of the exclusivity state.
1750        UpdateAuditSessionStatus(commandCode, resParmSize, resParmBuffer);
1751
1752        // Audit command.
1753        CommandAudit(commandCode, resParmSize, resParmBuffer);
1754
1755        // Process command with sessions.
1756        if(tag == TPM_ST_SESSIONS)
1757        {
1758            UINT32           i;
1759            BYTE            *buffer;
1760            TPM2B_DIGEST     responseAuths[MAX_SESSION_NUM];
1761
1762            pAssert(s_sessionNum > 0);
1763
1764            // Iterate over each session in the command session area, and create
1765            // corresponding sessions for response.
1766            for(i = 0; i < s_sessionNum; i++)
1767            {
1768                BuildSingleResponseAuth(
1769                                        i,
1770                                        commandCode,
1771                                        resParmSize,
1772                                        resParmBuffer,
1773                                        &responseAuths[i]);
1774                // Make sure that continueSession is SET on any Password session.
```

```
1775                // This makes it marginally easier for the management software
1776                // to keep track of the closed sessions.
1777                if(   s_attributes[i].continueSession == CLEAR
1778                   && s_sessionHandles[i] == TPM_RS_PW)
1779                {
1780                    s_attributes[i].continueSession = SET;
1781                }
1782            }
1783
1784            // Assemble Response Sessions.
1785            *resSessionSize = 0;
1786            buffer = resParmBuffer + resParmSize;
1787            for(i = 0; i < s_sessionNum; i++)
1788            {
1789                *resSessionSize += TPM2B_NONCE_Marshal(&responseNonces[i],
1790                                                      &buffer, NULL);
1791                *resSessionSize += TPMA_SESSION_Marshal(&s_attributes[i],
1792                                                       &buffer, NULL);
1793                *resSessionSize += TPM2B_DIGEST_Marshal(&responseAuths[i],
1794                                                       &buffer, NULL);
1795            }
1796
1797            // Update internal sessions after completing response buffer computation.
1798            UpdateInternalSession();
1799        }
1800        else
1801        {
1802            // Process command with no session.
1803            *resSessionSize = 0;
1804        }
1805
1806        return;
1807    }
```

7 Command Support Functions

7.1 Introduction

This clause contains support routines that are called by the command action code in TPM 2.0 Part 3. The functions are grouped by the command group that is supported by the functions.

7.2 Attestation Command Support (Attest_spt.c)

7.2.1 Includes

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

7.2.2 Functions

7.2.2.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

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

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

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

    // Magic number
    attest->magic = TPM_GENERATED_VALUE;

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

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

7.2.2.2 SignAttestInfo()

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

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

```
TPM_RC
SignAttestInfo(
    TPMI_DH_OBJECT       signHandle,        // IN: handle of sign object
    TPMT_SIG_SCHEME     *scheme,            // IN: sign scheme
    TPMS_ATTEST         *certifyInfo,       // IN: the data to be signed
    TPM2B_DATA          *qualifyingData,    // IN: extra data for the signing proce
    TPM2B_ATTEST        *attest,            // OUT: marshaled attest blob to be
                                            //     signed
    TPMT_SIGNATURE      *signature          // OUT: signature
    )
{
    TPM_RC                  result;
    TPMI_ALG_HASH           hashAlg;
    BYTE                    *buffer;
    HASH_STATE              hashState;
    TPM2B_DIGEST            digest;

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

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

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

        // If there is qualifying data, need to rehash the the data
        // hash(qualifyingData || hash(attestationData))
        if(qualifyingData->t.size != 0)
        {
            CryptStartHash(hashAlg, &hashState);
            CryptUpdateDigest(&hashState,
                              qualifyingData->t.size,
                              qualifyingData->t.buffer);
            CryptUpdateDigest(&hashState, digest.t.size, digest.t.buffer);
            CryptCompleteHash2B(&hashState, &digest.b);
```

```
145             }
146
147             // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
148             // TPM_RC_ATTRIBUTES error may be returned at this point
149             return CryptSign(signHandle,
150                              scheme,
151                              &digest,
152                              signature);
153         }
154
155         return TPM_RC_SUCCESS;
156     }
```

7.3 Context Management Command Support (Context_spt.c)

7.3.1 Includes

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

7.3.2 Functions

7.3.2.1 ComputeContextProtectionKey()

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

```
3   void
4   ComputeContextProtectionKey(
5       TPMS_CONTEXT    *contextBlob,   // IN: context blob
6       TPM2B_SYM_KEY   *symKey,        // OUT: the symmetric key
7       TPM2B_IV        *iv             // OUT: the IV.
8       )
9   {
10      UINT16           symKeyBits;    // number of bits in the parent's
11                                      //   symmetric key
12      TPM2B_AUTH      *proof = NULL;  // the proof value to use. Is null for
13                                      //   everything but a primary object in
14                                      //   the Endorsement Hierarchy
15
16      BYTE             kdfResult[sizeof(TPMU_HA) * 2];// Value produced by the KDF
17
18      TPM2B_DATA       sequence2B, handle2B;
19
20      // Get proof value
21      proof = HierarchyGetProof(contextBlob->hierarchy);
22
23      // Get sequence value in 2B format
24      sequence2B.t.size = sizeof(contextBlob->sequence);
25      MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence,
26                 sizeof(contextBlob->sequence),
27                 sizeof(sequence2B.t.buffer));
28
29      // Get handle value in 2B format
30      handle2B.t.size = sizeof(contextBlob->savedHandle);
31      MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle,
32                 sizeof(contextBlob->savedHandle),
33                 sizeof(handle2B.t.buffer));
34
35      // Get the symmetric encryption key size
36      symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
```

```
37      symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
38      // Get the size of the IV for the algorithm
39      iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);
40
41      // KDFa to generate symmetric key and IV value
42      KDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, "CONTEXT", &sequence2B.b,
43           &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL);
44
45      // Copy part of the returned value as the key
46      MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size,
47                 sizeof(symKey->t.buffer));
48
49      // Copy the rest as the IV
50      MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size,
51                 sizeof(iv->t.buffer));
52
53      return;
54  }
```

7.3.2.2 ComputeContextIntegrity()

Generate the integrity hash for a context It is used by TPM2_ContextSave() to create an integrity hash and by TPM2_ContextLoad() to compare an integrity hash

```
55  void
56  ComputeContextIntegrity(
57      TPMS_CONTEXT    *contextBlob,   // IN: context blob
58      TPM2B_DIGEST    *integrity      // OUT: integrity
59      )
60  {
61      HMAC_STATE          hmacState;
62      TPM2B_AUTH          *proof;
63      UINT16              integritySize;
64
65      // Get proof value
66      proof = HierarchyGetProof(contextBlob->hierarchy);
67
68      // Start HMAC
69      integrity->t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
70                                           &proof->b, &hmacState);
71
72      // Compute integrity size at the beginning of context blob
73      integritySize = sizeof(integrity->t.size) + integrity->t.size;
74
75      // Adding total reset counter so that the context cannot be
76      // used after a TPM Reset
77      CryptUpdateDigestInt(&hmacState, sizeof(gp.totalResetCount),
78                           &gp.totalResetCount);
79
80      // If this is a ST_CLEAR object, add the clear count
81      // so that this contest cannot be loaded after a TPM Restart
82      if(contextBlob->savedHandle == 0x80000002)
83          CryptUpdateDigestInt(&hmacState, sizeof(gr.clearCount), &gr.clearCount);
84
85      // Adding sequence number to the HMAC to make sure that it doesn't
86      // get changed
87      CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->sequence),
88                           &contextBlob->sequence);
89
90      // Protect the handle
91      CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->savedHandle),
92                           &contextBlob->savedHandle);
93
94      // Adding sensitive contextData, skip the leading integrity area
```

```
95      CryptUpdateDigest(&hmacState, contextBlob->contextBlob.t.size - integritySize,
96                        contextBlob->contextBlob.t.buffer + integritySize);
97
98      // Complete HMAC
99      CryptCompleteHMAC2B(&hmacState, &integrity->b);
100
101     return;
102 }
```

7.3.2.3 SequenceDataImportExport()

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

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

7.4 Policy Command Support (Policy_spt.c)

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

7.4.1 PolicyParameterChecks()

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

```
6   TPM_RC
7   PolicyParameterChecks(
8       SESSION         *session,
9       UINT64           authTimeout,
10      TPM2B_DIGEST    *cpHashA,
11      TPM2B_NONCE     *nonce,
12      TPM_RC           nonceParameterNumber,
13      TPM_RC           cpHashParameterNumber,
14      TPM_RC           expirationParameterNumber
15      )
16  {
17      TPM_RC           result;
18
19      // Validate that input nonceTPM is correct if present
20      if(nonce != NULL && nonce->t.size != 0)
```

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

7.4.2 PolicyContextUpdate()

Update policy hash Update the *policyDigest* in policy session by extending *policyRef* and *objectName* to it. This will also update the *cpHash* if it is present.

```
60  void
61  PolicyContextUpdate(
62      TPM_CC           commandCode,   // IN: command code
63      TPM2B_NAME      *name,          // IN: name of entity
64      TPM2B_NONCE     *ref,           // IN: the reference data
65      TPM2B_DIGEST    *cpHash,        // IN: the cpHash (optional)
66      UINT64           policyTimeout,
67      SESSION         *session        // IN/OUT: policy session to be updated
68      )
69  {
70      HASH_STATE           hashState;
71      UINT16               policyDigestSize;
72
73      // Start hash
74      policyDigestSize = CryptStartHash(session->authHashAlg, &hashState);
75
76      // policyDigest size should always be the digest size of session hash algorithm.
77      pAssert(session->u2.policyDigest.t.size == policyDigestSize);
78
```

```
79      // add old digest
80      CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);
81
82      // add commandCode
83      CryptUpdateDigestInt(&hashState, sizeof(commandCode), &commandCode);
84
85      // add name if applicable
86      if(name != NULL)
87          CryptUpdateDigest2B(&hashState, &name->b);
88
89      // Complete the digest and get the results
90      CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);
91
92      // Start second hash computation
93      CryptStartHash(session->authHashAlg, &hashState);
94
95      // add policyDigest
96      CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);
97
98      // add policyRef
99      if(ref != NULL)
100         CryptUpdateDigest2B(&hashState, &ref->b);
101
102     // Complete second digest
103     CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);
104
105     // Deal with the cpHash. If the cpHash value is present
106     // then it would have already been checked to make sure that
107     // it is compatible with the current value so all we need
108     // to do here is copy it and set the iscoHashDefined attribute
109     if(cpHash != NULL && cpHash->t.size != 0)
110     {
111         session->u1.cpHash = *cpHash;
112         session->attributes.iscpHashDefined = SET;
113     }
114
115     // update the timeout if it is specified
116     if(policyTimeout!= 0)
117     {
118     // If the timeout has not been set, then set it to the new value
119         if(session->timeOut == 0)
120             session->timeOut = policyTimeout;
121         else if(session->timeOut > policyTimeout)
122             session->timeOut =  policyTimeout;
123     }
124     return;
125 }
```

7.5 NV Command Support (NV_spt.c)

7.5.1 Includes

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

7.5.2 Fuctions

7.5.2.1 NvReadAccessChecks()

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

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

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

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

// This check may be done before doing authorization checks as is done in this
// version of the reference code. If not done there, then uncomment the next
// three lines.
//    // If data is read locked, returns an error
//    if(nvIndex.publicArea.attributes.TPMA_NV_READLOCKED == SET)
//        return TPM_RC_NV_LOCKED;

    // If the authorization was provided by the owner or platform, then check
    // that the attributes allow the read.  If the authorization handle
    // is the same as the index, then the checks were made when the authorization
    // was checked..
    if(authHandle == TPM_RH_OWNER)
    {
        // If Owner provided auth then ONWERWRITE must be SET
        if(! nvIndex.publicArea.attributes.TPMA_NV_OWNERREAD)
            return TPM_RC_NV_AUTHORIZATION;
    }
    else if(authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided auth then PPWRITE must be SET
        if(!nvIndex.publicArea.attributes.TPMA_NV_PPREAD)
            return TPM_RC_NV_AUTHORIZATION;
    }
    // If neither Owner nor Platform provided auth, make sure that it was
    // provided by this index.
    else if(authHandle != nvHandle)
            return TPM_RC_NV_AUTHORIZATION;

    // If the index has not been written, then the value cannot be read
    // NOTE: This has to come after other access checks to make sure that
    // the proper authorization is given to TPM2_NV_ReadLock()
    if(nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
        return TPM_RC_NV_UNINITIALIZED;

    return TPM_RC_SUCCESS;
}
```

7.5.2.2 NvWriteAccessChecks()

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

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	Authorization fails
TPM_RC_NV_LOCKED	Write locked

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

7.6 Object Command Support (Object_spt.c)

7.6.1 Includes

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

7.6.2 Local Functions

7.6.2.1 EqualCryptSet()

Check if the crypto sets in two public areas are equal

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

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

7.6.2.2 GetIV2BSize()

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

Return Value	Meaning

```
37  static UINT16
38  GetIV2BSize(
39      TPM_HANDLE       protectorHandle    // IN: the protector handle
40      )
41  {
42      OBJECT              *protector = NULL; // Pointer to the protector object
43      TPM_ALG_ID          symAlg;
```

```
44      UINT16              keyBits;
45
46      // Determine the symmetric algorithm and size of key
47      if(protectorHandle == TPM_RH_NULL)
48      {
49          // Use the context encryption algorithm and key size
50          symAlg = CONTEXT_ENCRYPT_ALG;
51          keyBits = CONTEXT_ENCRYPT_KEY_BITS;
52      }
53      else
54      {
55          protector = ObjectGet(protectorHandle);
56          symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
57          keyBits= protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
58      }
59
60      // The IV size is a UINT16 size field plus the block size of the symmetric
61      // algorithm
62      return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
63  }
```

7.6.2.3 ComputeProtectionKeyParms()

This function retrieves the symmetric protection key parameters for the sensitive data The parameters retrieved from this function include encryption algorithm, key size in bit, and a TPM2B_SYM_KEY containing the key material as well as the key size in bytes This function is used for any action that requires encrypting or decrypting of the sensitive area of an object or a credential blob

```
64  static void
65  ComputeProtectionKeyParms(
66      TPM_HANDLE       protectorHandle,   // IN: the protector handle
67      TPM_ALG_ID       hashAlg,           // IN: hash algorithm for KDFa
68      TPM2B_NAME      *name,              // IN: name of the object
69      TPM2B_SEED      *seedIn,            // IN: optional seed for duplication blob.
70                                          //     For non duplication blob, this
71                                          //     parameter should be NULL
72      TPM_ALG_ID      *symAlg,            // OUT: the symmetric algorithm
73      UINT16          *keyBits,           // OUT: the symmetric key size in bits
74      TPM2B_SYM_KEY   *symKey             // OUT: the symmetric key
75      )
76  {
77      TPM2B_SEED          *seed = NULL;
78      OBJECT              *protector = NULL; // Pointer to the protector
79
80      // Determine the algorithms for the KDF and the encryption/decryption
81      // For TPM_RH_NULL, using context settings
82      if(protectorHandle == TPM_RH_NULL)
83      {
84          // Use the context encryption algorithm and key size
85          *symAlg = CONTEXT_ENCRYPT_ALG;
86          symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
87          *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
88      }
89      else
90      {
91          TPMT_SYM_DEF_OBJECT *symDef;
92          protector = ObjectGet(protectorHandle);
93          symDef = &protector->publicArea.parameters.asymDetail.symmetric;
94          *symAlg = symDef->algorithm;
95          *keyBits= symDef->keyBits.sym;
96          symKey->t.size = (*keyBits + 7) / 8;
97      }
98
99      // Get seed for KDF
```

```
100         seed = GetSeedForKDF(protectorHandle, seedIn);
101
102         // KDFa to generate symmetric key and IV value
103         KDFa(hashAlg, (TPM2B *)seed, "STORAGE", (TPM2B *)name, NULL,
104              symKey->t.size * 8, symKey->t.buffer, NULL);
105
106         return;
107     }
```

7.6.2.4 ComputeOuterIntegrity()

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

```
108     static void
109     ComputeOuterIntegrity(
110         TPM2B_NAME      *name,              // IN: the name of the object
111         TPM_HANDLE       protectorHandle,   // IN: The handle of the object that
112                                             //     provides protection.  For object, it
113                                             //     is parent handle. For credential, it
114                                             //     is the handle of encrypt object.  For
115                                             //     a Temporary Object, it is TPM_RH_NULL
116         TPMI_ALG_HASH    hashAlg,           // IN: algorithm to use for integrity
117         TPM2B_SEED      *seedIn,            // IN: an external seed may be provided for
118                                             //     duplication blob. For non duplication
119                                             //     blob, this parameter should be NULL
120         UINT32           sensitiveSize,     // IN: size of the marshaled sensitive data
121         BYTE            *sensitiveData,     // IN: sensitive area
122         TPM2B_DIGEST    *integrity          // OUT: integrity
123         )
124     {
125         HMAC_STATE          hmacState;
126
127         TPM2B_DIGEST        hmacKey;
128         TPM2B_SEED          *seed = NULL;
129
130         // Get seed for KDF
131         seed = GetSeedForKDF(protectorHandle, seedIn);
132
133         // Determine the HMAC key bits
134         hmacKey.t.size = CryptGetHashDigestSize(hashAlg);
135
136         // KDFa to generate HMAC key
137         KDFa(hashAlg, (TPM2B *)seed, "INTEGRITY", NULL, NULL,
138              hmacKey.t.size * 8, hmacKey.t.buffer, NULL);
139
140         // Start HMAC and get the size of the digest which will become the integrity
141         integrity->t.size = CryptStartHMAC2B(hashAlg, &hmacKey.b, &hmacState);
142
143         // Adding the marshaled sensitive area to the integrity value
144         CryptUpdateDigest(&hmacState, sensitiveSize, sensitiveData);
145
146         // Adding name
147         CryptUpdateDigest2B(&hmacState, (TPM2B *)name);
148
149         // Compute HMAC
150         CryptCompleteHMAC2B(&hmacState, &integrity->b);
151
152         return;
153     }
```

7.6.2.5 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap

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

7.6.2.6 ProduceInnerIntegrity()

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

```
180  static UINT16
181  ProduceInnerIntegrity(
182      TPM2B_NAME      *name,          // IN: the name of the object
183      TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
184      UINT16           dataSize,      // IN: the size of sensitive data, excluding the
185                                      //     leading integrity buffer size
186      BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
187                                      //     it.  At input, the leading bytes of this
188                                      //     buffer is reserved for integrity
189      )
190  {
191      BYTE                *sensitiveData; // pointer to the sensitive data
192
193      TPM2B_DIGEST        integrity;
194      UINT16              integritySize;
195      BYTE                *buffer;        // Auxiliary buffer pointer
196
197      // sensitiveData points to the beginning of sensitive data in innerBuffer
198      integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
199      sensitiveData = innerBuffer + integritySize;
200
201      ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);
202
203      // Add integrity at the beginning of inner buffer
204      buffer = innerBuffer;
```

```
205         TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
206
207         return dataSize + integritySize;
208     }
```

7.6.2.7 CheckInnerIntegrity()

This function check integrity of inner blob

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

```
209     static TPM_RC
210     CheckInnerIntegrity(
211         TPM2B_NAME      *name,          // IN: the name of the object
212         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
213         UINT16           dataSize,      // IN: the size of sensitive data, including the
214                                         //     leading integrity buffer size
215         BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
216                                         //     it
217         )
218     {
219         TPM_RC          result;
220
221         TPM2B_DIGEST    integrity;
222         TPM2B_DIGEST    integrityToCompare;
223         BYTE            *buffer;                // Auxiliary buffer pointer
224         INT32           size;
225
226         // Unmarshal integrity
227         buffer = innerBuffer;
228         size = (INT32) dataSize;
229         result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
230         if(result == TPM_RC_SUCCESS)
231         {
232             // Compute integrity to compare
233             ComputeInnerIntegrity(hashAlg, name, (UINT16) size, buffer,
234                                   &integrityToCompare);
235
236             // Compare outer blob integrity
237             if(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
238                 result = TPM_RC_INTEGRITY;
239         }
240         return result;
241     }
```

7.6.3 Public Functions

7.6.3.1 AreAttributesForParent()

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

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

```
242     BOOL
```

```
243 AreAttributesForParent(
244     OBJECT          *parentObject   // IN: parent handle
245     )
246 {
247     // This function is only called when a parent is needed. Any
248     // time a "parent" is used, it must be authorized. When
249     // the authorization is checked, both the public and sensitive
250     // areas must be loaded. Just make sure...
251     pAssert(parentObject->attributes.publicOnly == CLEAR);
252 
253     if(ObjectDataIsStorage(&parentObject->publicArea))
254         return TRUE;
255     else
256         return FALSE;
257 }
```

7.6.3.2 SchemeChecks()

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

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

```
258 TPM_RC
259 SchemeChecks(
260     BOOL             load,          // IN: TRUE if load checks, FALSE if
261                                     //     TPM2_Create()
262     TPMI_DH_OBJECT   parentHandle,  // IN: input parent handle
263     TPMT_PUBLIC     *publicArea     // IN: public area of the object
264     )
265 {
266 
267     // Checks for an asymmetric key
268     if(CryptIsAsymAlgorithm(publicArea->type))
269     {
270         TPMT_ASYM_SCHEME        *keyScheme;
271         keyScheme = &publicArea->parameters.asymDetail.scheme;
272 
273         // An asymmetric key can't be injected
274         // This is only checked when creating an object
275         if(!load && (publicArea->objectAttributes.sensitiveDataOrigin == CLEAR))
276             return TPM_RC_ATTRIBUTES;
277 
```

```
278         if(load && !CryptAreKeySizesConsistent(publicArea))
279             return TPM_RC_KEY;
280 
281         // Keys that are both signing and decrypting must have TPM_ALG_NULL
282         // for scheme
283         if(    publicArea->objectAttributes.sign == SET
284            && publicArea->objectAttributes.decrypt == SET
285            && keyScheme->scheme != TPM_ALG_NULL)
286             return TPM_RC_SCHEME;
287 
288         // A restrict sign key must have a non-NULL scheme
289         if(     publicArea->objectAttributes.restricted == SET
290             &&  publicArea->objectAttributes.sign == SET
291             &&  keyScheme->scheme == TPM_ALG_NULL)
292             return TPM_RC_SCHEME;
293 
294         // Keys must have a valid sign or decrypt scheme, or a TPM_ALG_NULL
295         // scheme
296         // NOTE: The unmarshaling for a public area will unmarshal based on the
297         // object type. If the type is an RSA key, then only RSA schemes will be
298         // allowed because a TPMI_ALG_RSA_SCHEME will be unmarshaled and it
299         // consists only of those algorithms that are allowed with an RSA key.
300         // This means that there is no need to again make sure that the algorithm
301         // is compatible with the object type.
302         if(   keyScheme->scheme != TPM_ALG_NULL
303            && (   (   publicArea->objectAttributes.sign == SET
304                    && !CryptIsSignScheme(keyScheme->scheme)
305                   )
306                || (   publicArea->objectAttributes.decrypt == SET
307                    && !CryptIsDecryptScheme(keyScheme->scheme)
308                   )
309               )
310           )
311             return TPM_RC_SCHEME;
312 
313         // Special checks for an ECC key
314 #ifdef TPM_ALG_ECC
315         if(publicArea->type == TPM_ALG_ECC)
316         {
317             TPM_ECC_CURVE        curveID = publicArea->parameters.eccDetail.curveID;
318             const TPMT_ECC_SCHEME  *curveScheme = CryptGetCurveSignScheme(curveID);
319             // The curveId must be valid or the unmarshaling is busted.
320             pAssert(curveScheme != NULL);
321 
322             // If the curveID requires a specific scheme, then the key must select
323             // the same scheme
324             if(curveScheme->scheme != TPM_ALG_NULL)
325             {
326                 if(keyScheme->scheme != curveScheme->scheme)
327                     return TPM_RC_SCHEME;
328                 // The scheme can allow any hash, or not...
329                 if(   curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
330                    && (   keyScheme->details.anySig.hashAlg
331                        != curveScheme->details.anySig.hashAlg
332                       )
333                   )
334                     return TPM_RC_SCHEME;
335             }
336             // For now, the KDF must be TPM_ALG_NULL
337             if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
338                 return TPM_RC_KDF;
339         }
340 #endif
341 
342         // Checks for a storage key (restricted + decryption)
343         if(   publicArea->objectAttributes.restricted == SET
```

```
344             && publicArea->objectAttributes.decrypt == SET)
345         {
346             // A storage key must have a valid protection key
347             if(   publicArea->parameters.asymDetail.symmetric.algorithm
348                == TPM_ALG_NULL)
349                 return TPM_RC_SYMMETRIC;
350
351             // A storage key must have a null scheme
352             if(publicArea->parameters.asymDetail.scheme.scheme != TPM_ALG_NULL)
353                 return TPM_RC_SCHEME;
354
355             // A storage key must match its parent algorithms unless
356             // it is duplicable or a primary (including Temporary Primary Objects)
357             if(   HandleGetType(parentHandle) != TPM_HT_PERMANENT
358                && publicArea->objectAttributes.fixedParent == SET
359               )
360             {
361                 // If the object to be created is a storage key, and is fixedParent,
362                 // its crypto set has to match its parent's crypto set. TPM_RC_TYPE,
363                 // TPM_RC_HASH or TPM_RC_ASYMMETRIC may be returned at this point
364                 return EqualCryptSet(publicArea,
365                                      &(ObjectGet(parentHandle)->publicArea));
366             }
367         }
368         else
369         {
370             // Non-storage keys must have TPM_ALG_NULL for the symmetric algorithm
371             if(   publicArea->parameters.asymDetail.symmetric.algorithm
372                != TPM_ALG_NULL)
373                 return TPM_RC_SYMMETRIC;
374
375         }// End of asymmetric decryption key checks
376     } // End of asymmetric checks
377
378     // Check for bit attributes
379     else if(publicArea->type == TPM_ALG_KEYEDHASH)
380     {
381         TPMT_KEYEDHASH_SCHEME   *scheme
382             = &publicArea->parameters.keyedHashDetail.scheme;
383         // If both sign and decrypt are set the scheme must be TPM_ALG_NULL
384         // and the scheme selected when the key is used.
385         // If neither sign nor decrypt is set, the scheme must be TPM_ALG_NULL
386         // because this is a data object.
387         if(     publicArea->objectAttributes.sign
388             ==  publicArea->objectAttributes.decrypt)
389         {
390             if(scheme->scheme != TPM_ALG_NULL)
391                 return TPM_RC_SCHEME;
392             return TPM_RC_SUCCESS;
393         }
394         // If this is a decryption key, make sure that is is XOR and that there
395         // is a KDF
396         else if(publicArea->objectAttributes.decrypt)
397         {
398             if(   scheme->scheme != TPM_ALG_XOR
399                || scheme->details.xor.hashAlg == TPM_ALG_NULL)
400                 return TPM_RC_SCHEME;
401             if(scheme->details.xor.kdf == TPM_ALG_NULL)
402                 return TPM_RC_KDF;
403             return TPM_RC_SUCCESS;
404
405         }
406         // only supported signing scheme for keyedHash object is HMAC
407         if(   scheme->scheme != TPM_ALG_HMAC
408            || scheme->details.hmac.hashAlg == TPM_ALG_NULL)
409             return TPM_RC_SCHEME;
```

```
410
411         // end of the checks for keyedHash
412         return TPM_RC_SUCCESS;
413     }
414     else if (publicArea->type == TPM_ALG_SYMCIPHER)
415     {
416         // Must be a decrypting key and may not be a signing key
417         if(   publicArea->objectAttributes.decrypt == CLEAR
418            || publicArea->objectAttributes.sign == SET
419           )
420             return TPM_RC_ATTRIBUTES;
421     }
422     else
423         return TPM_RC_TYPE;
424
425     return TPM_RC_SUCCESS;
426 }
```

7.6.3.3 PublicAttributesValidation()

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

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

```
427 TPM_RC
428 PublicAttributesValidation(
429     BOOL              load,          // IN: TRUE if load checks, FALSE if
430                                      //     TPM2_Create()
431     TPMI_DH_OBJECT    parentHandle,  // IN: input parent handle
432     TPMT_PUBLIC      *publicArea     // IN: public area of the object
433     )
434 {
435     OBJECT               *parentObject = NULL;
436
437     if(HandleGetType(parentHandle) != TPM_HT_PERMANENT)
438         parentObject = ObjectGet(parentHandle);
```

```
439
440         // Check authPolicy digest consistency
441         if(   publicArea->authPolicy.t.size != 0
442            && (   publicArea->authPolicy.t.size
443                != CryptGetHashDigestSize(publicArea->nameAlg)
444               )
445           )
446             return TPM_RC_SIZE;
447
448         // If the parent is fixedTPM (including a Primary Object) the object must have
449         // the same value for fixedTPM and fixedParent
450         if(    parentObject == NULL
451             || parentObject->publicArea.objectAttributes.fixedTPM == SET)
452         {
453             if(   publicArea->objectAttributes.fixedParent
454                != publicArea->objectAttributes.fixedTPM
455               )
456                 return TPM_RC_ATTRIBUTES;
457         }
458         else
459             // The parent is not fixedTPM so the object can't be fixedTPM
460             if(publicArea->objectAttributes.fixedTPM == SET)
461                 return  TPM_RC_ATTRIBUTES;
462
463         // A restricted object cannot be both sign and decrypt and it can't be neither
464         // sign nor decrypt
465         if (   publicArea->objectAttributes.restricted == SET
466             && (   publicArea->objectAttributes.decrypt
467                 == publicArea->objectAttributes.sign)
468            )
469             return TPM_RC_ATTRIBUTES;
470
471         // A fixedTPM object can not have encryptedDuplication bit SET
472         if(    publicArea->objectAttributes.fixedTPM == SET
473             && publicArea->objectAttributes.encryptedDuplication == SET)
474             return TPM_RC_ATTRIBUTES;
475
476         // If a parent object has fixedTPM CLEAR, the child must have the
477         // same encryptedDuplication value as its parent.
478         // Primary objects are considered to have a fixedTPM parent (the seeds).
479        if(      (   parentObject != NULL
480                  && parentObject->publicArea.objectAttributes.fixedTPM == CLEAR)
481             // Get here if parent is not fixed TPM
482             &&  (   publicArea->objectAttributes.encryptedDuplication
483                  != parentObject->publicArea.objectAttributes.encryptedDuplication
484                 )
485           )
486             return TPM_RC_ATTRIBUTES;
487
488        return SchemeChecks(load, parentHandle, publicArea);
489    }
```

7.6.3.4 FillInCreationData()

Fill in creation data for an object.

```
490    void
491    FillInCreationData(
492        TPMI_DH_OBJECT           parentHandle,  // IN: handle of parent
493        TPMI_ALG_HASH            nameHashAlg,   // IN: name hash algorithm
494        TPML_PCR_SELECTION      *creationPCR,   // IN: PCR selection
495        TPM2B_DATA              *outsideData,   // IN: outside data
496        TPM2B_CREATION_DATA     *outCreation,   // OUT: creation data for output
497        TPM2B_DIGEST            *creationDigest // OUT: creation digest
```

```
498     )
499 {
500     BYTE                 creationBuffer[sizeof(TPMS_CREATION_DATA)];
501     BYTE                *buffer;
502     HASH_STATE           hashState;
503 
504     // Fill in TPMS_CREATION_DATA in outCreation
505 
506     // Compute PCR digest
507     PCRComputeCurrentDigest(nameHashAlg, creationPCR,
508                             &outCreation->t.creationData.pcrDigest);
509 
510     // Put back PCR selection list
511     outCreation->t.creationData.pcrSelect = *creationPCR;
512 
513     // Get locality
514     outCreation->t.creationData.locality
515         = LocalityGetAttributes(_plat__LocalityGet());
516 
517     outCreation->t.creationData.parentNameAlg = TPM_ALG_NULL;
518 
519     // If the parent is is either a primary seed or TPM_ALG_NULL, then  the Name
520     // and QN of the parent are the parent's handle.
521     if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
522     {
523         BYTE        *buffer = &outCreation->t.creationData.parentName.t.name[0];
524         outCreation->t.creationData.parentName.t.size =
525             TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
526 
527         // Parent qualified name of a Temporary Object is the same as parent's
528         // name
529         MemoryCopy2B(&outCreation->t.creationData.parentQualifiedName.b,
530                      &outCreation->t.creationData.parentName.b,
531                     sizeof(outCreation->t.creationData.parentQualifiedName.t.name));
532 
533     }
534     else         // Regular object
535     {
536         OBJECT          *parentObject = ObjectGet(parentHandle);
537 
538         // Set name algorithm
539         outCreation->t.creationData.parentNameAlg =
540             parentObject->publicArea.nameAlg;
541         // Copy parent name
542         outCreation->t.creationData.parentName = parentObject->name;
543 
544         // Copy parent qualified name
545         outCreation->t.creationData.parentQualifiedName =
546             parentObject->qualifiedName;
547     }
548 
549     // Copy outside information
550     outCreation->t.creationData.outsideInfo = *outsideData;
551 
552     // Marshal creation data to canonical form
553     buffer = creationBuffer;
554     outCreation->t.size = TPMS_CREATION_DATA_Marshal(&outCreation->t.creationData,
555                           &buffer, NULL);
556 
557     // Compute hash for creation field in public template
558     creationDigest->t.size = CryptStartHash(nameHashAlg, &hashState);
559     CryptUpdateDigest(&hashState, outCreation->t.size, creationBuffer);
560     CryptCompleteHash2B(&hashState, &creationDigest->b);
561 
562     return;
563 }
```

7.6.3.5 GetSeedForKDF()

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

```
564  TPM2B_SEED*
565  GetSeedForKDF(
566      TPM_HANDLE       protectorHandle,   // IN: the protector handle
567      TPM2B_SEED      *seedIn             // IN: the optional input seed
568      )
569  {
570      OBJECT              *protector = NULL; // Pointer to the protector
571
572      // Get seed for encryption key.  Use input seed if provided.
573      // Otherwise, using protector object's seedValue.  TPM_RH_NULL is the only
574      // exception that we may not have a loaded object as protector.  In such a
575      // case, use nullProof as seed.
576      if(seedIn != NULL)
577      {
578          return seedIn;
579      }
580      else
581      {
582          if(protectorHandle == TPM_RH_NULL)
583          {
584              return (TPM2B_SEED *) &gr.nullProof;
585          }
586          else
587          {
588              protector = ObjectGet(protectorHandle);
589              return (TPM2B_SEED *) &protector->sensitive.seedValue;
590          }
591      }
592  }
```

7.6.3.6 ProduceOuterWrap()

This function produce outer wrap for a buffer containing the sensitive data. It requires the sensitive data being marshaled to the *outerBuffer*, with the leading bytes reserved for integrity hash. If iv is used, iv space should be reserved at the beginning of the buffer. It assumes the sensitive data starts at address *(outerBuffer* + integrity size {+ iv size}). This function performs:

a) Add IV before sensitive area if required

b) encrypt sensitive data, if iv is required, encrypt by iv. otherwise, encrypted by a NULL iv

c) add HMAC integrity at the beginning of the buffer It returns the total size of blob with outer wrap

```
593  UINT16
594  ProduceOuterWrap(
595      TPM_HANDLE       protector,     // IN: The handle of the object that provides
596                                      //     protection.  For object, it is parent
597                                      //     handle. For credential, it is the handle
598                                      //     of encrypt object.
599      TPM2B_NAME      *name,          // IN: the name of the object
600      TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
601      TPM2B_SEED      *seed,          // IN: an external seed may be provided for
602                                      //     duplication blob. For non duplication
603                                      //     blob, this parameter should be NULL
604      BOOL             useIV,         // IN: indicate if an IV is used
605      UINT16           dataSize,      // IN: the size of sensitive data, excluding the
606                                      //     leading integrity buffer size or the
607                                      //     optional iv size
608      BYTE            *outerBuffer    // IN/OUT: outer buffer with sensitive data in
```

```
609                                     //     it
610     )
611 {
612     TPM_ALG_ID      symAlg;
613     UINT16          keyBits;
614     TPM2B_SYM_KEY   symKey;
615     TPM2B_IV        ivRNG;          // IV from RNG
616     TPM2B_IV        *iv = NULL;
617     UINT16          ivSize = 0;     // size of iv area, including the size field
618
619     BYTE            *sensitiveData; // pointer to the sensitive data
620
621     TPM2B_DIGEST    integrity;
622     UINT16          integritySize;
623     BYTE            *buffer;        // Auxiliary buffer pointer
624
625     // Compute the beginning of sensitive data.  The outer integrity should
626     // always exist if this function function is called to make an outer wrap
627     integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
628     sensitiveData = outerBuffer + integritySize;
629
630     // If iv is used, adjust the pointer of sensitive data and add iv before it
631     if(useIV)
632     {
633         ivSize = GetIV2BSize(protector);
634
635         // Generate IV from RNG.  The iv data size should be the total IV area
636         // size minus the size of size field
637         ivRNG.t.size = ivSize - sizeof(UINT16);
638         CryptGenerateRandom(ivRNG.t.size, ivRNG.t.buffer);
639
640         // Marshal IV to buffer
641         buffer = sensitiveData;
642         TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
643
644         // adjust sensitive data starting after IV area
645         sensitiveData += ivSize;
646
647         // Use iv for encryption
648         iv = &ivRNG;
649     }
650
651     // Compute symmetric key parameters for outer buffer encryption
652     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
653                               &symAlg, &keyBits, &symKey);
654     // Encrypt inner buffer in place
655     CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
656                           TPM_ALG_CFB, symKey.t.buffer, iv, dataSize,
657                           sensitiveData);
658
659     // Compute outer integrity.  Integrity computation includes the optional IV
660     // area
661     ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
662                           outerBuffer + integritySize, &integrity);
663
664     // Add integrity at the beginning of outer buffer
665     buffer = outerBuffer;
666     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
667
668     // return the total size in outer wrap
669     return dataSize + integritySize + ivSize;
670
671 }
```


7.6.3.7 UnwrapOuter()

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

a) check integrity of outer blob

b) decrypt outer blob

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

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

```
720                                     &symAlg, &keyBits, &symKey);
721
722         // Retrieve IV if it is used
723         if(useIV)
724         {
725             result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
726             if(result == TPM_RC_SUCCESS)
727             {
728                 // The input iv size for CFB must match the encryption algorithm
729                 // block size
730                 if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
731                     result = TPM_RC_VALUE;
732                 else
733                     iv = &ivIn;
734             }
735         }
736     }
737     // If no errors, decrypt private in place
738     if(result == TPM_RC_SUCCESS)
739         CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
740                               TPM_ALG_CFB, symKey.t.buffer, iv,
741                               (UINT16) size, sensitiveData);
742
743     return result;
744
745 }
```

7.6.3.8 SensitiveToPrivate()

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

a) marshal TPM2B_SENSITIVE structure into the buffer of TPM2B_PRIVATE

b) apply encryption to the sensitive area.

c) apply outer integrity computation.

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

```
774         {
775             // Otherwise, using parent's name algorithm
776             hashAlg = ObjectGetNameAlg(parentHandle);
777         }
778 
779         // Starting of sensitive data without wrappers
780         sensitiveData = outPrivate->t.buffer;
781 
782         // Compute the integrity size
783         integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
784 
785         // Reserve space for integrity
786         sensitiveData += integritySize;
787 
788         // Get iv size
789         ivSize = GetIV2BSize(parentHandle);
790 
791         // Reserve space for iv
792         sensitiveData += ivSize;
793 
794         // Marshal sensitive area, leaving the leading 2 bytes for size
795         buffer = sensitiveData + sizeof(UINT16);
796         dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
797 
798         // Adding size before the data area
799         buffer = sensitiveData;
800         UINT16_Marshal(&dataSize, &buffer, NULL);
801 
802         // Adjust the dataSize to include the size field
803         dataSize += sizeof(UINT16);
804 
805         // Adjust the pointer to inner buffer including the iv
806         sensitiveData = outPrivate->t.buffer + ivSize;
807 
808         //Produce outer wrap, including encryption and HMAC
809         outPrivate->t.size = ProduceOuterWrap(parentHandle, name, hashAlg, NULL,
810                                               TRUE, dataSize, outPrivate->t.buffer);
811 
812         return;
813     }
```

7.6.3.9 PrivateToSensitive()

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

a) check the integrity HMAC of the input private area

b) decrypt the private buffer

c) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

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

```
814     TPM_RC
815     PrivateToSensitive(
816         TPM2B_PRIVATE   *inPrivate,     // IN: input private structure
817         TPM2B_NAME      *name,          // IN: the name of the object
```

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

```
884             // Unmarshal sensitive buffer to sensitive structure
885             result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
886             if(result != TPM_RC_SUCCESS || size != 0)
887             {
888                 pAssert(    (parent == NULL)
889                         || parent->publicArea.objectAttributes.fixedTPM == CLEAR);
890                 result = TPM_RC_SENSITIVE;
891             }
892             else
893             {
894                 // Always remove trailing zeros at load so that it is not necessary
895                 // to check
896                 // each time auth is checked.
897                 MemoryRemoveTrailingZeros(&(sensitive->authValue));
898             }
899         }
900     }
901     return result;
902 }
```

7.6.3.10 SensitiveToDuplicate()

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

a) marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE

b) apply inner wrap to the sensitive area if required

c) apply outer wrap if required

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

```
938         pAssert(symDef != NULL && innerSymKey != NULL);
939
940         // Starting of sensitive data without wrappers
941         sensitiveData = outPrivate->t.buffer;
942
943         // Find out if inner wrap is required
944         if(symDef->algorithm != TPM_ALG_NULL)
945         {
946             doInnerWrap = TRUE;
947             // Use self nameAlg as inner hash algorithm
948             innerHash = nameAlg;
949             // Adjust sensitive data pointer
950             sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
951         }
952
953         // Find out if outer wrap is required
954         if(seed->t.size != 0)
955         {
956             doOuterWrap = TRUE;
957             // Use parent nameAlg as outer hash algorithm
958             outerHash = ObjectGetNameAlg(parentHandle);
959             // Adjust sensitive data pointer
960             sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
961         }
962
963         // Marshal sensitive area, leaving the leading 2 bytes for size
964         buffer = sensitiveData + sizeof(UINT16);
965         dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
966
967         // Adding size before the data area
968         buffer = sensitiveData;
969         UINT16_Marshal(&dataSize, &buffer, NULL);
970
971         // Adjust the dataSize to include the size field
972         dataSize += sizeof(UINT16);
973
974         // Apply inner wrap for duplication blob.  It includes both integrity and
975         // encryption
976         if(doInnerWrap)
977         {
978             BYTE            *innerBuffer = NULL;
979             BOOL            symKeyInput = TRUE;
980             innerBuffer = outPrivate->t.buffer;
981             // Skip outer integrity space
982             if(doOuterWrap)
983                 innerBuffer += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
984             dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
985                                              innerBuffer);
986
987             // Generate inner encryption key if needed
988             if(innerSymKey->t.size == 0)
989             {
990                 innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
991                 CryptGenerateRandom(innerSymKey->t.size, innerSymKey->t.buffer);
992
993                 // TPM generates symmetric encryption.  Set the flag to FALSE
994                 symKeyInput = FALSE;
995             }
996             else
997             {
998                 // assume the input key size should matches the symmetric definition
999                 pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1000
1001            }
1002
1003            // Encrypt inner buffer in place
```

```
1004            CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
1005                                  symDef->keyBits.sym, TPM_ALG_CFB,
1006                                  innerSymKey->t.buffer, NULL, dataSize,
1007                                  innerBuffer);
1008
1009            // If the symmetric encryption key is imported, clear the buffer for
1010            // output
1011            if(symKeyInput)
1012                innerSymKey->t.size = 0;
1013        }
1014
1015        // Apply outer wrap for duplication blob.  It includes both integrity and
1016        // encryption
1017        if(doOuterWrap)
1018        {
1019            dataSize = ProduceOuterWrap(parentHandle, name, outerHash, seed, FALSE,
1020                                        dataSize, outPrivate->t.buffer);
1021        }
1022
1023        // Data size for output
1024        outPrivate->t.size = dataSize;
1025
1026        return;
1027    }
```

7.6.3.11 DuplicateToSensitive()

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

a) check the integrity HMAC of the input private area

b) decrypt the private buffer

c) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

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

```
1028    TPM_RC
1029    DuplicateToSensitive(
1030        TPM2B_PRIVATE           *inPrivate,     // IN: input private structure
1031        TPM2B_NAME              *name,          // IN: the name of the object
1032        TPM_HANDLE               parentHandle,  // IN: The parent's handle
1033        TPM_ALG_ID               nameAlg,       // IN: hash algorithm in public area.
1034        TPM2B_SEED              *seed,          // IN: an external seed may be provided.
1035                                                //     If external seed is provided with
1036                                                //     size of 0, no outer wrap is
1037                                                //     applied
1038        TPMT_SYM_DEF_OBJECT     *symDef,        // IN: Symmetric key definition. If the
1039                                                //     symmetric key algorithm is NULL,
1040                                                //     no inner wrap is applied
1041        TPM2B_DATA              *innerSymKey,   // IN: a symmetric key may be provided
1042                                                //     to decrypt the inner wrap of a
1043                                                //     duplication blob.
1044        TPMT_SENSITIVE          *sensitive      // OUT: sensitive structure
1045        )
1046    {
1047        TPM_RC          result;
1048
```

```
1049        BYTE            *buffer;
1050        INT32           size;
1051        BYTE            *sensitiveData; // pointer to the sensitive data
1052        UINT16          dataSize;
1053        UINT16          dataSizeInput;
1054
1055        // Make sure that name is provided
1056        pAssert(name != NULL && name->t.size != 0);
1057
1058        // Make sure symDef and innerSymKey are not NULL
1059        pAssert(symDef != NULL && innerSymKey != NULL);
1060
1061        // Starting of sensitive data
1062        sensitiveData = inPrivate->t.buffer;
1063        dataSize = inPrivate->t.size;
1064
1065        // Find out if outer wrap is applied
1066        if(seed->t.size != 0)
1067        {
1068            TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;
1069
1070            // Use parent nameAlg as outer hash algorithm
1071            outerHash = ObjectGetNameAlg(parentHandle);
1072            result = UnwrapOuter(parentHandle, name, outerHash, seed, FALSE,
1073                                 dataSize, sensitiveData);
1074            if(result != TPM_RC_SUCCESS)
1075                return result;
1076
1077            // Adjust sensitive data pointer and size
1078            sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1079            dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1080        }
1081        // Find out if inner wrap is applied
1082        if(symDef->algorithm != TPM_ALG_NULL)
1083        {
1084            TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;
1085
1086            // assume the input key size should matches the symmetric definition
1087            pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1088
1089            // Decrypt inner buffer in place
1090            CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
1091                                  symDef->keyBits.sym, TPM_ALG_CFB,
1092                                  innerSymKey->t.buffer, NULL, dataSize,
1093                                  sensitiveData);
1094
1095            // Use self nameAlg as inner hash algorithm
1096            innerHash = nameAlg;
1097
1098            // Check inner integrity
1099            result = CheckInnerIntegrity(name, innerHash, dataSize, sensitiveData);
1100            if(result != TPM_RC_SUCCESS)
1101                return result;
1102
1103            // Adjust sensitive data pointer and size
1104            sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
1105            dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
1106        }
1107
1108        // Unmarshal input data size
1109        buffer = sensitiveData;
1110        size = (INT32) dataSize;
1111        result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1112        if(result == TPM_RC_SUCCESS)
1113        {
1114            if((dataSizeInput + sizeof(UINT16)) != dataSize)
```

```
1115                result = TPM_RC_SIZE;
1116            else
1117            {
1118                // Unmarshal sensitive buffer to sensitive structure
1119                result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1120                // if the results is OK make sure that all the data was unmarshaled
1121                if(result == TPM_RC_SUCCESS && size != 0)
1122                    result = TPM_RC_SIZE;
1123            }
1124        }
1125        // Always remove trailing zeros at load so that it is not necessary to check
1126        // each time auth is checked.
1127        if(result == TPM_RC_SUCCESS)
1128            MemoryRemoveTrailingZeros(&(sensitive->authValue));
1129        return result;
1130    }
```

7.6.3.12 SecretToCredential()

This function prepare the credential blob from a secret (a TPM2B_DIGEST) The operations in this function:

a) marshal TPM2B_DIGEST structure into the buffer of TPM2B_ID_OBJECT

b) encrypt the private buffer, excluding the leading integrity HMAC area

c) compute integrity HMAC and append to the beginning of the buffer.

d) Set the total size of TPM2B_ID_OBJECT buffer

```
1131    void
1132    SecretToCredential(
1133        TPM2B_DIGEST        *secret,        // IN: secret information
1134        TPM2B_NAME          *name,          // IN: the name of the object
1135        TPM2B_SEED          *seed,          // IN: an external seed.
1136        TPM_HANDLE           protector,     // IN: The protector's handle
1137        TPM2B_ID_OBJECT     *outIDObject    // OUT: output credential
1138        )
1139    {
1140        BYTE                *buffer;        // Auxiliary buffer pointer
1141        BYTE                *sensitiveData; // pointer to the sensitive data
1142        TPMI_ALG_HASH        outerHash;     // The hash algorithm for outer wrap
1143        UINT16               dataSize;      // data blob size
1144
1145        pAssert(secret != NULL && outIDObject != NULL);
1146
1147        // use protector's name algorithm as outer hash
1148        outerHash = ObjectGetNameAlg(protector);
1149
1150        // Marshal secret area to credential buffer, leave space for integrity
1151        sensitiveData = outIDObject->t.credential
1152                        + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1153
1154        // Marshal secret area
1155        buffer = sensitiveData;
1156        dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1157
1158        // Apply outer wrap
1159        outIDObject->t.size = ProduceOuterWrap(protector,
1160                                               name,
1161                                               outerHash,
1162                                               seed,
1163                                               FALSE,
1164                                               dataSize,
1165                                               outIDObject->t.credential);
```

```
1166        return;
1167    }
```

7.6.3.13 CredentialToSecret()

Unwrap a credential. Check the integrity, decrypt and retrieve data to a TPM2B_DIGEST structure. The operations in this function:

a) check the integrity HMAC of the input credential area

b) decrypt the credential buffer

c) unmarshal TPM2B_DIGEST structure into the buffer of TPM2B_DIGEST

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

```
1168    TPM_RC
1169    CredentialToSecret(
1170        TPM2B_ID_OBJECT     *inIDObject,    // IN: input credential blob
1171        TPM2B_NAME          *name,          // IN: the name of the object
1172        TPM2B_SEED          *seed,          // IN: an external seed.
1173        TPM_HANDLE           protector,     // IN: The protector's handle
1174        TPM2B_DIGEST        *secret         // OUT: secret information
1175        )
1176    {
1177        TPM_RC                   result;
1178        BYTE                    *buffer;
1179        INT32                    size;
1180        TPMI_ALG_HASH            outerHash;     // The hash algorithm for outer wrap
1181        BYTE                    *sensitiveData; // pointer to the sensitive data
1182        UINT16                   dataSize;
1183
1184        // use protector's name algorithm as outer hash
1185        outerHash = ObjectGetNameAlg(protector);
1186
1187        // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1188        result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
1189                             inIDObject->t.size, inIDObject->t.credential);
1190        if(result == TPM_RC_SUCCESS)
1191        {
1192            // Compute the beginning of sensitive data
1193            sensitiveData = inIDObject->t.credential
1194                            + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1195            dataSize = inIDObject->t.size
1196                       - (sizeof(UINT16) + CryptGetHashDigestSize(outerHash));
1197
1198            // Unmarshal secret buffer to TPM2B_DIGEST structure
1199            buffer = sensitiveData;
1200            size = (INT32) dataSize;
1201            result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1202            // If there were no other unmarshaling errors, make sure that the
1203            // expected amount of data was recovered
1204            if(result == TPM_RC_SUCCESS && size != 0)
1205                return TPM_RC_SIZE;
1206        }
1207        return result;
1208    }
```

8 Subsystem

8.1 CommandAudit.c

8.1.1 Introduction

This file contains the functions that support command audit.

8.1.2 Includes

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

8.1.3 Functions

8.1.3.1 CommandAuditPreInstall_Init()

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

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

```
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.

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

```
33  {
34      if(type == SU_RESET)
35      {
36          // Reset the digest size to initialize the digest
37          gr.commandAuditDigest.t.size = 0;
38      }
39
40  }
```

8.1.3.3 CommandAuditSet()

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

This function is only used by TPM2_SetCommandCodeAuditStatus().

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

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

```
41  BOOL
42  CommandAuditSet(
43      TPM_CC            commandCode    // IN: command code
44      )
45  {
46      UINT32      bitPos;
47
48      // Only SET a bit if the corresponding command is implemented
49      if(CommandIsImplemented(commandCode))
50      {
51          // Can't audit shutdown
52          if(commandCode != TPM_CC_Shutdown)
53          {
54              bitPos = commandCode - TPM_CC_FIRST;
55              if(!BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
56              {
57                  // Set bit
58                  BitSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
59                  return TRUE;
60              }
61          }
62      }
63      // No change
64      return FALSE;
65  }
```

8.1.3.4 CommandAuditClear()

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

This function is only used by TPM2_SetCommandCodeAuditStatus().

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

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

```
66  BOOL
67  CommandAuditClear(
68      TPM_CC            commandCode    // IN: command code
69      )
70  {
71      UINT32      bitPos;
72
73      // Do nothing if the command is not implemented
74      if(CommandIsImplemented(commandCode))
75      {
76          // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
77          // cleared
78          if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
79          {
80              bitPos = commandCode - TPM_CC_FIRST;
81              if(BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
82              {
83                  // Clear bit
84                  BitClear(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
85                  return TRUE;
86              }
87          }
88      }
89      // No change
90      return FALSE;
91  }
```

8.1.3.5 CommandAuditIsRequired()

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

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

```
92  BOOL
93  CommandAuditIsRequired(
94      TPM_CC            commandCode    // IN: command code
95      )
96  {
97      UINT32      bitPos;
98
99      bitPos = commandCode - TPM_CC_FIRST;
100
101     // Check the bit map.  If the bit is SET, command audit is required
102     if((gp.auditComands[bitPos/8] & (1 << (bitPos % 8))) != 0)
103         return TRUE;
104     else
105         return FALSE;
106
107 }
```

8.1.3.6 CommandAuditCapGetCCList()

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

The list starts at the input *commandCode*.

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

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

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.

```
152  void
153  CommandAuditGetDigest(
154      TPM2B_DIGEST    *digest         // OUT: command digest
155      )
156  {
```

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

8.2 DA.c

8.2.1 Introduction

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

8.2.2 Includes and Data Definitions

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

8.2.3 Functions

8.2.3.1 DAPreInstall_Init()

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

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

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

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

```
21      return;
22  }
```

8.2.3.2 DAStartup()

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

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

```
23  void
24  DAStartup(
25      STARTUP_TYPE     type           // IN: startup type
26      )
27  {
28      // For TPM Reset, if lockoutRecovery is 0, enable use of lockoutAuth.
29      if(type == SU_RESET)
30      {
31          if(gp.lockoutRecovery == 0)
32          {
33              gp.lockOutAuthEnabled = TRUE;
34              // Record the changes to NV
35              NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
36          }
37      }
38
39      // If DA has not been disabled and the previous shutdown is not orderly
40      // failedTries is not already at its maximum then increment 'failedTries'
41      if(    gp.recoveryTime != 0
42          && g_prevOrderlyState == SHUTDOWN_NONE
43          && gp.failedTries < gp.maxTries)
44      {
45          gp.failedTries++;
46          // Record the change to NV
47          NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
48      }
49
50      // Reset self healing timers
51      s_selfHealTimer = g_time;
52      s_lockoutTimer = g_time;
53
54      return;
55  }
```

8.2.3.3 DARegisterFailure()

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

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

```
67      return;
68  }
```

8.2.3.4 DASelfHeal()

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

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

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

```
123             {
124                 if(((g_time - s_lockoutTimer)/1000) >= gp.lockoutRecovery)
125                 {
126                     gp.lockOutAuthEnabled = TRUE;
127                     // Record the changes to NV
128                     NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
129                 }
130             }
131         }
132 
133         return;
134     }
```

8.3 Hierarchy.c

8.3.1 Introduction

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

8.3.2 Includes

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

8.3.3 Functions

8.3.3.1 HierarchyPreInstall()

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

```
2   void
3   HierarchyPreInstall_Init(
4       void
5       )
6   {
7       // Allow lockout clear command
8       gp.disableClear = FALSE;
9   
10      // Initialize Primary Seeds
11      gp.EPSeed.t.size = PRIMARY_SEED_SIZE;
12      CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.EPSeed.t.buffer);
13      gp.SPSeed.t.size = PRIMARY_SEED_SIZE;
14      CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.SPSeed.t.buffer);
15      gp.PPSeed.t.size = PRIMARY_SEED_SIZE;
16      CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.PPSeed.t.buffer);
17  
18      // Initialize owner, endorsement and lockout auth
19      gp.ownerAuth.t.size = 0;
20      gp.endorsementAuth.t.size = 0;
21      gp.lockoutAuth.t.size = 0;
22  
23      // Initialize owner, endorsement, and lockout policy
24      gp.ownerAlg = TPM_ALG_NULL;
25      gp.ownerPolicy.t.size = 0;
26      gp.endorsementAlg = TPM_ALG_NULL;
27      gp.endorsementPolicy.t.size = 0;
28      gp.lockoutAlg = TPM_ALG_NULL;
29      gp.lockoutPolicy.t.size = 0;
30  
```



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

8.3.3.2 HierarchyStartup()

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

```
59  void
60  HierarchyStartup(
61      STARTUP_TYPE     type           // IN: start up type
62      )
63  {
64      // phEnable is SET on any startup
65      g_phEnable = TRUE;
66
67      // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
68      // TPM_RESTART
69      if(type != SU_RESUME)
70      {
71          gc.platformAuth.t.size = 0;
72          gc.platformPolicy.t.size = 0;
73
74          // enable the storage and endorsement hierarchies and the platformNV
75          gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
76      }
77
78      // nullProof and nullSeed are updated at every TPM_RESET
79      if(type == SU_RESET)
80      {
81          gr.nullProof.t.size = PROOF_SIZE;
82          CryptGenerateRandom(gr.nullProof.t.size,
83                              gr.nullProof.t.buffer);
84          gr.nullSeed.t.size = PRIMARY_SEED_SIZE;
85          CryptGenerateRandom(PRIMARY_SEED_SIZE, gr.nullSeed.t.buffer);
86      }
87
88      return;
89  }
```

8.3.3.3 HierarchyGetProof()

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

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

8.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```
122 TPM2B_SEED *
123 HierarchyGetPrimarySeed(
124     TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy
125     )
126 {
127     TPM2B_SEED          *seed = NULL;
128     switch(hierarchy)
129     {
130     case TPM_RH_PLATFORM:
131         seed = &gp.PPSeed;
132         break;
133     case TPM_RH_OWNER:
134         seed = &gp.SPSeed;
135         break;
136     case TPM_RH_ENDORSEMENT:
137         seed = &gp.EPSeed;
138         break;
139     case TPM_RH_NULL:
140         return &gr.nullSeed;
141     default:
142         pAssert(FALSE);
143         break;
144     }
```

```
145         return seed;
146     }
```

8.3.3.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE: The TPM_RH_NULL hierarchy is always enabled.

Return Value	Meaning
TRUE	hierarchy is enabled
FALSE	hierarchy is disabled

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

8.4 NV.c

8.4.1 Introduction

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

8.4.2 Includes, Defines and Data Definitions

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

NV Index/evict object iterator value

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

```
6                                       // iterator
```

8.4.3 NV Utility Functions

8.4.3.1 NvCheckState()

Function to check the NV state by accessing the platform-specific function to get the NV state. The result state is registered in *s_NvIsAvailable* that will be reported by NvIsAvailable().

This function is called at the beginning of ExecuteCommand() before any potential call to NvIsAvailable().

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

8.4.3.2 NvIsAvailable()

This function returns the NV availability parameter.

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

```
28  TPM_RC
29  NvIsAvailable(
30      void
31      )
32  {
33      return s_NvStatus;
34  }
```

8.4.3.3 NvCommit

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

```
35  BOOL
36  NvCommit(
37      void
38      )
```

```
39  {
40      BOOL    success = (_plat__NvCommit() == 0);
41      return success;
42  }
```

8.4.3.4 NvReadMaxCount()

This function returns the max NV counter value.

```
43  static UINT64
44  NvReadMaxCount(
45      void
46      )
47  {
48      UINT64      countValue;
49      _plat__NvMemoryRead(s_maxCountAddr, sizeof(UINT64), &countValue);
50      return countValue;
51  }
```

8.4.3.5 NvWriteMaxCount()

This function updates the max counter value to NV memory.

```
52  static void
53  NvWriteMaxCount(
54      UINT64          maxCount
55      )
56  {
57      _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &maxCount);
58      return;
59  }
```

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.

```
60  static UINT32
61  NvNext(
62      NV_ITER         *iter
63      )
64  {
```

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

8.4.4.3 NvGetEnd()

Function to find the end of the NV dynamic data list

```
89  static UINT32
90  NvGetEnd(
91      void
92      )
93  {
94      NV_ITER         iter = NV_ITER_INIT;
95      UINT32          endAddr = s_evictNvStart;
96      UINT32          currentAddr;
97
98      while((currentAddr = NvNext(&iter)) != 0)
99          endAddr = currentAddr;
100
101     if(endAddr != s_evictNvStart)
102     {
103         // Read offset
104         endAddr -= sizeof(UINT32);
105         _plat__NvMemoryRead(endAddr, sizeof(UINT32), &endAddr);
106     }
107
108     return endAddr;
109 }
```

8.4.4.4 NvGetFreeByte

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

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


8.4.4.5 NvGetEvictObjectSize

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

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

8.4.4.6 NvGetCounterSize

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

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

8.4.4.7 NvTestSpace()

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

Return Value	Meaning
TRUE	space available
FALSE	no enough space

```
133  static BOOL
134  NvTestSpace(
135      UINT32            size,          // IN: size of the entity to be added
136      BOOL              isIndex        // IN: TRUE if the entity is an index
137      )
138  {
139      UINT32      remainByte = NvGetFreeByte();
140
141      // For NV Index, need to make sure that we do not allocate and Index if this
142      // would mean that the TPM cannot allocate the minimum number of evict
143      // objects.
144      if(isIndex)
145      {
146          // Get the number of persistent objects allocated
147          UINT32      persistentNum = NvCapGetPersistentNumber();
148
149          // If we have not allocated the requisite number of evict objects, then we
150          // need to reserve space for them.
151          // NOTE: some of this is not written as simply as it might seem because
152          // the values are all unsigned and subtracting needs to be done carefully
153          // so that an underflow doesn't cause problems.
154          if(persistentNum < MIN_EVICT_OBJECTS)
155          {
156              UINT32      needed = (MIN_EVICT_OBJECTS - persistentNum)
157                                   * NvGetEvictObjectSize();
158              if(needed > remainByte)
```

```
159                 remainByte = 0;
160             else
161                 remainByte -= needed;
162         }
163         // if the requisite number of evict objects have been allocated then
164         // no need to reserve additional space
165     }
166     // This checks for the size of the value being added plus the index value.
167     // NOTE: This does not check to see if the end marker can be placed in
168     // memory because the end marker will not be written if it will not fit.
169     return (size + sizeof(UINT32) <= remainByte);
170 }
```

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).

```
171 static void
172 NvAdd(
173     UINT32           totalSize,     // IN: total size needed for this entity For
174                                     //     evict object, totalSize is the same as
175                                     //     bufferSize.  For NV Index, totalSize is
176                                     //     bufferSize plus index data size
177     UINT32           bufferSize,    // IN: size of initial buffer
178     BYTE            *entity         // IN: initial buffer
179     )
180 {
181     UINT32          endAddr;
182     UINT32          nextAddr;
183     UINT32          listEnd = 0;
184
185     // Get the end of data list
186     endAddr = NvGetEnd();
187
188     // Calculate the value of next pointer, which is the size of a pointer +
189     // the entity data size
190     nextAddr = endAddr + sizeof(UINT32) + totalSize;
191
192     // Write next pointer
193     _plat__NvMemoryWrite(endAddr, sizeof(UINT32), &nextAddr);
194
195     // Write entity data
196     _plat__NvMemoryWrite(endAddr + sizeof(UINT32), bufferSize, entity);
197
198     // Write the end of list if it is not going to exceed the NV space
199     if(nextAddr + sizeof(UINT32) <= s_evictNvEnd)
200         _plat__NvMemoryWrite(nextAddr, sizeof(UINT32), &listEnd);
201
202     // Set the flag so that NV changes are committed before the command completes.
203     g_updateNV = TRUE;
204 }
```

8.4.4.9 NvDelete()

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

```
205 static void
206 NvDelete(
207     UINT32           entityAddr     // IN: address of entity to be deleted
208     )
```

```
209 {
210     UINT32          next;
211     UINT32          entrySize;
212     UINT32          entryAddr = entityAddr - sizeof(UINT32);
213     UINT32          listEnd = 0;
214 
215     // Get the offset of the next entry.
216     _plat__NvMemoryRead(entryAddr, sizeof(UINT32), &next);
217 
218     // The size of this entry is the difference between the current entry and the
219     // next entry.
220     entrySize = next - entryAddr;
221 
222     // Move each entry after the current one to fill the freed space.
223     // Stop when we have reached the end of all the indexes. There are two
224     // ways to detect the end of the list. The first is to notice that there
225     // is no room for anything else because we are at the end of NV. The other
226     // indication is that we find an end marker.
227 
228     // The loop condition checks for the end of NV.
229     while(next + sizeof(UINT32) <= s_evictNvEnd)
230     {
231         UINT32      size, oldAddr, newAddr;
232 
233         // Now check for the end marker
234         _plat__NvMemoryRead(next, sizeof(UINT32), &oldAddr);
235         if(oldAddr == 0)
236             break;
237 
238         size = oldAddr - next;
239 
240         // Move entry
241         _plat__NvMemoryMove(next, next - entrySize, size);
242 
243         // Update forward link
244         newAddr = oldAddr - entrySize;
245         _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &newAddr);
246         next = oldAddr;
247     }
248     // Mark the end of list
249     _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &listEnd);
250 
251     // Set the flag so that NV changes are committed before the command completes.
252     g_updateNV = TRUE;
253 }
```

8.4.5 RAM-based NV Index Data Access Functions

8.4.5.1 Introduction

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

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

8.4.5.2 NvTestRAMSpace()

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

Return Value	Meaning
TRUE	space available
FALSE	no enough space

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

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. That is, the index must be known to exist.

```
265 static UINT32
266 NvGetRAMIndexOffset(
267     TPMI_RH_NV_INDEX     handle         // IN: NV handle
268     )
269 {
270     UINT32      currAddr = 0;
271
272     while(currAddr < s_ramIndexSize)
273     {
274         TPMI_RH_NV_INDEX    currHandle;
275         UINT32              currSize;
276         currHandle = * (TPM_HANDLE *) &s_ramIndex[currAddr + sizeof(UINT32)];
277
278         // Found a match
279         if(currHandle == handle)
280
281             // data buffer follows the handle and size field
282             break;
283
284         currSize = * (UINT32 *) &s_ramIndex[currAddr];
285         currAddr += sizeof(UINT32) + currSize;
286     }
287
288     // We assume the index data is existing in RAM space
289     pAssert(currAddr < s_ramIndexSize);
290     return currAddr + sizeof(TPMI_RH_NV_INDEX) + sizeof(UINT32);
291 }
```

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.

```
292 static void
293 NvAddRAM(
294     TPMI_RH_NV_INDEX     handle,        // IN: NV handle
295     UINT32               size           // IN: size of data
296     )
```

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

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

```
313 static void
314 NvDeleteRAM(
315     TPMI_RH_NV_INDEX     handle          // IN: NV handle
316     )
317 {
318     UINT32          nodeOffset;
319     UINT32          nextNode;
320     UINT32          size;
321 
322     nodeOffset = NvGetRAMIndexOffset(handle);
323 
324     // Move the pointer back to get the size field of this node
325     nodeOffset -= sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX);
326 
327     // Get node size
328     size = * (UINT32 *) &s_ramIndex[nodeOffset];
329 
330     // Get the offset of next node
331     nextNode = nodeOffset + sizeof(UINT32) + size;
332 
333     // Move data
334     MemoryMove(s_ramIndex + nodeOffset, s_ramIndex + nextNode,
335                s_ramIndexSize - nextNode, s_ramIndexSize - nextNode);
336 
337     // Update RAM size
338     s_ramIndexSize -= size + sizeof(UINT32);
339 
340     // Update NV version of s_ramIndexSize
341     _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
342 
343     // Write reserved RAM space to NV to reflect the newly delete NV Index
344     _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
345 
346     return;
347 }
```

8.4.6 Utility Functions

8.4.6.1 NvInitStatic()

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

```
348 static void
349 NvInitStatic(
350     void
351     )
352 {
353     UINT16      i;
354     UINT32      reservedAddr;
355 
356     s_reservedSize[NV_DISABLE_CLEAR] = sizeof(gp.disableClear);
357     s_reservedSize[NV_OWNER_ALG] = sizeof(gp.ownerAlg);
358     s_reservedSize[NV_ENDORSEMENT_ALG] = sizeof(gp.endorsementAlg);
359     s_reservedSize[NV_LOCKOUT_ALG] = sizeof(gp.lockoutAlg);
360     s_reservedSize[NV_OWNER_POLICY] = sizeof(gp.ownerPolicy);
361     s_reservedSize[NV_ENDORSEMENT_POLICY] = sizeof(gp.endorsementPolicy);
362     s_reservedSize[NV_LOCKOUT_POLICY] = sizeof(gp.lockoutPolicy);
363     s_reservedSize[NV_OWNER_AUTH] = sizeof(gp.ownerAuth);
364     s_reservedSize[NV_ENDORSEMENT_AUTH] = sizeof(gp.endorsementAuth);
365     s_reservedSize[NV_LOCKOUT_AUTH] = sizeof(gp.lockoutAuth);
366     s_reservedSize[NV_EP_SEED] = sizeof(gp.EPSeed);
367     s_reservedSize[NV_SP_SEED] = sizeof(gp.SPSeed);
368     s_reservedSize[NV_PP_SEED] = sizeof(gp.PPSeed);
369     s_reservedSize[NV_PH_PROOF] = sizeof(gp.phProof);
370     s_reservedSize[NV_SH_PROOF] = sizeof(gp.shProof);
371     s_reservedSize[NV_EH_PROOF] = sizeof(gp.ehProof);
372     s_reservedSize[NV_TOTAL_RESET_COUNT] = sizeof(gp.totalResetCount);
373     s_reservedSize[NV_RESET_COUNT] = sizeof(gp.resetCount);
374     s_reservedSize[NV_PCR_POLICIES] = sizeof(gp.pcrPolicies);
375     s_reservedSize[NV_PCR_ALLOCATED] = sizeof(gp.pcrAllocated);
376     s_reservedSize[NV_PP_LIST] = sizeof(gp.ppList);
377     s_reservedSize[NV_FAILED_TRIES] = sizeof(gp.failedTries);
378     s_reservedSize[NV_MAX_TRIES] = sizeof(gp.maxTries);
379     s_reservedSize[NV_RECOVERY_TIME] = sizeof(gp.recoveryTime);
380     s_reservedSize[NV_LOCKOUT_RECOVERY] = sizeof(gp.lockoutRecovery);
381     s_reservedSize[NV_LOCKOUT_AUTH_ENABLED] = sizeof(gp.lockOutAuthEnabled);
382     s_reservedSize[NV_ORDERLY] = sizeof(gp.orderlyState);
383     s_reservedSize[NV_AUDIT_COMMANDS] = sizeof(gp.auditComands);
384     s_reservedSize[NV_AUDIT_HASH_ALG] = sizeof(gp.auditHashAlg);
385     s_reservedSize[NV_AUDIT_COUNTER] = sizeof(gp.auditCounter);
386     s_reservedSize[NV_ALGORITHM_SET] = sizeof(gp.algorithmSet);
387     s_reservedSize[NV_FIRMWARE_V1] = sizeof(gp.firmwareV1);
388     s_reservedSize[NV_FIRMWARE_V2] = sizeof(gp.firmwareV2);
389     s_reservedSize[NV_ORDERLY_DATA] = sizeof(go);
390     s_reservedSize[NV_STATE_CLEAR] = sizeof(gc);
391     s_reservedSize[NV_STATE_RESET] = sizeof(gr);
392 
393     // Initialize reserved data address.  In this implementation, reserved data
394     // is stored at the start of NV memory
395     reservedAddr = 0;
396     for(i = 0; i < NV_RESERVE_LAST; i++)
397     {
398         s_reservedAddr[i] = reservedAddr;
399         reservedAddr += s_reservedSize[i];
400     }
401 
402     // Initialize auxiliary variable space for index/evict implementation.
403     // Auxiliary variables are stored after reserved data area
404     // RAM index copy starts at the beginning
405     s_ramIndexSizeAddr = reservedAddr;
```

```
406         s_ramIndexAddr = s_ramIndexSizeAddr + sizeof(UINT32);
407
408         // Maximum counter value
409         s_maxCountAddr = s_ramIndexAddr + RAM_INDEX_SPACE;
410
411         // dynamic memory start
412         s_evictNvStart = s_maxCountAddr + sizeof(UINT64);
413
414         // dynamic memory ends at the end of NV memory
415         s_evictNvEnd = NV_MEMORY_SIZE;
416
417         return;
418     }
```

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.

```
419     void
420     NvInit(
421         void
422         )
423     {
424         UINT32      nullPointer = 0;
425         UINT64      zeroCounter = 0;
426
427         // Initialize static variables
428         NvInitStatic();
429
430         // Initialize RAM index space as unused
431         _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &nullPointer);
432
433         // Initialize max counter value to 0
434         _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &zeroCounter);
435
436         // Initialize the next offset of the first entry in evict/index list to 0
437         _plat__NvMemoryWrite(s_evictNvStart, sizeof(TPM_HANDLE), &nullPointer);
438
439         return;
440
441     }
```

8.4.6.3 NvReadReserved()

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

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

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.

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

8.4.6.5 NvReadPersistent()

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

```
469 void
470 NvReadPersistent(
471     void
472     )
473 {
474     // Hierarchy persistent data
475     NvReadReserved(NV_DISABLE_CLEAR, &gp.disableClear);
476     NvReadReserved(NV_OWNER_ALG, &gp.ownerAlg);
477     NvReadReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
478     NvReadReserved(NV_LOCKOUT_ALG, &gp.lockoutAlg);
479     NvReadReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
480     NvReadReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
481     NvReadReserved(NV_LOCKOUT_POLICY, &gp.lockoutPolicy);
482     NvReadReserved(NV_OWNER_AUTH, &gp.ownerAuth);
483     NvReadReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
484     NvReadReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
485     NvReadReserved(NV_EP_SEED, &gp.EPSeed);
486     NvReadReserved(NV_SP_SEED, &gp.SPSeed);
487     NvReadReserved(NV_PP_SEED, &gp.PPSeed);
488     NvReadReserved(NV_PH_PROOF, &gp.phProof);
489     NvReadReserved(NV_SH_PROOF, &gp.shProof);
490     NvReadReserved(NV_EH_PROOF, &gp.ehProof);
491
492     // Time persistent data
493     NvReadReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
494     NvReadReserved(NV_RESET_COUNT, &gp.resetCount);
495
496     // PCR persistent data
497     NvReadReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
498     NvReadReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
499
500     // Physical Presence persistent data
501     NvReadReserved(NV_PP_LIST, &gp.ppList);
502
503     // Dictionary attack values persistent data
504     NvReadReserved(NV_FAILED_TRIES, &gp.failedTries);
505     NvReadReserved(NV_MAX_TRIES, &gp.maxTries);
506     NvReadReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
```



```
507         NvReadReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
508         NvReadReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
509
510         // Orderly State persistent data
511         NvReadReserved(NV_ORDERLY, &gp.orderlyState);
512
513         // Command audit values persistent data
514         NvReadReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
515         NvReadReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
516         NvReadReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
517
518         // Algorithm selection persistent data
519         NvReadReserved(NV_ALGORITHM_SET, &gp.algorithmSet);
520
521         // Firmware version persistent data
522         NvReadReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
523         NvReadReserved(NV_FIRMWARE_V2, &gp.firmwareV2);
524
525         return;
526     }
```

8.4.6.6 NvIsPlatformPersistentHandle()

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

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

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

8.4.6.7 NvIsOwnerPersistentHandle()

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

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

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

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.

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

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.

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

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.

```
579 static UINT32
580 NvFindHandle(
581     TPM_HANDLE       handle
582     )
583 {
584     UINT32          addr;
585     NV_ITER         iter = NV_ITER_INIT;
586
587     while((addr = NvNext(&iter)) != 0)
588     {
589         TPM_HANDLE          entityHandle;
590         // Read handle
```

```
591             _plat__NvMemoryRead(addr, sizeof(TPM_HANDLE), &entityHandle);
592             if(entityHandle == handle)
593                 return addr;
594         }
595
596         pAssert(addr == 0);
597         return addr;
598     }
```

8.4.6.11 NvPowerOn()

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

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

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

8.4.6.12 NvStateSave()

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

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

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.

```
632 void
633 NvEntityStartup(
634     STARTUP_TYPE     type           // IN: start up type
635     )
636 {
637     NV_ITER             iter = NV_ITER_INIT;
638     UINT32              currentAddr;        // offset points to the current entity
639 
640     // Restore RAM index data
641     _plat__NvMemoryRead(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
642     _plat__NvMemoryRead(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
643 
644     // If recovering from state save, do nothing
645     if(type == SU_RESUME)
646         return;
647 
648     // Iterate all the NV Index to clear the locks
649     while((currentAddr = NvNextIndex(&iter)) != 0)
650     {
651         NV_INDEX    nvIndex;
652         UINT32      indexAddr;              // NV address points to index info
653         TPMA_NV     attributes;
654 
655         indexAddr = currentAddr + sizeof(TPM_HANDLE);
656 
657         // Read NV Index info structure
658         _plat__NvMemoryRead(indexAddr, sizeof(NV_INDEX), &nvIndex);
659         attributes = nvIndex.publicArea.attributes;
660 
661         // Clear read/write lock
662         if(attributes.TPMA_NV_READLOCKED == SET)
663             attributes.TPMA_NV_READLOCKED = CLEAR;
664 
665         if(     attributes.TPMA_NV_WRITELOCKED == SET
666             &&  (   attributes.TPMA_NV_WRITTEN == CLEAR
667                 ||  attributes.TPMA_NV_WRITEDEFINE == CLEAR
668                 )
669            )
670             attributes.TPMA_NV_WRITELOCKED = CLEAR;
671 
672         // Reset NV data for TPMA_NV_CLEAR_STCLEAR
673         if(attributes.TPMA_NV_CLEAR_STCLEAR == SET)
674         {
675             attributes.TPMA_NV_WRITTEN = CLEAR;
676             attributes.TPMA_NV_WRITELOCKED = CLEAR;
677         }
678 
679         // Reset NV data for orderly values that are not counters
680         // NOTE: The function has already exited on a TPM Resume, so the only
681         // things being processed are TPM Restart and TPM Reset
682         if(     type == SU_RESET
683             &&  attributes.TPMA_NV_ORDERLY == SET
684             &&  attributes.TPMA_NV_COUNTER == CLEAR
```



```
685             )
686               attributes.TPMA_NV_WRITTEN = CLEAR;
687
688           // Write NV Index info back if it has changed
689           if(*((UINT32 *)&attributes) != *((UINT32 *)&nvIndex.publicArea.attributes))
690           {
691               nvIndex.publicArea.attributes = attributes;
692               _plat__NvMemoryWrite(indexAddr, sizeof(NV_INDEX), &nvIndex);
693
694               // Set the flag that a NV write happens
695               g_updateNV = TRUE;
696           }
697           // Set the lower bits in an orderly counter to 1 for a non-orderly startup
698           if(   g_prevOrderlyState == SHUTDOWN_NONE
699              && attributes.TPMA_NV_WRITTEN == SET)
700           {
701               if(   attributes.TPMA_NV_ORDERLY == SET
702                  && attributes.TPMA_NV_COUNTER == SET)
703               {
704                   TPMI_RH_NV_INDEX    nvHandle;
705                   UINT64              counter;
706
707                   // Read NV handle
708                   _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);
709
710                   // Read the counter value saved to NV upon the last roll over.
711                   // Do not use RAM backed storage for this once.
712                   nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = CLEAR;
713                   NvGetIntIndexData(nvHandle, &nvIndex, &counter);
714                   nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = SET;
715
716                   // Set the lower bits of counter to 1's
717                   counter |= MAX_ORDERLY_COUNT;
718
719                   // Write back to RAM
720                   NvWriteIndexData(nvHandle, &nvIndex, 0, sizeof(counter), &counter);
721
722                   // No write to NV because an orderly shutdown will update the
723                   // counters.
724
725               }
726           }
727       }
728
729       return;
730
731   }
```

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. This is only used by TPM2_NV_DefineSpace().

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

```
732 BOOL
733 NvIsUndefinedIndex(
734     TPMI_RH_NV_INDEX     handle          // IN: handle
735     )
736 {
737     UINT32          entityAddr;         // offset points to the entity
738
739     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
740
741     // Find the address of index
742     entityAddr = NvFindHandle(handle);
743
744     // If handle is not found, return TPM_RC_SUCCESS
745     if(entityAddr == 0)
746         return TPM_RC_SUCCESS;
747
748     // NV Index is defined
749     return TPM_RC_NV_DEFINED;
750 }
```

8.4.7.3 NvIndexIsAccessible()

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

Error Returns	Meaning
TPM_RC_HANDLE	the handle points to an undefined NV Index If *shEnable* is CLEAR, this would include an index created using *ownerAuth*. If *phEnableNV* is CLEAR, this would include and index created using platform auth
TPM_RC_NV_READLOCKED	Index is present but locked for reading and command does not write to the index
TPM_RC_NV_WRITELOCKED	Index is present but locked for writing and command writes to the index

```
751 TPM_RC
752 NvIndexIsAccessible(
753     TPMI_RH_NV_INDEX     handle,         // IN: handle
754     TPM_CC               commandCode     // IN: the command
755     )
756 {
757     UINT32              entityAddr;     // offset points to the entity
758     NV_INDEX            nvIndex;        //
759
760     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
761
762     // Find the address of index
763     entityAddr = NvFindHandle(handle);
764
765     // If handle is not found, return TPM_RC_HANDLE
766     if(entityAddr == 0)
767         return TPM_RC_HANDLE;
768
769     // Read NV Index info structure
770     _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
771                         &nvIndex);
```

```
772
773         if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
774         {
775             // if shEnable is CLEAR, an ownerCreate NV Index should not be
776             // indicated as present
777             if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
778             {
779                 if(gc.shEnable == FALSE)
780                     return TPM_RC_HANDLE;
781             }
782             // if phEnableNV is CLEAR, a platform created Index should not
783             // be visible
784             else if(gc.phEnableNV == FALSE)
785                 return TPM_RC_HANDLE;
786         }
787
788         // If the Index is write locked and this is an NV Write operation...
789         if(     nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED
790             &&  IsWriteOperation(commandCode))
791         {
792             // then return a locked indication unless the command is TPM2_NV_WriteLock
793             if(commandCode != TPM_CC_NV_WriteLock)
794                 return TPM_RC_NV_LOCKED;
795             return TPM_RC_SUCCESS;
796         }
797         // If the Index is read locked and this is an NV Read operation...
798         if(     nvIndex.publicArea.attributes.TPMA_NV_READLOCKED
799             && IsReadOperation(commandCode))
800         {
801             // then return a locked indication unless the command is TPM2_NV_ReadLock
802             if(commandCode != TPM_CC_NV_ReadLock)
803                 return TPM_RC_NV_LOCKED;
804             return TPM_RC_SUCCESS;
805         }
806
807         // NV Index is accessible
808         return TPM_RC_SUCCESS;
809     }
```

8.4.7.4 NvIsUndefinedEvictHandle()

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

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

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

```
824         else
825             return FALSE;
826     }
```

8.4.7.5 NvGetEvictObject()

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

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

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

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.

```
856     void
857     NvGetIndexInfo(
858         TPMI_RH_NV_INDEX     handle,         // IN: handle
859         NV_INDEX             *nvIndex        // OUT: NV index structure
860         )
861     {
862         UINT32               entityAddr;     // offset points to the entity
863
864         pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
865
866         // Find the address of NV index
```

```
867         entityAddr = NvFindHandle(handle);
868         pAssert(entityAddr != 0);
869
870         // This implementation uses the default format so just
871         // read the data in
872         _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
873                             nvIndex);
874
875         return;
876     }
```

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.

```
877     UINT64
878     NvInitialCounter(
879         void
880         )
881     {
882         UINT64          maxCount;
883         NV_ITER         iter = NV_ITER_INIT;
884         UINT32          currentAddr;
885
886         // Read the maxCount value
887         maxCount = NvReadMaxCount();
888
889         // Iterate all existing counters
890         while((currentAddr = NvNextIndex(&iter)) != 0)
891         {
892             TPMI_RH_NV_INDEX    nvHandle;
893             NV_INDEX            nvIndex;
894
895             // Read NV handle
896             _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);
897
898             // Get NV Index
899             NvGetIndexInfo(nvHandle, &nvIndex);
900             if(    nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
901                 && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
902             {
903                 UINT64      countValue;
904                 // Read counter value
905                 NvGetIntIndexData(nvHandle, &nvIndex, &countValue);
906                 if(countValue > maxCount)
907                     maxCount = countValue;
908             }
909         }
910         // Initialize the new counter value to be maxCount + 1
911         // A counter is only initialized the first time it is written. The
912         // way to write a counter is with TPM2_NV_INCREMENT(). Since the
913         // "initial" value of a defined counter is the largest count value that
914         // may have existed in this index previously, then the first use would
915         // add one to that value.
916         return maxCount;
917     }
```

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.

```
void
NvGetIndexData(
    TPMI_RH_NV_INDEX     handle,         // IN: handle
    NV_INDEX            *nvIndex,        // IN: RAM image of index header
    UINT32               offset,         // IN: offset of NV data
    UINT16               size,           // IN: size of NV data
    void                *data            // OUT: data buffer
    )
{

    pAssert(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET);

    if(   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
       || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET)
    {
        // Read bit or counter data in canonical form
        UINT64      dataInInt;
        NvGetIntIndexData(handle, nvIndex, &dataInInt);
        UINT64_TO_BYTE_ARRAY(dataInInt, (BYTE *)data);
    }
    else
    {
        if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
        {
            UINT32      ramAddr;

            // Get data from RAM buffer
            ramAddr = NvGetRAMIndexOffset(handle);
            MemoryCopy(data, s_ramIndex + ramAddr + offset, size, size);
        }
        else
        {
            UINT32      entityAddr;
            entityAddr = NvFindHandle(handle);
            // Get data from NV
            // Skip NV Index info, read data buffer
            entityAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
            // Read the data
            _plat__NvMemoryRead(entityAddr, size, data);
        }
    }
    return;
}
```

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.

```
void
NvGetIntIndexData(
    TPMI_RH_NV_INDEX     handle,         // IN: handle
    NV_INDEX            *nvIndex,        // IN: RAM image of NV Index header
    UINT64              *data            // IN: UINT64 pointer for counter or bit
    )
{
    // Validate that index has been written and is the right type
    pAssert(   nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET
            && (   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
                || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET
```

```
                )
            );

    // bit and counter value is store in native format for TPM CPU.  So we directly
    // copy the contents of NV to output data buffer
    if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
    {
        UINT32      ramAddr;

        // Get data from RAM buffer
        ramAddr = NvGetRAMIndexOffset(handle);
        MemoryCopy(data, s_ramIndex + ramAddr, sizeof(*data), sizeof(*data));
    }
    else
    {
        UINT32      entityAddr;
        entityAddr = NvFindHandle(handle);

        // Get data from NV
        // Skip NV Index info, read data buffer
        _plat__NvMemoryRead(
            entityAddr + sizeof(TPM_HANDLE) + sizeof(NV_INDEX),
            sizeof(UINT64), data);
    }

    return;
}
```

8.4.7.10 NvWriteIndexInfo()

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

This function requires that NV Index is defined.

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

```
TPM_RC
NvWriteIndexInfo(
    TPMI_RH_NV_INDEX     handle,         // IN: handle
    NV_INDEX            *nvIndex         // IN: NV Index info to be written
    )
{
    UINT32      entryAddr;
    TPM_RC      result;

    // Get the starting offset for the index in the RAM image of NV
    entryAddr = NvFindHandle(handle);
    pAssert(entryAddr != 0);

    // Step over the link value
    entryAddr = entryAddr + sizeof(TPM_HANDLE);

    // If the index data is actually changed, then a write to NV is required
    if(_plat__NvIsDifferent(entryAddr, sizeof(NV_INDEX),nvIndex))
    {
        // Make sure that NV is available
        result = NvIsAvailable();
        if(result != TPM_RC_SUCCESS)
            return result;
        _plat__NvMemoryWrite(entryAddr, sizeof(NV_INDEX), nvIndex);
        g_updateNV = TRUE;
```

```
1024        }
1025        return TPM_RC_SUCCESS;
1026    }
```

8.4.7.11 NvWriteIndexData()

This function is used to write NV index data.

This function requires that the NV Index is defined, and the data is within the defined data range for the index.

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

```
1027    TPM_RC
1028    NvWriteIndexData(
1029        TPMI_RH_NV_INDEX     handle,         // IN: handle
1030        NV_INDEX            *nvIndex,        // IN: RAM copy of NV Index
1031        UINT32               offset,         // IN: offset of NV data
1032        UINT32               size,           // IN: size of NV data
1033        void                *data            // OUT: data buffer
1034        )
1035    {
1036        TPM_RC               result;
1037        // Validate that write falls within range of the index
1038        pAssert(nvIndex->publicArea.dataSize >= offset + size);
1039
1040        // Update TPMA_NV_WRITTEN bit if necessary
1041        if(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
1042        {
1043            nvIndex->publicArea.attributes.TPMA_NV_WRITTEN = SET;
1044            result = NvWriteIndexInfo(handle, nvIndex);
1045            if(result != TPM_RC_SUCCESS)
1046                return result;
1047        }
1048
1049        // Check to see if process for an orderly index is required.
1050        if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
1051        {
1052            UINT32      ramAddr;
1053
1054            // Write data to RAM buffer
1055            ramAddr = NvGetRAMIndexOffset(handle);
1056            MemoryCopy(s_ramIndex + ramAddr + offset, data, size,
1057                       sizeof(s_ramIndex) - ramAddr - offset);
1058
1059            // NV update does not happen for orderly index.  Have
1060            // to clear orderlyState to reflect that we have changed the
1061            // NV and an orderly shutdown is required. Only going to do this if we
1062            // are not processing a counter that has just rolled over
1063            if(g_updateNV == FALSE)
1064                g_clearOrderly = TRUE;
1065        }
1066        // Need to process this part if the Index isn't orderly or if it is
1067        // an orderly counter that just rolled over.
1068        if(g_updateNV || nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == CLEAR)
1069        {
1070            // Processing for an index with TPMA_NV_ORDERLY CLEAR
1071            UINT32      entryAddr = NvFindHandle(handle);
1072
1073            pAssert(entryAddr != 0);
```

```
1074
1075            // Offset into the index to the first byte of the data to be written
1076            entryAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
1077
1078            // If the data is actually changed, then a write to NV is required
1079            if(_plat__NvIsDifferent(entryAddr, size, data))
1080            {
1081                // Make sure that NV is available
1082                result = NvIsAvailable();
1083                if(result != TPM_RC_SUCCESS)
1084                    return result;
1085                _plat__NvMemoryWrite(entryAddr, size, data);
1086                g_updateNV = TRUE;
1087            }
1088        }
1089        return TPM_RC_SUCCESS;
1090    }
```

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.

```
1091    UINT16
1092    NvGetName(
1093        TPMI_RH_NV_INDEX     handle,        // IN: handle of the index
1094        NAME                *name           // OUT: name of the index
1095        )
1096    {
1097        UINT16               dataSize, digestSize;
1098        NV_INDEX             nvIndex;
1099        BYTE                 marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
1100        BYTE                *buffer;
1101        HASH_STATE           hashState;
1102
1103        // Get NV public info
1104        NvGetIndexInfo(handle, &nvIndex);
1105
1106        // Marshal public area
1107        buffer = marshalBuffer;
1108        dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex.publicArea, &buffer, NULL);
1109
1110        // hash public area
1111        digestSize = CryptStartHash(nvIndex.publicArea.nameAlg, &hashState);
1112        CryptUpdateDigest(&hashState, dataSize, marshalBuffer);
1113
1114        // Complete digest leaving room for the nameAlg
1115        CryptCompleteHash(&hashState, digestSize, &((BYTE *)name)[2]);
1116
1117        // Include the nameAlg
1118        UINT16_TO_BYTE_ARRAY(nvIndex.publicArea.nameAlg, (BYTE *)name);
1119        return digestSize + 2;
1120    }
```

8.4.7.13 NvDefineIndex()

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

Error Returns	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```
1121    TPM_RC
1122    NvDefineIndex(
1123        TPMS_NV_PUBLIC  *publicArea,    // IN: A template for an area to create.
1124        TPM2B_AUTH      *authValue      // IN: The initial authorization value
1125        )
1126    {
1127        // The buffer to be written to NV memory
1128        BYTE            nvBuffer[sizeof(TPM_HANDLE) + sizeof(NV_INDEX)];
1129
1130        NV_INDEX        *nvIndex;           // a pointer to the NV_INDEX data in
1131                                            //   nvBuffer
1132        UINT16          entrySize;          // size of entry
1133
1134        entrySize = sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + publicArea->dataSize;
1135
1136        // Check if we have enough space to create the NV Index
1137        // In this implementation, the only resource limitation is the available NV
1138        // space.  Other implementation may have other limitation on counter or on
1139        // NV slot
1140        if(!NvTestSpace(entrySize, TRUE)) return TPM_RC_NV_SPACE;
1141
1142        // if the index to be defined is RAM backed, check RAM space availability
1143        // as well
1144        if(publicArea->attributes.TPMA_NV_ORDERLY == SET
1145                && !NvTestRAMSpace(publicArea->dataSize))
1146            return TPM_RC_NV_SPACE;
1147
1148        // Copy input value to nvBuffer
1149            // Copy handle
1150        * (TPM_HANDLE *) nvBuffer = publicArea->nvIndex;
1151
1152            // Copy NV_INDEX
1153        nvIndex = (NV_INDEX *) (nvBuffer + sizeof(TPM_HANDLE));
1154        nvIndex->publicArea = *publicArea;
1155        nvIndex->authValue = *authValue;
1156
1157        // Add index to NV memory
1158        NvAdd(entrySize, sizeof(TPM_HANDLE) + sizeof(NV_INDEX), nvBuffer);
1159
1160        // If the data of NV Index is RAM backed, add the data area in RAM as well
1161        if(publicArea->attributes.TPMA_NV_ORDERLY == SET)
1162            NvAddRAM(publicArea->nvIndex, publicArea->dataSize);
1163
1164        return TPM_RC_SUCCESS;
1165    }
```

8.4.7.14 NvAddEvictObject()

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

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

```
1166    TPM_RC
1167    NvAddEvictObject(
1168        TPMI_DH_OBJECT   evictHandle,   // IN: new evict handle
```

```
1169        OBJECT          *object         // IN: object to be added
1170        )
1171    {
1172        // The buffer to be written to NV memory
1173        BYTE            nvBuffer[sizeof(TPM_HANDLE) + sizeof(OBJECT)];
1174
1175        OBJECT          *nvObject;          // a pointer to the OBJECT data in
1176                                            // nvBuffer
1177        UINT16          entrySize;          // size of entry
1178
1179        // evict handle type should match the object hierarchy
1180        pAssert(    (   NvIsPlatformPersistentHandle(evictHandle)
1181                     && object->attributes.ppsHierarchy == SET)
1182                || (   NvIsOwnerPersistentHandle(evictHandle)
1183                     && (   object->attributes.spsHierarchy == SET
1184                         || object->attributes.epsHierarchy == SET)));
1185
1186        // An evict needs 4 bytes of handle + sizeof OBJECT
1187        entrySize = sizeof(TPM_HANDLE) + sizeof(OBJECT);
1188
1189        // Check if we have enough space to add the evict object
1190        // An evict object needs 8 bytes in index table + sizeof OBJECT
1191        // In this implementation, the only resource limitation is the available NV
1192        // space.  Other implementation may have other limitation on evict object
1193        // handle space
1194        if(!NvTestSpace(entrySize, FALSE)) return TPM_RC_NV_SPACE;
1195
1196        // Allocate a new evict handle
1197        if(!NvIsUndefinedEvictHandle(evictHandle))
1198            return TPM_RC_NV_DEFINED;
1199
1200        // Copy evict object to nvBuffer
1201            // Copy handle
1202        * (TPM_HANDLE *) nvBuffer = evictHandle;
1203
1204            // Copy OBJECT
1205        nvObject = (OBJECT *) (nvBuffer + sizeof(TPM_HANDLE));
1206        *nvObject = *object;
1207
1208        // Set evict attribute and handle
1209        nvObject->attributes.evict = SET;
1210        nvObject->evictHandle = evictHandle;
1211
1212        // Add evict to NV memory
1213        NvAdd(entrySize, entrySize, nvBuffer);
1214
1215        return TPM_RC_SUCCESS;
1216
1217    }
```

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.

```
1218    void
1219    NvDeleteEntity(
1220        TPM_HANDLE       handle         // IN: handle of entity to be deleted
1221        )
1222    {
1223        UINT32      entityAddr;     // pointer to entity
1224
1225        entityAddr = NvFindHandle(handle);
1226        pAssert(entityAddr != 0);
```

```
1227
1228        if(HandleGetType(handle) == TPM_HT_NV_INDEX)
1229        {
1230            NV_INDEX    nvIndex;
1231
1232            // Read the NV Index info
1233            _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
1234                                &nvIndex);
1235
1236            // If the entity to be deleted is a counter with the maximum counter
1237            // value, record it in NV memory
1238            if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
1239                    && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
1240            {
1241                UINT64      countValue;
1242                UINT64      maxCount;
1243                NvGetIntIndexData(handle, &nvIndex, &countValue);
1244                maxCount = NvReadMaxCount();
1245                if(countValue > maxCount)
1246                    NvWriteMaxCount(countValue);
1247            }
1248            // If the NV Index is RAM back, delete the RAM data as well
1249            if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
1250                NvDeleteRAM(handle);
1251        }
1252        NvDelete(entityAddr);
1253
1254        return;
1255
1256    }
```

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*.

```
1257    void
1258    NvFlushHierarchy(
1259        TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flushed.
1260        )
1261    {
1262        NV_ITER         iter = NV_ITER_INIT;
1263        UINT32          currentAddr;
1264
1265        while((currentAddr = NvNext(&iter)) != 0)
1266        {
1267            TPM_HANDLE      entityHandle;
1268
1269            // Read handle information.
1270            _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1271
1272            if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
1273            {
1274                // Handle NV Index
1275                NV_INDEX    nvIndex;
1276
1277                // If flush endorsement or platform hierarchy, no NV Index would be
1278                // flushed
1279                if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
1280                    continue;
1281                _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1282                                    sizeof(NV_INDEX), &nvIndex);
1283
1284                // For storage hierarchy, flush OwnerCreated index
```

```
1285                if(  nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
1286                {
1287                    // Delete the NV Index
1288                    NvDelete(currentAddr);
1289
1290                    // Re-iterate from beginning after a delete
1291                    iter = NV_ITER_INIT;
1292
1293                    // If the NV Index is RAM back, delete the RAM data as well
1294                    if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
1295                        NvDeleteRAM(entityHandle);
1296                }
1297            }
1298            else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
1299            {
1300                OBJECT          object;
1301
1302                // Get evict object
1303                NvGetEvictObject(entityHandle, &object);
1304
1305                // If the evict object belongs to the hierarchy to be flushed
1306                if(    (    hierarchy == TPM_RH_PLATFORM
1307                        &&  object.attributes.ppsHierarchy == SET)
1308                    || (    hierarchy == TPM_RH_OWNER
1309                        &&  object.attributes.spsHierarchy == SET)
1310                    || (    hierarchy == TPM_RH_ENDORSEMENT
1311                        &&  object.attributes.epsHierarchy == SET)
1312                    )
1313                {
1314                    // Delete the evict object
1315                    NvDelete(currentAddr);
1316
1317                    // Re-iterate from beginning after a delete
1318                    iter = NV_ITER_INIT;
1319                }
1320            }
1321            else
1322            {
1323                pAssert(FALSE);
1324            }
1325        }
1326
1327        return;
1328    }
```

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().

```
1329    void
1330    NvSetGlobalLock(
1331        void
1332        )
1333    {
1334        NV_ITER         iter = NV_ITER_INIT;
1335        UINT32          currentAddr;
1336
1337        // Check all Indices
1338        while((currentAddr = NvNextIndex(&iter)) != 0)
1339        {
1340            NV_INDEX    nvIndex;
1341
1342            // Read the index data
```

```
1343            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1344                                sizeof(NV_INDEX), &nvIndex);
1345
1346            // See if it should be locked
1347            if(nvIndex.publicArea.attributes.TPMA_NV_GLOBALLOCK == SET)
1348            {
1349
1350                // if so, lock it
1351                nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED = SET;
1352
1353                _plat__NvMemoryWrite(currentAddr + sizeof(TPM_HANDLE),
1354                                     sizeof(NV_INDEX), &nvIndex);
1355                // Set the flag that a NV write happens
1356                g_updateNV = TRUE;
1357            }
1358        }
1359
1360        return;
1361
1362    }
```

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

```
1363    static void
1364    InsertSort(
1365        TPML_HANDLE     *handleList,    // IN/OUT: sorted handle list
1366        UINT32           count,         // IN: maximum count in the handle list
1367        TPM_HANDLE       entityHandle   // IN: handle to be inserted
1368        )
1369    {
1370        UINT32          i, j;
1371        UINT32          originalCount;
1372
1373        // For a corner case that the maximum count is 0, do nothing
1374        if(count == 0) return;
1375
1376        // For empty list, add the handle at the beginning and return
1377        if(handleList->count == 0)
1378        {
1379            handleList->handle[0] = entityHandle;
1380            handleList->count++;
1381            return;
1382        }
1383
1384        // Check if the maximum of the list has been reached
1385        originalCount = handleList->count;
1386        if(originalCount < count)
1387            handleList->count++;
1388
1389        // Insert the handle to the list
1390        for(i = 0; i < originalCount; i++)
1391        {
1392            if(handleList->handle[i] > entityHandle)
1393            {
1394                for(j = handleList->count - 1; j > i; j--)
1395                {
1396                    handleList->handle[j] = handleList->handle[j-1];
1397                }
1398                break;
1399            }
1400        }
```

```
1401
1402        // If a slot was found, insert the handle in this position
1403        if(i < originalCount || handleList->count > originalCount)
1404            handleList->handle[i] = entityHandle;
1405
1406        return;
1407    }
```

8.4.7.19 NvCapGetPersistent()

This function is used to get a list of handles of the persistent objects, starting at *handle*.

Handle must be in valid persistent object handle range, but does not have to reference an existing persistent object.

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

```
1408    TPMI_YES_NO
1409    NvCapGetPersistent(
1410        TPMI_DH_OBJECT   handle,        // IN: start handle
1411        UINT32           count,         // IN: maximum number of returned handle
1412        TPML_HANDLE     *handleList     // OUT: list of handle
1413        )
1414    {
1415        TPMI_YES_NO          more = NO;
1416        NV_ITER              iter = NV_ITER_INIT;
1417        UINT32               currentAddr;
1418
1419        pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1420
1421        // Initialize output handle list
1422        handleList->count = 0;
1423
1424        // The maximum count of handles we may return is MAX_CAP_HANDLES
1425        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1426
1427        while((currentAddr = NvNextEvict(&iter)) != 0)
1428        {
1429            TPM_HANDLE      entityHandle;
1430
1431            // Read handle information.
1432            _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1433
1434            // Ignore persistent handles that have values less than the input handle
1435            if(entityHandle < handle)
1436                continue;
1437
1438            // if the handles in the list have reached the requested count, and there
1439            // are still handles need to be inserted, indicate that there are more.
1440            if(handleList->count == count)
1441                more = YES;
1442
1443            // A handle with a value larger than start handle is a candidate
1444            // for return. Insert sort it to the return list.  Insert sort algorithm
1445            // is chosen here for simplicity based on the assumption that the total
1446            // number of NV Indices is small.  For an implementation that may allow
1447            // large number of NV Indices, a more efficient sorting algorithm may be
1448            // used here.
1449            InsertSort(handleList, count, entityHandle);
1450
```

```
1451        }
1452        return more;
1453    }
```

8.4.7.20 NvCapGetIndex()

This function returns a list of handles of NV Indices, starting from *handle*. *Handle* must be in the range of NV Indices, but does not have to reference an existing NV Index.

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

```
1454    TPMI_YES_NO
1455    NvCapGetIndex(
1456        TPMI_DH_OBJECT   handle,        // IN: start handle
1457        UINT32           count,         // IN: maximum number of returned handle
1458        TPML_HANDLE     *handleList     // OUT: list of handle
1459        )
1460    {
1461        TPMI_YES_NO          more = NO;
1462        NV_ITER              iter = NV_ITER_INIT;
1463        UINT32               currentAddr;
1464
1465        pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1466
1467        // Initialize output handle list
1468        handleList->count = 0;
1469
1470        // The maximum count of handles we may return is MAX_CAP_HANDLES
1471        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1472
1473        while((currentAddr = NvNextIndex(&iter)) != 0)
1474        {
1475            TPM_HANDLE      entityHandle;
1476
1477            // Read handle information.
1478            _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1479
1480            // Ignore index handles that have values less than the 'handle'
1481            if(entityHandle < handle)
1482                continue;
1483
1484            // if the count of handles in the list has reached the requested count,
1485            // and there are still handles to report, set more.
1486            if(handleList->count == count)
1487                more = YES;
1488
1489            // A handle with a value larger than start handle is a candidate
1490            // for return. Insert sort it to the return list.  Insert sort algorithm
1491            // is chosen here for simplicity based on the assumption that the total
1492            // number of NV Indices is small.  For an implementation that may allow
1493            // large number of NV Indices, a more efficient sorting algorithm may be
1494            // used here.
1495            InsertSort(handleList, count, entityHandle);
1496        }
1497        return more;
1498    }
```

8.4.7.21 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```
1499    UINT32
1500    NvCapGetIndexNumber(
1501        void
1502        )
1503    {
1504        UINT32          num = 0;
1505        NV_ITER         iter = NV_ITER_INIT;
1506
1507        while(NvNextIndex(&iter) != 0) num++;
1508
1509        return num;
1510    }
```

8.4.7.22 NvCapGetPersistentNumber()

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

```
1511    UINT32
1512    NvCapGetPersistentNumber(
1513        void
1514        )
1515    {
1516        UINT32          num = 0;
1517        NV_ITER         iter = NV_ITER_INIT;
1518
1519        while(NvNextEvict(&iter) != 0) num++;
1520
1521        return num;
1522    }
```

8.4.7.23 NvCapGetPersistentAvail()

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

```
1523    UINT32
1524    NvCapGetPersistentAvail(
1525        void
1526        )
1527    {
1528        UINT32          availSpace;
1529        UINT32          objectSpace;
1530
1531        // Compute the available space in NV storage
1532        availSpace = NvGetFreeByte();
1533
1534        // Get the space needed to add a persistent object to NV storage
1535        objectSpace = NvGetEvictObjectSize();
1536
1537        return availSpace / objectSpace;
1538    }
```

8.4.7.24 NvCapGetCounterNumber()

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

```
1539    UINT32
1540    NvCapGetCounterNumber(
1541        void
1542        )
1543    {
1544        NV_ITER         iter = NV_ITER_INIT;
1545        UINT32          currentAddr;
1546        UINT32          num = 0;
1547
1548        while((currentAddr = NvNextIndex(&iter)) != 0)
1549        {
1550            NV_INDEX    nvIndex;
1551
1552            // Get NV Index info
1553            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1554                                sizeof(NV_INDEX), &nvIndex);
1555            if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET) num++;
1556        }
1557
1558        return num;
1559    }
```

8.4.7.25 NvCapGetCounterAvail()

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

```
1560    UINT32
1561    NvCapGetCounterAvail(
1562        void
1563        )
1564    {
1565        UINT32          availNVSpace;
1566        UINT32          availRAMSpace;
1567        UINT32          counterNVSpace;
1568        UINT32          counterRAMSpace;
1569        UINT32          persistentNum = NvCapGetPersistentNumber();
1570
1571        // Get the available space in NV storage
1572        availNVSpace = NvGetFreeByte();
1573
1574        if (persistentNum < MIN_EVICT_OBJECTS)
1575        {
1576            // Some space have to be reserved for evict object. Adjust availNVSpace.
1577            UINT32      reserved = (MIN_EVICT_OBJECTS - persistentNum)
1578                                   * NvGetEvictObjectSize();
1579            if (reserved > availNVSpace)
1580                availNVSpace = 0;
1581            else
1582                availNVSpace -= reserved;
1583        }
1584
1585        // Get the space needed to add a counter index to NV storage
1586        counterNVSpace = NvGetCounterSize();
1587
1588        // Compute the available space in RAM
1589        availRAMSpace = RAM_INDEX_SPACE - s_ramIndexSize;
1590
1591        // Compute the space needed to add a counter index to RAM storage
1592        // It takes an size field, a handle and sizeof(UINT64) for counter data
1593        counterRAMSpace = sizeof(UINT32) + sizeof(TPM_HANDLE) + sizeof(UINT64);
1594
1595        // Return the min of counter number in NV and in RAM
1596        if(availNVSpace / counterNVSpace > availRAMSpace / counterRAMSpace)
1597            return availRAMSpace / counterRAMSpace;
```

```
1598        else
1599            return availNVSpace / counterNVSpace;
1600    }
```

8.5 Object.c

8.5.1 Introduction

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

8.5.2 Includes and Data Definitions

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

8.5.3 Functions

8.5.3.1 ObjectStartup()

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

```
4   void
5   ObjectStartup(
6       void
7       )
8   {
9       UINT32      i;
10
11      // object slots initialization
12      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
13      {
14          //Set the slot to not occupied
15          s_objects[i].occupied = FALSE;
16      }
17      return;
18  }
```

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().

```
19  void
20  ObjectCleanupEvict(
21      void
22      )
23  {
24      UINT32      i;
25
26      // This has to be iterated because a command may have two handles
27      // and they may both be persistent.
28      // This could be made to be more efficient so that a search is not needed.
29      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
30      {
31          // If an object is a temporary evict object, flush it from slot
32          if(s_objects[i].object.entity.attributes.evict == SET)
33              s_objects[i].occupied = FALSE;
34      }
```

```
35
36      return;
37  }
```

8.5.3.3 ObjectIsPresent()

This function checks to see if a transient handle references a loaded object. This routine should not be called if the handle is not a transient handle. The function validates that the handle is in the implementation-dependent allowed in range for loaded transient objects.

Return Value	Meaning
TRUE	if the handle references a loaded object
FALSE	if the handle is not an object handle, or it does not reference to a loaded object

```
38  BOOL
39  ObjectIsPresent(
40      TPMI_DH_OBJECT    handle          // IN: handle to be checked
41      )
42  {
43      UINT32          slotIndex;          // index of object slot
44
45      pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
46
47      // The index in the loaded object array is found by subtracting the first
48      // object handle number from the input handle number. If the indicated
49      // slot is occupied, then indicate that there is already is a loaded
50      //  object associated with the handle.
51      slotIndex = handle - TRANSIENT_FIRST;
52      if(slotIndex >= MAX_LOADED_OBJECTS)
53          return FALSE;
54
55      return s_objects[slotIndex].occupied;
56  }
```

8.5.3.4 ObjectIsSequence()

This function is used to check if the object is a sequence object. This function should not be called if the handle does not reference a loaded object.

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

```
57  BOOL
58  ObjectIsSequence(
59      OBJECT          *object         // IN: handle to be checked
60      )
61  {
62      pAssert (object != NULL);
63      if(   object->attributes.hmacSeq == SET
64         || object->attributes.hashSeq == SET
65         || object->attributes.eventSeq == SET)
66          return TRUE;
67      else
68          return FALSE;
69  }
```

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.

```
70  OBJECT*
71  ObjectGet(
72      TPMI_DH_OBJECT   handle         // IN: handle of the object
73      )
74  {
75      pAssert(   handle >= TRANSIENT_FIRST
76              && handle - TRANSIENT_FIRST < MAX_LOADED_OBJECTS);
77      pAssert(s_objects[handle - TRANSIENT_FIRST].occupied == TRUE);
78
79      // In this implementation, the handle is determined by the slot occupied by the
80      // object.
81      return &s_objects[handle - TRANSIENT_FIRST].object.entity;
82  }
```

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.

```
83  UINT16
84  ObjectGetName(
85      TPMI_DH_OBJECT   handle,        // IN: handle of the object
86      NAME            *name           // OUT: name of the object
87      )
88  {
89      OBJECT      *object = ObjectGet(handle);
90      if(object->publicArea.nameAlg == TPM_ALG_NULL)
91          return 0;
92
93      // Copy the Name data to the output
94      MemoryCopy(name, object->name.t.name, object->name.t.size, sizeof(NAME));
95      return  object->name.t.size;
96  }
```

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.

```
97  TPMI_ALG_HASH
98  ObjectGetNameAlg(
99      TPMI_DH_OBJECT   handle         // IN: handle of the object
100     )
101 {
102     OBJECT              *object = ObjectGet(handle);
103
104     return object->publicArea.nameAlg;
105 }
```

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.

```
106  void
107  ObjectGetQualifiedName(
108      TPMI_DH_OBJECT   handle,        // IN: handle of the object
109      TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
110      )
111  {
112      OBJECT      *object = ObjectGet(handle);
113      if(object->publicArea.nameAlg == TPM_ALG_NULL)
114          qualifiedName->t.size = 0;
115      else
116          // Copy the name
117          *qualifiedName = object->qualifiedName;
118
119      return;
120  }
```

8.5.3.9 ObjectDataGetHierarchy()

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

```
121  TPMI_RH_HIERARCHY
122  ObjectDataGetHierarchy(
123      OBJECT          *object         // IN :object
124      )
125  {
126      if(object->attributes.spsHierarchy)
127      {
128          return TPM_RH_OWNER;
129      }
130      else if(object->attributes.epsHierarchy)
131      {
132          return TPM_RH_ENDORSEMENT;
133      }
134      else if(object->attributes.ppsHierarchy)
135      {
136          return TPM_RH_PLATFORM;
137      }
138      else
139      {
140          return TPM_RH_NULL;
141      }
142
143  }
```

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.

```
144  TPMI_RH_HIERARCHY
```

```
145 ObjectGetHierarchy(
146     TPMI_DH_OBJECT   handle          // IN :object handle
147     )
148 {
149     OBJECT           *object = ObjectGet(handle);
150 
151     return ObjectDataGetHierarchy(object);
152 }
```

8.5.3.11 ObjectAllocateSlot()

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

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

```
153 static BOOL
154 ObjectAllocateSlot(
155     TPMI_DH_OBJECT  *handle,         // OUT: handle of allocated object
156     OBJECT          **object         // OUT: points to the allocated object
157     )
158 {
159     UINT32      i;
160 
161     // find an unoccupied handle slot
162     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
163     {
164         if(!s_objects[i].occupied)           // If found a free slot
165         {
166             // Mark the slot as occupied
167             s_objects[i].occupied = TRUE;
168             break;
169         }
170     }
171     // If we reach the end of object slot without finding a free one, return
172     // error.
173     if(i == MAX_LOADED_OBJECTS) return FALSE;
174 
175     *handle = i + TRANSIENT_FIRST;
176     *object = &s_objects[i].object.entity;
177 
178     // Initialize the object attributes
179     MemorySet(&((*object)->attributes), 0, sizeof(OBJECT_ATTRIBUTES));
180 
181     return TRUE;
182 }
```

8.5.3.12 ObjectLoad()

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

Error Returns	Meaning
TPM_RC_BINDING	if the public and sensitive parts of the object are not matched
TPM_RC_KEY	if the parameters in the public area of the object are not consistent
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object
TPM_RC_TYPE	the public and private parts are not the same type

```
183  TPM_RC
184  ObjectLoad(
185      TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy to which the object belongs
186      TPMT_PUBLIC         *publicArea,     // IN: public area
187      TPMT_SENSITIVE      *sensitive,      // IN: sensitive area (may be null)
188      TPM2B_NAME          *name,           // IN: object's name (may be null)
189      TPM_HANDLE           parentHandle,   // IN: handle of parent
190      BOOL                 skipChecks,     // IN: flag to indicate if it is OK to skip
191                                           //     consistency checks.
192      TPMI_DH_OBJECT      *handle          // OUT: object handle
193      )
194  {
195      OBJECT               *object = NULL;
196      OBJECT               *parent = NULL;
197      TPM_RC                result = TPM_RC_SUCCESS;
198      TPM2B_NAME            parentQN;         // Parent qualified name
199
200      // Try to allocate a slot for new object
201      if(!ObjectAllocateSlot(handle, &object))
202          return TPM_RC_OBJECT_MEMORY;
203
204      // Initialize public
205      object->publicArea = *publicArea;
206      if(sensitive != NULL)
207          object->sensitive = *sensitive;
208
209      // Are the consistency checks needed
210      if(!skipChecks)
211      {
212          // Check if key size matches
213          if(!CryptObjectIsPublicConsistent(&object->publicArea))
214          {
215              result = TPM_RC_KEY;
216              goto ErrorExit;
217          }
218          if(sensitive != NULL)
219          {
220              // Check if public type matches sensitive type
221              result = CryptObjectPublicPrivateMatch(object);
222              if(result != TPM_RC_SUCCESS)
223                  goto ErrorExit;
224          }
225      }
226      object->attributes.publicOnly = (sensitive == NULL);
227
228      // If 'name' is NULL, then there is nothing left to do for this
229      // object as it has no qualified name and it is not a member of any
230      // hierarchy and it is temporary
231      if(name == NULL || name->t.size == 0)
232      {
233          object->qualifiedName.t.size = 0;
234          object->name.t.size = 0;
235          object->attributes.temporary = SET;
236          return TPM_RC_SUCCESS;
237      }
238      // If parent handle is a permanent handle, it is a primary or temporary
```

```
239         // object
240         if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
241         {
242             // initialize QN
243             parentQN.t.size = 4;
244 
245             // for a primary key, parent qualified name is the handle of hierarchy
246             UINT32_TO_BYTE_ARRAY(parentHandle, parentQN.t.name);
247         }
248         else
249         {
250             // Get hierarchy and qualified name of parent
251             ObjectGetQualifiedName(parentHandle, &parentQN);
252 
253             // Check for stClear object
254             parent = ObjectGet(parentHandle);
255             if(   publicArea->objectAttributes.stClear == SET
256                || parent->attributes.stClear == SET)
257                 object->attributes.stClear = SET;
258 
259         }
260         object->name = *name;
261 
262         // Compute object qualified name
263         ObjectComputeQualifiedName(&parentQN, publicArea->nameAlg,
264                                    name, &object->qualifiedName);
265 
266         // Any object in TPM_RH_NULL hierarchy is temporary
267         if(hierarchy == TPM_RH_NULL)
268         {
269             object->attributes.temporary = SET;
270         }
271         else if(parentQN.t.size == sizeof(TPM_HANDLE))
272         {
273             // Otherwise, if the size of parent's qualified name is the size of a
274             // handle, this object is a primary object
275             object->attributes.primary = SET;
276         }
277         switch(hierarchy)
278         {
279             case TPM_RH_PLATFORM:
280                 object->attributes.ppsHierarchy = SET;
281                 break;
282             case TPM_RH_OWNER:
283                 object->attributes.spsHierarchy = SET;
284                 break;
285             case TPM_RH_ENDORSEMENT:
286                 object->attributes.epsHierarchy = SET;
287                 break;
288             case TPM_RH_NULL:
289                 break;
290             default:
291                 pAssert(FALSE);
292                 break;
293         }
294         return TPM_RC_SUCCESS;
295 
296     ErrorExit:
297         ObjectFlush(*handle);
298         return result;
299     }
```

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.

```
300  static BOOL
301  AllocateSequenceSlot(
302      TPM_HANDLE      *newHandle,     // OUT: receives the allocated handle
303      HASH_OBJECT     **object,       // OUT: receives pointer to allocated object
304      TPM2B_AUTH      *auth           // IN: the authValue for the slot
305      )
306  {
307      OBJECT              *objectHash;        // the hash as an object
308
309      if(!ObjectAllocateSlot(newHandle, &objectHash))
310          return FALSE;
311
312      *object = (HASH_OBJECT *)objectHash;
313
314      // Validate that the proper location of the hash state data relative to the
315      // object state data.
316      pAssert(&((*object)->auth) == &objectHash->publicArea.authPolicy);
317
318      // Set the common values that a sequence object shares with an ordinary object
319      // The type is TPM_ALG_NULL
320      (*object)->type = TPM_ALG_NULL;
321
322      // This has no name algorithm and the name is the Empty Buffer
323      (*object)->nameAlg = TPM_ALG_NULL;
324
325      // Clear the attributes
326      MemorySet(&((*object)->objectAttributes), 0, sizeof(TPMA_OBJECT));
327
328      // A sequence object is considered to be in the NULL hierarchy so it should
329      // be marked as temporary so that it can't be persisted
330      (*object)->attributes.temporary = SET;
331
332      // A sequence object is DA exempt.
333      (*object)->objectAttributes.noDA = SET;
334
335      if(auth != NULL)
336      {
337          MemoryRemoveTrailingZeros(auth);
338          (*object)->auth = *auth;
339      }
340      else
341          (*object)->auth.t.size = 0;
342      return TRUE;
343  }
```

8.5.3.14 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

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

```
344  TPM_RC
345  ObjectCreateHMACSequence(
346      TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
347      TPM_HANDLE       handle,        // IN: the handle associated with sequence
348                                      //     object
```

```
349         TPM2B_AUTH      *auth,          // IN: authValue
350         TPMI_DH_OBJECT  *newHandle      // OUT: HMAC sequence object handle
351         )
352     {
353         HASH_OBJECT         *hmacObject;
354         OBJECT              *keyObject;
355
356         // Try to allocate a slot for new object
357         if(!AllocateSequenceSlot(newHandle, &hmacObject, auth))
358             return TPM_RC_OBJECT_MEMORY;
359
360         // Set HMAC sequence bit
361         hmacObject->attributes.hmacSeq = SET;
362
363         // Get pointer to the HMAC key object
364         keyObject = ObjectGet(handle);
365
366         CryptStartHMACSequence2B(hashAlg, &keyObject->sensitive.sensitive.bits.b,
367                                  &hmacObject->state.hmacState);
368
369         return TPM_RC_SUCCESS;
370     }
```

8.5.3.15 ObjectCreateHashSequence()

This function creates a hash sequence object.

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

```
371     TPM_RC
372     ObjectCreateHashSequence(
373         TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
374         TPM2B_AUTH      *auth,          // IN: authValue
375         TPMI_DH_OBJECT  *newHandle      // OUT: sequence object handle
376         )
377     {
378         HASH_OBJECT         *hashObject;
379
380         // Try to allocate a slot for new object
381         if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
382             return TPM_RC_OBJECT_MEMORY;
383
384         // Set hash sequence bit
385         hashObject->attributes.hashSeq = SET;
386
387         // Start hash for hash sequence
388         CryptStartHashSequence(hashAlg, &hashObject->state.hashState[0]);
389
390         return TPM_RC_SUCCESS;
391     }
```

8.5.3.16 ObjectCreateEventSequence()

This function creates an event sequence object.

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

```
392     TPM_RC
```

```
393 ObjectCreateEventSequence(
394     TPM2B_AUTH      *auth,          // IN: authValue
395     TPMI_DH_OBJECT  *newHandle      // OUT: sequence object handle
396     )
397 {
398     HASH_OBJECT         *hashObject;
399     UINT32               count;
400     TPM_ALG_ID           hash;
401 
402     // Try to allocate a slot for new object
403     if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
404         return TPM_RC_OBJECT_MEMORY;
405 
406     // Set the event sequence attribute
407     hashObject->attributes.eventSeq = SET;
408 
409     // Initialize hash states for each implemented PCR algorithms
410     for(count = 0; (hash = CryptGetHashAlgByIndex(count)) != TPM_ALG_NULL; count++)
411     {
412         // If this is a _TPM_Init or _TPM_HashStart, the sequence object will
413         // not leave the TPM so it doesn't need the sequence handling
414         if(auth == NULL)
415             CryptStartHash(hash, &hashObject->state.hashState[count]);
416         else
417             CryptStartHashSequence(hash, &hashObject->state.hashState[count]);
418     }
419     return TPM_RC_SUCCESS;
420 }
```

8.5.3.17 ObjectTerminateEvent()

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

```
421 void
422 ObjectTerminateEvent(
423     void
424     )
425 {
426     HASH_OBJECT         *hashObject;
427     int                  count;
428     BYTE                 buffer[MAX_DIGEST_SIZE];
429     hashObject = (HASH_OBJECT *)ObjectGet(g_DRTMHandle);
430 
431     // Don't assume that this is a proper sequence object
432     if(hashObject->attributes.eventSeq)
433     {
434         // If it is, close any open hash contexts. This is done in case
435         // the crypto implementation has some context values that need to be
436         // cleaned up (hygiene).
437         //
438         for(count = 0; CryptGetHashAlgByIndex(count) != TPM_ALG_NULL; count++)
439         {
440             CryptCompleteHash(&hashObject->state.hashState[count], 0, buffer);
441         }
442         // Flush sequence object
443         ObjectFlush(g_DRTMHandle);
444     }
445 
446     g_DRTMHandle = TPM_RH_UNASSIGNED;
447 }
```


8.5.3.18 ObjectContextLoad()

This function loads an object from a saved object context.

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

```
448  TPM_RC
449  ObjectContextLoad(
450      OBJECT          *object,        // IN: object structure from saved context
451      TPMI_DH_OBJECT  *handle         // OUT: object handle
452      )
453  {
454      OBJECT      *newObject;
455
456      // Try to allocate a slot for new object
457      if(!ObjectAllocateSlot(handle, &newObject))
458          return TPM_RC_OBJECT_MEMORY;
459
460      // Copy input object data to internal structure
461      *newObject = *object;
462
463      return TPM_RC_SUCCESS;
464  }
```

8.5.3.19 ObjectFlush()

This function frees an object slot.

This function requires that the object is loaded.

```
465  void
466  ObjectFlush(
467      TPMI_DH_OBJECT   handle         // IN: handle to be freed
468      )
469  {
470      UINT32      index = handle - TRANSIENT_FIRST;
471      pAssert(ObjectIsPresent(handle));
472
473      // Mark the handle slot as unoccupied
474      s_objects[index].occupied = FALSE;
475
476      // With no attributes
477      MemorySet((BYTE*)&(s_objects[index].object.entity.attributes),
478                0, sizeof(OBJECT_ATTRIBUTES));
479      return;
480  }
```

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.

```
481  void
482  ObjectFlushHierarchy(
483      TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flush
484      )
485  {
486      UINT16          i;
487
488      // iterate object slots
```

```
489         for(i = 0; i < MAX_LOADED_OBJECTS; i++)
490         {
491             if(s_objects[i].occupied)           // If found an occupied slot
492             {
493                 switch(hierarchy)
494                 {
495                     case TPM_RH_PLATFORM:
496                         if(s_objects[i].object.entity.attributes.ppsHierarchy == SET)
497                             s_objects[i].occupied = FALSE;
498                         break;
499                     case TPM_RH_OWNER:
500                         if(s_objects[i].object.entity.attributes.spsHierarchy == SET)
501                             s_objects[i].occupied = FALSE;
502                         break;
503                     case TPM_RH_ENDORSEMENT:
504                         if(s_objects[i].object.entity.attributes.epsHierarchy == SET)
505                             s_objects[i].occupied = FALSE;
506                         break;
507                     default:
508                         pAssert(FALSE);
509                         break;
510                 }
511             }
512         }
513
514         return;
515
516     }
```

8.5.3.21 ObjectLoadEvict()

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

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

Error Returns	Meaning
TPM_RC_HANDLE	the persistent object does not exist or the associated hierarchy is disabled.
TPM_RC_OBJECT_MEMORY	no object slot

```
517     TPM_RC
518     ObjectLoadEvict(
519         TPM_HANDLE      *handle,        // IN:OUT: evict object handle.  If success, it
520                                         // will be replace by the loaded object handle
521         TPM_CC           commandCode    // IN: the command being processed
522         )
523     {
524         TPM_RC          result;
525         TPM_HANDLE      evictHandle = *handle;   // Save the evict handle
526         OBJECT          *object;
527
528         // If this is an index that references a persistent object created by
529         // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
530         if(*handle >=  PLATFORM_PERSISTENT)
531         {
532             // belongs to platform
533             if(g_phEnable == CLEAR)
534                 return TPM_RC_HANDLE;
535         }
536         // belongs to owner
537         else if(gc.shEnable == CLEAR)
538             return TPM_RC_HANDLE;
539
```

```
540         // Try to allocate a slot for an object
541         if(!ObjectAllocateSlot(handle, &object))
542             return TPM_RC_OBJECT_MEMORY;
543 
544         // Copy persistent object to transient object slot.  A TPM_RC_HANDLE
545         // may be returned at this point. This will mark the slot as containing
546         // a transient object so that it will be flushed at the end of the
547         // command
548         result = NvGetEvictObject(evictHandle, object);
549 
550         // Bail out if this failed
551         if(result != TPM_RC_SUCCESS)
552             return result;
553 
554         // check the object to see if it is in the endorsement hierarchy
555         // if it is and this is not a TPM2_EvictControl() command, indicate
556         // that the hierarchy is disabled.
557         // If the associated hierarchy is disabled, make it look like the
558         // handle is not defined
559         if(    ObjectDataGetHierarchy(object) == TPM_RH_ENDORSEMENT
560             && gc.ehEnable == CLEAR
561             && commandCode != TPM_CC_EvictControl
562            )
563             return TPM_RC_HANDLE;
564 
565         return result;
566     }
```

8.5.3.22 ObjectComputeName()

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

```
567     void
568     ObjectComputeName(
569         TPMT_PUBLIC     *publicArea,    // IN: public area of an object
570         TPM2B_NAME      *name           // OUT: name of the object
571         )
572     {
573         TPM2B_PUBLIC         marshalBuffer;
574         BYTE                *buffer;            // auxiliary marshal buffer pointer
575         HASH_STATE           hashState;         // hash state
576 
577         // if the nameAlg is NULL then there is no name.
578         if(publicArea->nameAlg == TPM_ALG_NULL)
579         {
580             name->t.size = 0;
581             return;
582         }
583         // Start hash stack
584         name->t.size = CryptStartHash(publicArea->nameAlg, &hashState);
585 
586         // Marshal the public area into its canonical form
587         buffer = marshalBuffer.b.buffer;
588 
589         marshalBuffer.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
590 
591         // Adding public area
592         CryptUpdateDigest2B(&hashState, &marshalBuffer.b);
593 
594         // Complete hash leaving room for the name algorithm
595         CryptCompleteHash(&hashState, name->t.size, &name->t.name[2]);
596 
597         // set the nameAlg
598         UINT16_TO_BYTE_ARRAY(publicArea->nameAlg, name->t.name);
```

```
599         name->t.size += 2;
600         return;
601     }
```

8.5.3.23 ObjectComputeQualifiedName()

This function computes the qualified name of an object.

```
602     void
603     ObjectComputeQualifiedName(
604         TPM2B_NAME      *parentQN,      // IN: parent's qualified name
605         TPM_ALG_ID       nameAlg,       // IN: name hash
606         TPM2B_NAME      *name,          // IN: name of the object
607         TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
608         )
609     {
610         HASH_STATE      hashState;   // hash state
611
612         //      QN_A = hash_A (QN of parent || NAME_A)
613
614         // Start hash
615         qualifiedName->t.size = CryptStartHash(nameAlg, &hashState);
616
617         // Add parent's qualified name
618         CryptUpdateDigest2B(&hashState, &parentQN->b);
619
620         // Add self name
621         CryptUpdateDigest2B(&hashState, &name->b);
622
623         // Complete hash leaving room for the name algorithm
624         CryptCompleteHash(&hashState, qualifiedName->t.size,
625                           &qualifiedName->t.name[2]);
626         UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
627         qualifiedName->t.size += 2;
628         return;
629     }
```

8.5.3.24 ObjectDataIsStorage()

This function determines if a public area has the attributes associated with a storage key. A storage key is an asymmetric object that has its *restricted* and *decrypt* attributes SET, and *sign* CLEAR.

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

```
630     BOOL
631     ObjectDataIsStorage(
632         TPMT_PUBLIC     *publicArea     // IN: public area of the object
633         )
634     {
635         if(   CryptIsAsymAlgorithm(publicArea->type)            // must be asymmetric,
636            && publicArea->objectAttributes.restricted == SET    // restricted,
637            && publicArea->objectAttributes.decrypt == SET       // decryption key
638            && publicArea->objectAttributes.sign == CLEAR        // can not be sign key
639           )
640             return TRUE;
641         else
642             return FALSE;
643     }
```

8.5.3.25 ObjectIsStorage()

This function determines if an object has the attributes associated with a storage key. A storage key is an asymmetric object that has its *restricted* and *decrypt* attributes SET, and *sign* CLEAR.

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

```
644 BOOL
645 ObjectIsStorage(
646     TPMI_DH_OBJECT   handle          // IN: object handle
647     )
648 {
649     OBJECT           *object = ObjectGet(handle);
650     return ObjectDataIsStorage(&object->publicArea);
651 }
```

8.5.3.26 ObjectCapGetLoaded()

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

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

```
652 TPMI_YES_NO
653 ObjectCapGetLoaded(
654     TPMI_DH_OBJECT   handle,         // IN: start handle
655     UINT32           count,          // IN: count of returned handles
656     TPML_HANDLE     *handleList      // OUT: list of handle
657     )
658 {
659     TPMI_YES_NO          more = NO;
660     UINT32               i;
661
662     pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
663
664     // Initialize output handle list
665     handleList->count = 0;
666
667     // The maximum count of handles we may return is MAX_CAP_HANDLES
668     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
669
670     // Iterate object slots to get loaded object handles
671     for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
672     {
673         if(s_objects[i].occupied == TRUE)
674         {
675             // A valid transient object can not be the copy of a persistent object
676             pAssert(s_objects[i].object.entity.attributes.evict == CLEAR);
677
678             if(handleList->count < count)
679             {
680                 // If we have not filled up the return list, add this object
681                 // handle to it
682                 handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
683                 handleList->count++;
```

```
684                 }
685                 else
686                 {
687                     // If the return list is full but we still have loaded object
688                     // available, report this and stop iterating
689                     more = YES;
690                     break;
691                 }
692             }
693         }
694
695         return more;
696     }
```

8.5.3.27 ObjectCapGetTransientAvail()

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

```
697     UINT32
698     ObjectCapGetTransientAvail(
699         void
700         )
701     {
702         UINT32      i;
703         UINT32      num = 0;
704
705         // Iterate object slot to get the number of unoccupied slots
706         for(i = 0; i < MAX_LOADED_OBJECTS; i++)
707         {
708             if(s_objects[i].occupied == FALSE) num++;
709         }
710
711         return num;
712     }
```

8.6 PCR.c

8.6.1 Introduction

This function contains the functions needed for PCR access and manipulation.

This implementation uses a static allocation for the PCR. The amount of memory is allocated based on the number of PCR in the implementation and the number of implemented hash algorithms. This is not the expected implementation. PCR SPACE DEFINITIONS.

In the definitions below, the *g_hashPcrMap* is a bit array that indicates which of the PCR are implemented. The *g_hashPcr* array is an array of digests. In this implementation, the space is allocated whether the PCR is implemented or not.

8.6.2 Includes, Defines, and Data Definitions

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

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

```
4   static const PCR_Attributes s_initAttributes[] =
```

```
{
    // 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.

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

```
BOOL
PCRBelongsAuthGroup(
    TPMI_DH_PCR      handle,         // IN: handle of PCR
    UINT32          *groupIndex      // OUT: group index if PCR belongs a
                                     //      group that allows authValue.  If PCR
                                     //      does not belong to an auth group,
                                     //      the value in this parameter is
                                     //      invalid
)
{
#if NUM_AUTHVALUE_PCR_GROUP > 0
    // Platform specification determines to which auth group a PCR belongs (if
    // any). In this implementation, we assume there is only
    // one auth group which contains PCR[20-22].  If the platform specification
    // requires differently, the implementation should be changed accordingly
    if(handle >= 20 && handle <= 22)
    {
        *groupIndex = 0;
        return TRUE;
    }

#endif
    return FALSE;
}
```

8.6.3.2 PCRBelongsPolicyGroup()

This function indicates if a PCR belongs to a group that requires a policy authorization in order to modify the PCR. If it does, *groupIndex* is set to value of the group index. This feature of PCR is decided by the platform specification.

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

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

8.6.3.3 PCRBelongsTCBGroup()

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

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

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

8.6.3.4 PCRPolicyIsAvailable()

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

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

```
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.

```
void
PCRGetAuthValue(
    TPMI_DH_PCR      handle,         // IN: PCR handle
    TPM2B_AUTH      *auth            // 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.

```
TPMI_ALG_HASH
PCRGetAuthPolicy(
    TPMI_DH_PCR      handle,         // IN: PCR handle
    TPM2B_DIGEST    *policy          // OUT: policy of PCR
    )
{
    UINT32           groupIndex;

    if(PCRBelongsPolicyGroup(handle, &groupIndex))
    {
        *policy = gp.pcrPolicies.policy[groupIndex];
        return gp.pcrPolicies.hashAlg[groupIndex];
    }
    else
    {
        policy->t.size = 0;
```

```
128             return TPM_ALG_NULL;
129         }
130     }
```

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.

```
131     void
132     PCRSimStart(
133         void
134         )
135     {
136         UINT32  i;
137         for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
138         {
139             gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
140             gp.pcrPolicies.policy[i].t.size = 0;
141         }
142 
143         for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
144         {
145             gc.pcrAuthValues.auth[i].t.size = 0;
146         }
147 
148         // We need to give an initial configuration on allocated PCR before
149         // receiving any TPM2_PCR_Allocate command to change this configuration
150         // When the simulation environment starts, we allocate all the PCRs
151         for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
152                 gp.pcrAllocated.count++)
153         {
154             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
155                 = CryptGetHashAlgByIndex(gp.pcrAllocated.count);
156 
157             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
158                 = PCR_SELECT_MAX;
159             for(i = 0; i < PCR_SELECT_MAX; i++)
160                 gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
161                     = 0xFF;
162         }
163 
164         // Store the initial configuration to NV
165         NvWriteReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
166         NvWriteReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
167 
168         return;
169     }
```

8.6.3.8 GetSavedPcrPointer()

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

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

```
170     static BYTE *
171     GetSavedPcrPointer (
172         TPM_ALG_ID       alg,           // IN: algorithm for bank
173         UINT32           pcrIndex       // IN: PCR index in PCR_SAVE
```

```
174         )
175     {
176         switch(alg)
177         {
178     #ifdef TPM_ALG_SHA1
179         case TPM_ALG_SHA1:
180             return gc.pcrSave.sha1[pcrIndex];
181             break;
182     #endif
183     #ifdef TPM_ALG_SHA256
184         case TPM_ALG_SHA256:
185             return gc.pcrSave.sha256[pcrIndex];
186             break;
187     #endif
188     #ifdef TPM_ALG_SHA384
189         case TPM_ALG_SHA384:
190             return gc.pcrSave.sha384[pcrIndex];
191             break;
192     #endif
193
194     #ifdef TPM_ALG_SHA512
195         case TPM_ALG_SHA512:
196             return gc.pcrSave.sha512[pcrIndex];
197             break;
198     #endif
199     #ifdef TPM_ALG_SM3_256
200         case TPM_ALG_SM3_256:
201             return gc.pcrSave.sm3_256[pcrIndex];
202             break;
203     #endif
204         default:
205             FAIL(FATAL_ERROR_INTERNAL);
206         }
207         //return NULL; // Can't be reached
208     }
```

8.6.3.9 PcrIsAllocated()

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

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

```
209     BOOL
210     PcrIsAllocated (
211         UINT32          pcr,            // IN: The number of the PCR
212         TPMI_ALG_HASH   hashAlg         // IN: The PCR algorithm
213         )
214     {
215         UINT32              i;
216         BOOL                allocated = FALSE;
217
218         if(pcr < IMPLEMENTATION_PCR)
219         {
220
221             for(i = 0; i < gp.pcrAllocated.count; i++)
222             {
223                 if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
224                 {
225                     if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr/8])
226                             & (1 << (pcr % 8))) != 0)
```

```
227                     allocated = TRUE;
228                 else
229                     allocated = FALSE;
230                 break;
231             }
232         }
233     }
234     return allocated;
235 }
```

8.6.3.10 GetPcrPointer()

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

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

```
236 static BYTE *
237 GetPcrPointer (
238     TPM_ALG_ID        alg,            // IN: algorithm for bank
239     UINT32            pcrNumber       // IN: PCR number
240     )
241 {
242     static BYTE     *pcr = NULL;
243 
244     if(!PcrIsAllocated(pcrNumber, alg))
245         return NULL;
246 
247     switch(alg)
248     {
249 #ifdef TPM_ALG_SHA1
250     case TPM_ALG_SHA1:
251         pcr = s_pcrs[pcrNumber].sha1Pcr;
252         break;
253 #endif
254 #ifdef TPM_ALG_SHA256
255     case TPM_ALG_SHA256:
256         pcr = s_pcrs[pcrNumber].sha256Pcr;
257         break;
258 #endif
259 #ifdef TPM_ALG_SHA384
260     case TPM_ALG_SHA384:
261         pcr = s_pcrs[pcrNumber].sha384Pcr;
262         break;
263 #endif
264 #ifdef TPM_ALG_SHA512
265     case TPM_ALG_SHA512:
266         pcr = s_pcrs[pcrNumber].sha512Pcr;
267         break;
268 #endif
269 #ifdef TPM_ALG_SM3_256
270     case TPM_ALG_SM3_256:
271         pcr = s_pcrs[pcrNumber].sm3_256Pcr;
272         break;
273 #endif
274     default:
275         pAssert(FALSE);
276         break;
277     }
278 
279     return pcr;
```

```
}
```

8.6.3.11 IsPcrSelected()

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

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

```
static BOOL
IsPcrSelected (
    UINT32               pcr,           // IN: The number of the PCR
    TPMS_PCR_SELECTION  *selection      // IN: The selection structure
    )
{
    BOOL                 selected = FALSE;
    if(   pcr < IMPLEMENTATION_PCR
       && ((selection->pcrSelect[pcr/8]) & (1 << (pcr % 8))) != 0)
        selected = TRUE;

    return selected;
}
```

8.6.3.12 FilterPcr()

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

```
static void
FilterPcr(
    TPMS_PCR_SELECTION  *selection      // IN: input PCR selection
    )
{
    UINT32     i;
    TPMS_PCR_SELECTION      *allocated = NULL;

    // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
    for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
        selection->pcrSelect[i] = 0;

    // Find the internal configuration for the bank
    for(i = 0; i < gp.pcrAllocated.count; i++)
    {
        if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
        {
            allocated = &gp.pcrAllocated.pcrSelections[i];
            break;
        }
    }

    for (i = 0; i < selection->sizeofSelect; i++)
    {
        if(allocated == NULL)
        {
            // If the required bank does not exist, clear input selection
            selection->pcrSelect[i] = 0;
        }
        else
            selection->pcrSelect[i] &= allocated->pcrSelect[i];
    }

```

```
327         return;
328     }
```

8.6.3.13 PcrDrtm()

This function does the DRTM and H-CRTM processing it is called from _TPM_Hash_End().

```
329     void
330     PcrDrtm(
331         const TPMI_DH_PCR        pcrHandle,     // IN: the index of the PCR to be
332                                                 //     modified
333         const TPMI_ALG_HASH      hash,          // IN: the bank identifier
334         const TPM2B_DIGEST      *digest         // IN: the digest to modify the PCR
335         )
336     {
337         BYTE        *pcrData = GetPcrPointer(hash, pcrHandle);
338
339         if(pcrData != NULL)
340         {
341             // Rest the PCR to zeros
342             MemorySet(pcrData, 0, digest->t.size);
343
344             // if the TPM has not started, then set the PCR to 0...04 and then extend
345             if(!TPMIsStarted())
346             {
347                 pcrData[digest->t.size - 1] = 4;
348             }
349             // Now, extend the value
350             PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
351         }
352     }
```

8.6.3.14 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```
353     void
354     PCRStartup(
355         STARTUP_TYPE     type,          // IN: startup type
356         BYTE             locality       // IN: startup locality
357         )
358     {
359         UINT32              pcr, j;
360         UINT32              saveIndex = 0;
361
362         g_pcrReConfig = FALSE;
363
364         if(type != SU_RESUME)
365         {
366             // PCR generation counter is cleared at TPM_RESET and TPM_RESTART
367             gr.pcrCounter = 0;
368         }
369
370         // Initialize/Restore PCR values
371         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
372         {
373             // On resume, need to know if this PCR had its state saved or not
374             UINT32      stateSaved =
375                 (type == SU_RESUME && s_initAttributes[pcr].stateSave == SET) ? 1 : 0;
376
377             // If this is the H-CRTM PCR and we are not doing a resume and we
378             // had an H-CRTM event, then we don't change this PCR
379             if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
```

```
380             continue;
381
382         // Iterate each hash algorithm bank
383         for(j = 0; j < gp.pcrAllocated.count; j++)
384         {
385             TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
386             BYTE            *pcrData = GetPcrPointer(hash, pcr);
387             UINT16           pcrSize = CryptGetHashDigestSize(hash);
388
389             if(pcrData != NULL)
390             {
391                 // if state was saved
392                 if(stateSaved == 1)
393                 {
394                     // Restore saved PCR value
395                     BYTE    *pcrSavedData;
396                     pcrSavedData = GetSavedPcrPointer(
397                                         gp.pcrAllocated.pcrSelections[j].hash,
398                                         saveIndex);
399                     MemoryCopy(pcrData, pcrSavedData, pcrSize, pcrSize);
400                 }
401                 else
402                     // PCR was not restored by state save
403                 {
404                     // If the reset locality of the PCR is 4, then
405                     // the reset value is all one's, otherwise it is
406                     // all zero.
407                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
408                         MemorySet(pcrData, 0xFF, pcrSize);
409                     else
410                     {
411                         MemorySet(pcrData, 0, pcrSize);
412                         if(pcr == HCRTM_PCR)
413                             pcrData[pcrSize-1] = locality;
414                     }
415                 }
416             }
417         }
418         saveIndex += stateSaved;
419     }
420
421     // Reset authValues
422     if(type != SU_RESUME)
423     {
424         for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
425         {
426             gc.pcrAuthValues.auth[j].t.size = 0;
427         }
428     }
429
430 }
```

8.6.3.15 PCRStateSave()

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

```
431 void
432 PCRStateSave(
433     TPM_SU            type            // IN: startup type
434     )
435 {
436     UINT32            pcr, j;
437     UINT32            saveIndex = 0;
438
```

```
439         // if state save CLEAR, nothing to be done.  Return here
440         if(type == TPM_SU_CLEAR) return;
441 
442         // Copy PCR values to the structure that should be saved to NV
443         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
444         {
445             UINT32  stateSaved = (s_initAttributes[pcr].stateSave == SET) ? 1 : 0;
446 
447             // Iterate each hash algorithm bank
448             for(j = 0; j < gp.pcrAllocated.count; j++)
449             {
450                 BYTE    *pcrData;
451                 UINT32  pcrSize;
452 
453                 pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
454 
455                 if(pcrData != NULL)
456                 {
457                     pcrSize
458                         = CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
459 
460                     if(stateSaved == 1)
461                     {
462                         // Restore saved PCR value
463                         BYTE    *pcrSavedData;
464                         pcrSavedData
465                             = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
466                                                  saveIndex);
467                         MemoryCopy(pcrSavedData, pcrData, pcrSize, pcrSize);
468                     }
469                 }
470             }
471             saveIndex += stateSaved;
472         }
473 
474         return;
475     }
```

8.6.3.16 PCRIsStateSaved()

This function indicates if the selected PCR is a PCR that is state saved on TPM2_Shutdown(STATE). The return value is based on PCR attributes.

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

```
476     BOOL
477     PCRIsStateSaved(
478         TPMI_DH_PCR      handle         // IN: PCR handle to be extended
479         )
480     {
481         UINT32              pcr = handle - PCR_FIRST;
482 
483         if(s_initAttributes[pcr].stateSave == SET)
484             return TRUE;
485         else
486             return FALSE;
487     }
```

8.6.3.17 PCRIsResetAllowed()

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

Return Value	Meaning
TRUE	TPM2_PCR_Reset() is allowed
FALSE	TPM2_PCR_Reset() is not allowed

```
488  BOOL
489  PCRIsResetAllowed(
490      TPMI_DH_PCR      handle          // IN: PCR handle to be extended
491      )
492  {
493      UINT8               commandLocality;
494      UINT8               localityBits = 1;
495      UINT32              pcr = handle - PCR_FIRST;
496
497      // Check for the locality
498      commandLocality = _plat__LocalityGet();
499
500  #ifdef DRTM_PCR
501      // For a TPM that does DRTM, Reset is not allowed at locality 4
502      if(commandLocality == 4)
503          return FALSE;
504  #endif
505
506      localityBits = localityBits << commandLocality;
507      if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
508          return FALSE;
509      else
510          return TRUE;
511
512  }
```

8.6.3.18 PCRChanged()

This function checks a PCR handle to see if the attributes for the PCR are set so that any change to the PCR causes an increment of the *pcrCounter*. If it does, then the function increments the counter.

```
513  void
514  PCRChanged(
515      TPM_HANDLE       pcrHandle       // IN: the handle of the PCR that changed.
516      )
517  {
518      // For the reference implementation, the only change that does not cause
519      // increment is a change to a PCR in the TCB group.
520      if(!PCRBelongsTCBGroup(pcrHandle))
521          gr.pcrCounter++;
522  }
```

8.6.3.19 PCRIsExtendAllowed()

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

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

```
523 BOOL
524 PCRIsExtendAllowed(
525     TPMI_DH_PCR      handle          // IN: PCR handle to be extended
526     )
527 {
528     UINT8               commandLocality;
529     UINT8               localityBits = 1;
530     UINT32              pcr = handle - PCR_FIRST;
531 
532     // Check for the locality
533     commandLocality = _plat__LocalityGet();
534     localityBits = localityBits << commandLocality;
535     if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
536         return FALSE;
537     else
538         return TRUE;
539 
540 }
```

8.6.3.20 PCRExtend()

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

```
541 void
542 PCRExtend(
543     TPMI_DH_PCR      handle,         // IN: PCR handle to be extended
544     TPMI_ALG_HASH    hash,           // IN: hash algorithm of PCR
545     UINT32           size,           // IN: size of data to be extended
546     BYTE            *data            // IN: data to be extended
547     )
548 {
549     UINT32               pcr = handle - PCR_FIRST;
550     BYTE                *pcrData;
551     HASH_STATE           hashState;
552     UINT16               pcrSize;
553 
554     pcrData = GetPcrPointer(hash, pcr);
555 
556     // Extend PCR if it is allocated
557     if(pcrData != NULL)
558     {
559         pcrSize = CryptGetHashDigestSize(hash);
560         CryptStartHash(hash, &hashState);
561         CryptUpdateDigest(&hashState, pcrSize, pcrData);
562         CryptUpdateDigest(&hashState, size, data);
563         CryptCompleteHash(&hashState, pcrSize, pcrData);
564 
565         // If PCR does not belong to TCB group, increment PCR counter
566         if(!PCRBelongsTCBGroup(handle))
567             gr.pcrCounter++;
568     }
569 
570     return;
571 }
```

8.6.3.21 PCRComputeCurrentDigest()

This function computes the digest of the selected PCR.

As a side-effect, *selection* is modified so that only the implemented PCR will have their bits still set.

```
572 void
573 PCRComputeCurrentDigest(
574     TPMI_ALG_HASH        hashAlg,       // IN: hash algorithm to compute digest
575     TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
576                                         //     output)
577     TPM2B_DIGEST        *digest         // OUT: digest
578     )
579 {
580     HASH_STATE               hashState;
581     TPMS_PCR_SELECTION      *select;
582     BYTE                    *pcrData;   // will point to a digest
583     UINT32                   pcrSize;
584     UINT32                   pcr;
585     UINT32                   i;
586
587     // Initialize the hash
588     digest->t.size = CryptStartHash(hashAlg, &hashState);
589     pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
590
591     // Iterate through the list of PCR selection structures
592     for(i = 0; i < selection->count; i++)
593     {
594         // Point to the current selection
595         select = &selection->pcrSelections[i]; // Point to the current selection
596         FilterPcr(select);      // Clear out the bits for unimplemented PCR
597
598         // Need the size of each digest
599         pcrSize = CryptGetHashDigestSize(selection->pcrSelections[i].hash);
600
601         // Iterate through the selection
602         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
603         {
604             if(IsPcrSelected(pcr, select))         // Is this PCR selected
605             {
606                 // Get pointer to the digest data for the bank
607                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
608                 pAssert(pcrData != NULL);
609                 CryptUpdateDigest(&hashState, pcrSize, pcrData);  // add to digest
610             }
611         }
612     }
613     // Complete hash stack
614     CryptCompleteHash2B(&hashState, &digest->b);
615
616     return;
617 }
```

8.6.3.22 PCRRead()

This function is used to read a list of selected PCR. If the requested PCR number exceeds the maximum number that can be output, the *selection* is adjusted to reflect the actual output PCR.

```
618 void
619 PCRRead(
620     TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
621                                         //     output)
622     TPML_DIGEST         *digest,        // OUT: digest
623     UINT32              *pcrCounter     // OUT: the current value of PCR generation
```

```
624                                         //     number
625     )
626 {
627     TPMS_PCR_SELECTION      *select;
628     BYTE                    *pcrData;       // will point to a digest
629     UINT32                   pcr;
630     UINT32                   i;
631 
632     digest->count = 0;
633 
634     // Iterate through the list of PCR selection structures
635     for(i = 0; i < selection->count; i++)
636     {
637         // Point to the current selection
638         select = &selection->pcrSelections[i]; // Point to the current selection
639         FilterPcr(select);      // Clear out the bits for unimplemented PCR
640 
641         // Iterate through the selection
642         for (pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
643         {
644             if(IsPcrSelected(pcr, select))         // Is this PCR selected
645             {
646                 // Check if number of digest exceed upper bound
647                 if(digest->count > 7)
648                 {
649                     // Clear rest of the current select bitmap
650                     while(   pcr < IMPLEMENTATION_PCR
651                              // do not round up!
652                           && (pcr / 8) < select->sizeofSelect)
653                     {
654                         // do not round up!
655                         select->pcrSelect[pcr/8] &= (BYTE) ~(1 << (pcr % 8));
656                         pcr++;
657                     }
658                     // Exit inner loop
659                     break;;
660                 }
661                 // Need the size of each digest
662                 digest->digests[digest->count].t.size =
663                     CryptGetHashDigestSize(selection->pcrSelections[i].hash);
664 
665                 // Get pointer to the digest data for the bank
666                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
667                 pAssert(pcrData != NULL);
668                 // Add to the data to digest
669                 MemoryCopy(digest->digests[digest->count].t.buffer,
670                            pcrData,
671                            digest->digests[digest->count].t.size,
672                            digest->digests[digest->count].t.size);
673                 digest->count++;
674             }
675         }
676         // If we exit inner loop because we have exceed the output upper bound
677         if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
678         {
679             // Clear rest of the selection
680             while(i < selection->count)
681             {
682                 MemorySet(selection->pcrSelections[i].pcrSelect, 0,
683                           selection->pcrSelections[i].sizeofSelect);
684                 i++;
685             }
686             // exit outer loop
687             break;
688         }
689     }
```

```
690
691         *pcrCounter = gr.pcrCounter;
692
693         return;
694     }
```

8.6.3.23 PcrWrite()

This function is used by _TPM_Hash_End() to set a PCR to the computed hash of the H-CRTM event.

```
695     void
696     PcrWrite(
697         TPMI_DH_PCR      handle,         // IN: PCR handle to be extended
698         TPMI_ALG_HASH    hash,           // IN: hash algorithm of PCR
699         TPM2B_DIGEST    *digest          // IN: the new value
700         )
701     {
702         UINT32               pcr = handle - PCR_FIRST;
703         BYTE                *pcrData;
704
705         // Copy value to the PCR if it is allocated
706         pcrData = GetPcrPointer(hash, pcr);
707         if(pcrData != NULL)
708         {
709             MemoryCopy(pcrData, digest->t.buffer, digest->t.size, digest->t.size); ;
710         }
711
712         return;
713     }
```

8.6.3.24 PCRAllocate()

This function is used to change the PCR allocation.

Error Returns	Meaning
TPM_RC_SUCCESS	allocate success
TPM_RC_NO_RESULTS	allocate failed
TPM_RC_PCR	improper allocation

```
714     TPM_RC
715     PCRAllocate(
716         TPML_PCR_SELECTION  *allocate,      // IN: required allocation
717         UINT32              *maxPCR,        // OUT: Maximum number of PCR
718         UINT32              *sizeNeeded,    // OUT: required space
719         UINT32              *sizeAvailable  // OUT: available space
720         )
721     {
722         UINT32                  i, j, k;
723         TPML_PCR_SELECTION      newAllocate;
724         // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
725         BOOL                    pcrHcrtm = FALSE;
726         BOOL                    pcrDrtm = FALSE;
727
728         // Create the expected new PCR allocation based on the existing allocation
729         // and the new input:
730         //  1. if a PCR bank does not appear in the new allocation, the existing
731         //     allocation of this PCR bank will be preserved.
732         //  2. if a PCR bank appears multiple times in the new allocation, only the
733         //     last one will be in effect.
734         newAllocate = gp.pcrAllocated;
```

```
735         for(i = 0; i < allocate->count; i++)
736         {
737             for(j = 0; j < newAllocate.count; j++)
738             {
739                 // If hash matches, the new allocation covers the old allocation
740                 // for this particular bank.
741                 // The assumption is the initial PCR allocation (from manufacture)
742                 // has all the supported hash algorithms with an assigned bank
743                 // (possibly empty).  So there must be a match for any new bank
744                 // allocation from the input.
745                 if(newAllocate.pcrSelections[j].hash ==
746                     allocate->pcrSelections[i].hash)
747                 {
748                     newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
749                     break;
750                 }
751             }
752             // The j loop must exit with a match.
753             pAssert(j < newAllocate.count);
754         }
755 
756         // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
757         *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
758         if(*maxPCR > IMPLEMENTATION_PCR)
759             *maxPCR = IMPLEMENTATION_PCR;
760 
761         // Compute required size for allocation
762         *sizeNeeded = 0;
763         for(i = 0; i < newAllocate.count; i++)
764         {
765             UINT32      digestSize
766                 = CryptGetHashDigestSize(newAllocate.pcrSelections[i].hash);
767 #if defined(DRTM_PCR)
768             // Make sure that we end up with at least one DRTM PCR
769 #   define PCR_DRTM  (PCR_FIRST + DRTM_PCR)     // for cosmetics
770             pcrDrtm =    pcrDrtm || TEST_BIT(PCR_DRTM, newAllocate.pcrSelections[i]);
771 #else   // if DRTM PCR is not required, indicate that the allocation is OK
772             pcrDrtm = TRUE;
773 #endif
774 
775 #if defined(HCRTM_PCR)
776             // and one HCRTM PCR (since this is usually PCR 0...)
777 #   define PCR_HCRTM (PCR_FIRST + HCRTM_PCR)
778             pcrHcrtm =    pcrDrtm || TEST_BIT(PCR_HCRTM, newAllocate.pcrSelections[i]);
779 #else
780             pcrHcrtm = TRUE;
781 #endif
782             for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
783             {
784                 BYTE        mask = 1;
785                 for(k = 0; k < 8; k++)
786                 {
787                     if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
788                         *sizeNeeded += digestSize;
789                     mask = mask << 1;
790                 }
791             }
792         }
793 
794         if(!pcrDrtm || !pcrHcrtm)
795             return TPM_RC_PCR;
796 
797         // In this particular implementation, we always have enough space to
798         // allocate PCR.  Different implementation may return a sizeAvailable less
799         // than the sizeNeed.
800         *sizeAvailable = sizeof(s_pcrs);
```

```
801
802         // Save the required allocation to NV.  Note that after NV is written, the
803         // PCR allocation in NV is no longer consistent with the RAM data
804         // gp.pcrAllocated.  The NV version reflect the allocate after next
805         // TPM_RESET, while the RAM version reflects the current allocation
806         NvWriteReserved(NV_PCR_ALLOCATED, &newAllocate);
807
808         return TPM_RC_SUCCESS;
809
810     }
```

8.6.3.25 PCRSetValue()

This function is used to set the designated PCR in all banks to an initial value. The initial value is signed and will be sign extended into the entire PCR.

```
811     void
812     PCRSetValue(
813         TPM_HANDLE        handle,         // IN: the handle of the PCR to set
814         INT8              initialValue    // IN: the value to set
815         )
816     {
817         int               i;
818         UINT32            pcr = handle - PCR_FIRST;
819         TPMI_ALG_HASH     hash;
820         UINT16            digestSize;
821         BYTE             *pcrData;
822
823         // Iterate supported PCR bank algorithms to reset
824         for(i = 0; i < HASH_COUNT; i++)
825         {
826             hash = CryptGetHashAlgByIndex(i);
827             // Prevent runaway
828             if(hash == TPM_ALG_NULL)
829                 break;
830
831             // Get a pointer to the data
832             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
833
834             // If the PCR is allocated
835             if(pcrData != NULL)
836             {
837                 // And the size of the digest
838                 digestSize = CryptGetHashDigestSize(hash);
839
840                 // Set the LSO to the input value
841                 pcrData[digestSize - 1] = initialValue;
842
843                 // Sign extend
844                 if(initialValue >= 0)
845                     MemorySet(pcrData, 0, digestSize - 1);
846                 else
847                     MemorySet(pcrData, -1, digestSize - 1);
848             }
849         }
850     }
```

8.6.3.26 PCRResetDynamics

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

```
851     void
852     PCRResetDynamics(
```

```
853         void
854         )
855     {
856         UINT32              pcr, i;
857 
858         // Initialize PCR values
859         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
860         {
861             // Iterate each hash algorithm bank
862             for(i = 0; i < gp.pcrAllocated.count; i++)
863             {
864                 BYTE    *pcrData;
865                 UINT32  pcrSize;
866 
867                 pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
868 
869                 if(pcrData != NULL)
870                 {
871                     pcrSize =
872                         CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
873 
874                     // Reset PCR
875                     // Any PCR can be reset by locality 4 should be reset to 0
876                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
877                         MemorySet(pcrData, 0, pcrSize);
878                 }
879             }
880         }
881         return;
882     }
```

8.6.3.27 PCRCapGetAllocation()

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

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

```
883     TPMI_YES_NO
884     PCRCapGetAllocation(
885         UINT32                count,          // IN: count of return
886         TPML_PCR_SELECTION  *pcrSelection   // OUT: PCR allocation list
887         )
888     {
889         if(count == 0)
890         {
891             pcrSelection->count = 0;
892             return YES;
893         }
894         else
895         {
896             *pcrSelection = gp.pcrAllocated;
897             return NO;
898         }
899     }
```

8.6.3.28 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```
900 static void
901 PCRSetSelectBit(
902     UINT32           pcr,           // IN: PCR number
903     BYTE            *bitmap         // OUT: bit map to be set
904     )
905 {
906     bitmap[pcr / 8] |= (1 << (pcr % 8));
907     return;
908 }
```

8.6.3.29 PCRGetProperty()

This function returns the selected PCR property.

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

```
909 static BOOL
910 PCRGetProperty(
911     TPM_PT_PCR               property,
912     TPMS_TAGGED_PCR_SELECT  *select
913     )
914 {
915     UINT32              pcr;
916     UINT32              groupIndex;
917
918     select->tag = property;
919     // Always set the bitmap to be the size of all PCR
920     select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
921
922     // Initialize bitmap
923     MemorySet(select->pcrSelect, 0, select->sizeofSelect);
924
925     // Collecting properties
926     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
927     {
928         switch(property)
929         {
930             case TPM_PT_PCR_SAVE:
931                 if(s_initAttributes[pcr].stateSave == SET)
932                     PCRSetSelectBit(pcr, select->pcrSelect);
933                 break;
934             case TPM_PT_PCR_EXTEND_L0:
935                 if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
936                     PCRSetSelectBit(pcr, select->pcrSelect);
937                 break;
938             case TPM_PT_PCR_RESET_L0:
939                 if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
940                     PCRSetSelectBit(pcr, select->pcrSelect);
941                 break;
942             case TPM_PT_PCR_EXTEND_L1:
943                 if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
944                     PCRSetSelectBit(pcr, select->pcrSelect);
945                 break;
946             case TPM_PT_PCR_RESET_L1:
947                 if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
948                     PCRSetSelectBit(pcr, select->pcrSelect);
949                 break;
950             case TPM_PT_PCR_EXTEND_L2:
951                 if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
952                     PCRSetSelectBit(pcr, select->pcrSelect);
```

```
953                 break;
954             case TPM_PT_PCR_RESET_L2:
955                 if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
956                     PCRSetSelectBit(pcr, select->pcrSelect);
957                 break;
958             case TPM_PT_PCR_EXTEND_L3:
959                 if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
960                     PCRSetSelectBit(pcr, select->pcrSelect);
961                 break;
962             case TPM_PT_PCR_RESET_L3:
963                 if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
964                     PCRSetSelectBit(pcr, select->pcrSelect);
965                 break;
966             case TPM_PT_PCR_EXTEND_L4:
967                 if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
968                     PCRSetSelectBit(pcr, select->pcrSelect);
969                 break;
970             case TPM_PT_PCR_RESET_L4:
971                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
972                     PCRSetSelectBit(pcr, select->pcrSelect);
973                 break;
974             case TPM_PT_PCR_DRTM_RESET:
975                 // DRTM reset PCRs are the PCR reset by locality 4
976                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
977                     PCRSetSelectBit(pcr, select->pcrSelect);
978                 break;
979 #if NUM_POLICY_PCR_GROUP > 0
980             case TPM_PT_PCR_POLICY:
981                 if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
982                     PCRSetSelectBit(pcr, select->pcrSelect);
983                 break;
984 #endif
985 #if NUM_AUTHVALUE_PCR_GROUP > 0
986             case TPM_PT_PCR_AUTH:
987                 if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
988                     PCRSetSelectBit(pcr, select->pcrSelect);
989                 break;
990 #endif
991 #if ENABLE_PCR_NO_INCREMENT == YES
992             case TPM_PT_PCR_NO_INCREMENT:
993                 if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
994                     PCRSetSelectBit(pcr, select->pcrSelect);
995                 break;
996 #endif
997             default:
998                 // If property is not supported, stop scanning PCR attributes
999                 // and return.
1000                return FALSE;
1001                break;
1002        }
1003    }
1004    return TRUE;
1005 }
```

8.6.3.30 PCRCapGetProperties()

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

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

```
1006  TPMI_YES_NO
1007  PCRCapGetProperties(
1008      TPM_PT_PCR                   property,      // IN: the starting PCR property
1009      UINT32                       count,         // IN: count of returned propertie
1010      TPML_TAGGED_PCR_PROPERTY    *select         // OUT: PCR select
1011      )
1012  {
1013      TPMI_YES_NO     more = NO;
1014      UINT32          i;
1015
1016      // Initialize output property list
1017      select->count = 0;
1018
1019      // The maximum count of properties we may return is MAX_PCR_PROPERTIES
1020      if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
1021
1022      // TPM_PT_PCR_FIRST is defined as 0 in spec.  It ensures that property
1023      // value would never be less than TPM_PT_PCR_FIRST
1024      pAssert(TPM_PT_PCR_FIRST == 0);
1025
1026      // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
1027      // implemented on the TPM.
1028      for(i = property; i <= TPM_PT_PCR_LAST; i++)
1029      {
1030          if(select->count < count)
1031          {
1032              // If we have not filled up the return list, add more properties to it
1033              if(PCRGetProperty(i, &select->pcrProperty[select->count]))
1034                  // only increment if the property is implemented
1035              select->count++;
1036          }
1037          else
1038          {
1039              // If the return list is full but we still have properties
1040              // available, report this and stop iterating.
1041              more = YES;
1042              break;
1043          }
1044      }
1045      return more;
1046  }
```

8.6.3.31 PCRCapGetHandles()

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

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

```
1047  TPMI_YES_NO
1048  PCRCapGetHandles(
1049      TPMI_DH_PCR      handle,         // IN: start handle
1050      UINT32           count,          // IN: count of returned handle
1051      TPML_HANDLE     *handleList      // OUT: list of handle
```

```
1052        )
1053    {
1054        TPMI_YES_NO     more = NO;
1055        UINT32          i;
1056
1057        pAssert(HandleGetType(handle) == TPM_HT_PCR);
1058
1059        // Initialize output handle list
1060        handleList->count = 0;
1061
1062        // The maximum count of handles we may return is MAX_CAP_HANDLES
1063        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1064
1065        // Iterate PCR handle range
1066        for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1067        {
1068            if(handleList->count < count)
1069            {
1070                // If we have not filled up the return list, add this PCR
1071                // handle to it
1072                handleList->handle[handleList->count] = i + PCR_FIRST;
1073                handleList->count++;
1074            }
1075            else
1076            {
1077                // If the return list is full but we still have PCR handle
1078                // available, report this and stop iterating
1079                more = YES;
1080                break;
1081            }
1082        }
1083        return more;
1084    }
```

8.7 PP.c

8.7.1 Introduction

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

8.7.2 Includes

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

8.7.3 Functions

8.7.3.1 PhysicalPresencePreInstall_Init()

This function is used to initialize the array of commands that require confirmation with physical presence. The array is an array of bits that has a correspondence with the command code.

This command should only ever be executable in a manufacturing setting or in a simulation.

```
2   void
3   PhysicalPresencePreInstall_Init(
4       void
5       )
6   {
7       // Clear all the PP commands
8       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.

```
void
PhysicalPresenceCommandSet(
    TPM_CC            commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // Assume command is implemented.  It should be checked before this
    // function is called
    pAssert(CommandIsImplemented(commandCode));

    // If the command is not a PP command, ignore it
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return;

    bitPos = commandCode - TPM_CC_PP_FIRST;

    // Set bit
    gp.ppList[bitPos/8] |= 1 << (bitPos % 8);

    return;
}
```

8.7.3.3 PhysicalPresenceCommandClear()

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

```
void
PhysicalPresenceCommandClear(
    TPM_CC            commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // Assume command is implemented.  It should be checked before this
    // function is called
    pAssert(CommandIsImplemented(commandCode));

    // If the command is not a PP command, ignore it
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return;

    // if the input code is TPM_CC_PP_Commands, it can not be cleared
    if(commandCode == TPM_CC_PP_Commands)
        return;

    bitPos = commandCode - TPM_CC_PP_FIRST;
```

```
62
63      // Set bit
64      gp.ppList[bitPos/8] |= (1 << (bitPos % 8));
65      // Flip it to off
66      gp.ppList[bitPos/8] ^= (1 << (bitPos % 8));
67
68      return;
69  }
```

8.7.3.4 PhysicalPresenceIsRequired()

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

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

```
70  BOOL
71  PhysicalPresenceIsRequired(
72      TPM_CC            commandCode     // IN: command code
73      )
74  {
75      UINT32      bitPos;
76
77      // if the input commandCode is not a PP command, return FALSE
78      if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
79          return FALSE;
80
81      bitPos = commandCode - TPM_CC_PP_FIRST;
82
83      // Check the bit map.  If the bit is SET, PP authorization is required
84      return ((gp.ppList[bitPos/8] & (1 << (bitPos % 8))) != 0);
85
86  }
```

8.7.3.5 PhysicalPresenceCapGetCCList()

This function returns a list of commands that require PP confirmation. The list starts from the first implemented command that has a command code that the same or greater than *commandCode*.

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

```
87  TPMI_YES_NO
88  PhysicalPresenceCapGetCCList(
89      TPM_CC            commandCode,    // IN: start command code
90      UINT32            count,          // IN: count of returned TPM_CC
91      TPML_CC          *commandList     // OUT: list of TPM_CC
92      )
93  {
94      TPMI_YES_NO     more = NO;
95      UINT32          i;
96
97      // Initialize output handle list
98      commandList->count = 0;
99
100     // The maximum count of command we may return is MAX_CAP_CC
101     if(count > MAX_CAP_CC) count = MAX_CAP_CC;
```



```
102
103         // Collect PP commands
104         for(i = commandCode; i <= TPM_CC_PP_LAST; i++)
105         {
106             if(PhysicalPresenceIsRequired(i))
107             {
108                 if(commandList->count < count)
109                 {
110                     // If we have not filled up the return list, add this command
111                     // code to it
112                     commandList->commandCodes[commandList->count] = i;
113                     commandList->count++;
114                 }
115                 else
116                 {
117                     // If the return list is full but we still have PP command
118                     // available, report this and stop iterating
119                     more = YES;
120                     break;
121                 }
122             }
123         }
124         return more;
125     }
```

8.8 Session.c

8.8.1 Introduction

The code in this file is used to manage the session context counter. The scheme implemented here is a "truncated counter". This scheme allows the TPM to not need TPM_SU_CLEAR for a very long period of time and still not have the context count for a session repeated.

The counter *(contextCounter*)in this implementation is a UINT64 but can be smaller. The "tracking array" *(contextArray*) only has 16-bits per context. The tracking array is the data that needs to be saved and restored across TPM_SU_STATE so that sessions are not lost when the system enters the sleep state. Also, when the TPM is active, the tracking array is kept in RAM making it important that the number of bytes for each entry be kept as small as possible.

The TPM prevents **collisions** of these truncated values by not allowing a *contextID* to be assigned if it would be the same as an existing value. Since the array holds 16 bits, after a context has been saved, an additional 2^16-1 contexts may be saved before the count would again match. The normal expectation is that the context will be flushed before its count value is needed again but it is always possible to have long-lived sessions.

The *contextID* is assigned when the context is saved (TPM2_ContextSave()). At that time, the TPM will compare the low-order 16 bits of *contextCounter* to the existing values in *contextArray* and if one matches, the TPM will return TPM_RC_CONTEXT_GAP (by construction, the entry that contains the matching value is the oldest context).

The expected remediation by the TRM is to load the oldest saved session context (the one found by the TPM), and save it. Since loading the oldest session also eliminates its *contextID* value from *contextArray*, there TPM will always be able to load and save the oldest existing context.

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

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

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

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

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

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

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

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

8.8.2 Includes, Defines, and Local Variables

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

8.8.3 File Scope Function -- ContextIdSetOldest()

This function is called when the oldest *contextID* is being loaded or deleted. Once a saved context becomes the oldest, it stays the oldest until it is deleted.

Finding the oldest is a bit tricky. It is not just the numeric comparison of values but is dependent on the value of *contextCounter*.

Assume we have a small *contextArray* with 8, 4-bit values with values 1 and 2 used to indicate the loaded context slot number. Also assume that the array contains hex values of (0 0 1 0 3 0 9 F) and that the *contextCounter* is an 8-bit counter with a value of 0x37. Since the low nibble is 7, that means that values above 7 are older than values below it and, in this example, 9 is the oldest value.

Note if we subtract the counter value, from each slot that contains a saved *contextID* we get (- - - - B - 2 - 8) and the oldest entry is now easy to find.

```
5   static void
6   ContextIdSetOldest(
7       void
8       )
9   {
10      CONTEXT_SLOT    lowBits;
11      CONTEXT_SLOT    entry;
12      CONTEXT_SLOT    smallest = ((CONTEXT_SLOT) ~0);
13      UINT32  i;
```

```
14
15      // Set oldestSaveContext to a value indicating none assigned
16      s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
17
18      lowBits = (CONTEXT_SLOT)gr.contextCounter;
19      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
20      {
21          entry = gr.contextArray[i];
22
23          // only look at entries that are saved contexts
24          if(entry > MAX_LOADED_SESSIONS)
25          {
26              // Use a less than or equal in case the oldest
27              // is brand new (= lowBits-1) and equal to our initial
28              // value for smallest.
29              if(((CONTEXT_SLOT) (entry - lowBits)) <= smallest)
30              {
31                  smallest = (entry - lowBits);
32                  s_oldestSavedSession = i;
33              }
34          }
35      }
36      // When we finish, either the s_oldestSavedSession still has its initial
37      // value, or it has the index of the oldest saved context.
38  }
```

8.8.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```
39  void
40  SessionStartup(
41      STARTUP_TYPE     type
42      )
43  {
44      UINT32               i;
45
46      // Initialize session slots.  At startup, all the in-memory session slots
47      // are cleared and marked as not occupied
48      for(i = 0; i < MAX_LOADED_SESSIONS; i++)
49          s_sessions[i].occupied = FALSE;   // session slot is not occupied
50
51      // The free session slots the number of maximum allowed loaded sessions
52      s_freeSessionSlots = MAX_LOADED_SESSIONS;
53
54      // Initialize context ID data.  On a ST_SAVE or hibernate sequence, it will
55      // scan the saved array of session context counts, and clear any entry that
56      // references a session that was in memory during the state save since that
57      // memory was not preserved over the ST_SAVE.
58      if(type == SU_RESUME || type == SU_RESTART)
59      {
60          // On ST_SAVE we preserve the contexts that were saved but not the ones
61          // in memory
62          for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
63          {
64              // If the array value is unused or references a loaded session then
65              // that loaded session context is lost and the array entry is
66              // reclaimed.
67              if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
68                  gr.contextArray[i] = 0;
69          }
70          // Find the oldest session in context ID data and set it in
71          // s_oldestSavedSession
72          ContextIdSetOldest();
```

```
    }
    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.

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

```
BOOL
SessionIsLoaded(
    TPM_HANDLE       handle          // IN: session handle
    )
{
    pAssert(   HandleGetType(handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION);

    handle = handle & HR_HANDLE_MASK;

    // if out of range of possible active session, or not assigned to a loaded
    // session return false
    if(   handle >= MAX_ACTIVE_SESSIONS
       || gr.contextArray[handle] == 0
       || gr.contextArray[handle] > MAX_LOADED_SESSIONS
      )
        return FALSE;

    return TRUE;
}
```

8.8.5.2 SessionIsSaved()

This function test a session handle references a saved session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE: An password authorization does not have a session.

This function requires that the handle be a valid session handle.

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

```
BOOL
SessionIsSaved(
    TPM_HANDLE       handle          // IN: session handle
    )
{
    pAssert(   HandleGetType(handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION);

    handle = handle & HR_HANDLE_MASK;
    // if out of range of possible active session, or not assigned, or
    // assigned to a loaded session, return false
    if(   handle >= MAX_ACTIVE_SESSIONS
       || gr.contextArray[handle] == 0
       || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
      )
        return FALSE;

    return TRUE;
}
```

8.8.5.3 SessionPCRValueIsCurrent()

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

This function requires the session is loaded.

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

```
BOOL
SessionPCRValueIsCurrent(
    TPMI_SH_POLICY   handle          // IN: session handle
    )
{
    SESSION             *session;

    pAssert(SessionIsLoaded(handle));

    session = SessionGet(handle);
    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.

```
144  SESSION *
145  SessionGet(
146      TPM_HANDLE       handle          // IN: session handle
147      )
148  {
149      CONTEXT_SLOT    sessionIndex;
150
151      pAssert(   HandleGetType(handle) == TPM_HT_POLICY_SESSION
152              || HandleGetType(handle) == TPM_HT_HMAC_SESSION
153             );
154
155      pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);
156
157      // get the contents of the session array.  Because session is loaded, we
158      // should always get a valid sessionIndex
159      sessionIndex = gr.contextArray[handle & HR_HANDLE_MASK] - 1;
160
161      pAssert(sessionIndex < MAX_LOADED_SESSIONS);
162
163      return &s_sessions[sessionIndex].session;
164  }
```

8.8.6 Utility Functions

8.8.6.1 ContextIdSessionCreate()

This function is called when a session is created. It will check to see if the current gap would prevent a context from being saved. If so it will return TPM_RC_CONTEXT_GAP. Otherwise, it will try to find an open slot in *contextArray*, set *contextArray* to the slot.

This routine requires that the caller has determined the session array index for the session.

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

```
165  static TPM_RC
166  ContextIdSessionCreate (
167      TPM_HANDLE      *handle,        // OUT: receives the assigned handle. This will
168                                      //     be an index that must be adjusted by the
169                                      //     caller according to the type of the
170                                      //     session created
171      UINT32           sessionIndex   // IN: The session context array entry that will
172                                      //     be occupied by the created session
173      )
174  {
175
176      pAssert(sessionIndex < MAX_LOADED_SESSIONS);
177
178      // check to see if creating the context is safe
179      // Is this going to be an assignment for the last session context
180      // array entry?  If so, then there will be no room to recycle the
181      // oldest context if needed.  If the gap is not at maximum, then
182      // it will be possible to save a context if it becomes necessary.
183      if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
184         && s_freeSessionSlots == 1)
185      {
186          // See if the gap is at maximum
```

```
187             if(   (CONTEXT_SLOT)gr.contextCounter
188                == gr.contextArray[s_oldestSavedSession])
189
190                 // Note: if this is being used on a TPM.combined, this return
191                 //       code should be transformed to an appropriate 1.2 error
192                 //       code for this case.
193                 return TPM_RC_CONTEXT_GAP;
194         }
195
196         // Find an unoccupied entry in the contextArray
197         for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
198         {
199             if(gr.contextArray[*handle] == 0)
200             {
201                 // indicate that the session associated with this handle
202                 // references a loaded session
203                 gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex+1);
204                 return TPM_RC_SUCCESS;
205             }
206         }
207         return TPM_RC_SESSION_HANDLES;
208     }
```

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

```
209     TPM_RC
210     SessionCreate(
211         TPM_SE            sessionType,    // IN: the session type
212         TPMI_ALG_HASH     authHash,       // IN: the hash algorithm
213         TPM2B_NONCE      *nonceCaller,    // IN: initial nonceCaller
214         TPMT_SYM_DEF     *symmetric,      // IN: the symmetric algorithm
215         TPMI_DH_ENTITY    bind,           // IN: the bind object
216         TPM2B_DATA       *seed,           // IN: seed data
217         TPM_HANDLE       *sessionHandle   // OUT: the session handle
218         )
219     {
220         TPM_RC               result = TPM_RC_SUCCESS;
221         CONTEXT_SLOT         slotIndex;
222         SESSION             *session = NULL;
223
224         pAssert(   sessionType == TPM_SE_HMAC
225                 || sessionType == TPM_SE_POLICY
226                 || sessionType == TPM_SE_TRIAL);
227
228         // If there are no open spots in the session array, then no point in searching
229         if(s_freeSessionSlots == 0)
230             return TPM_RC_SESSION_MEMORY;
231
232         // Find a space for loading a session
233         for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
234         {
```

```
235             // Is this available?
236             if(s_sessions[slotIndex].occupied == FALSE)
237             {
238                 session = &s_sessions[slotIndex].session;
239                 break;
240             }
241         }
242         // if no spot found, then this is an internal error
243         pAssert (slotIndex < MAX_LOADED_SESSIONS);
244 
245         // Call context ID function to get a handle.  TPM_RC_SESSION_HANDLE may be
246         // returned from ContextIdHandelAssign()
247         result = ContextIdSessionCreate(sessionHandle, slotIndex);
248         if(result != TPM_RC_SUCCESS)
249             return result;
250 
251         //*** Only return from this point on is TPM_RC_SUCCESS
252 
253         // Can now indicate that the session array entry is occupied.
254         s_freeSessionSlots--;
255         s_sessions[slotIndex].occupied = TRUE;
256 
257         // Initialize the session data
258         MemorySet(session, 0, sizeof(SESSION));
259 
260         // Initialize internal session data
261         session->authHashAlg = authHash;
262         // Initialize session type
263         if(sessionType == TPM_SE_HMAC)
264         {
265             *sessionHandle += HMAC_SESSION_FIRST;
266 
267         }
268         else
269         {
270             *sessionHandle += POLICY_SESSION_FIRST;
271 
272             // For TPM_SE_POLICY or TPM_SE_TRIAL
273             session->attributes.isPolicy = SET;
274             if(sessionType == TPM_SE_TRIAL)
275                 session->attributes.isTrialPolicy = SET;
276 
277             // Initialize policy session data
278             SessionInitPolicyData(session);
279         }
280         // Create initial session nonce
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 crated by this command
300             TPM2B_AUTH  entityAuth;             // The authValue of the entity
```

```
                                                // associated with HMAC session

        // Get hash size, which is also the length of sessionKey
        hashSize = CryptGetHashDigestSize(session->authHashAlg);

        // Get authValue of associated entity
        entityAuth.t.size = EntityGetAuthValue(bind, &entityAuth.t.buffer);

        // Concatenate authValue and seed
        pAssert(entityAuth.t.size + seed->t.size <= sizeof(key.t.buffer));
        MemoryCopy2B(&key.b, &entityAuth.b, sizeof(key.t.buffer));
        MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));

        session->sessionKey.t.size = hashSize;

        // Compute the session key
        KDFa(session->authHashAlg, &key.b, "ATH", &session->nonceTPM.b,
             &nonceCaller->b, hashSize * 8, session->sessionKey.t.buffer, NULL);
    }

    // Copy the name of the entity that the HMAC session is bound to
    // Policy session is not bound to an entity
    if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
    {
        session->attributes.isBound = SET;
        SessionComputeBoundEntity(bind, &session->u1.boundEntity);
    }
    // If there is a bind object and it is subject to DA, then use of this session
    // is subject to DA regardless of how it is used.
    session->attributes.isDaBound =     (bind != TPM_RH_NULL)
                                    && (IsDAExempted(bind) == FALSE);

    // If the session is bound, then check to see if it is bound to lockoutAuth
    session->attributes.isLockoutBound =     (session->attributes.isDaBound  == SET)
                                         && (bind == TPM_RH_LOCKOUT);
    return TPM_RC_SUCCESS;

}
```

8.8.6.3 SessionContextSave()

This function is called when a session context is to be saved. The *contextID* of the saved session is returned. If no *contextID* can be assigned, then the routine returns TPM_RC_CONTEXT_GAP. If the function completes normally, the session slot will be freed.

This function requires that *handle* references a loaded session. Otherwise, it should not be called at the first place.

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	a *contextID* could not be assigned.
TPM_RC_TOO_MANY_CONTEXTS	the counter maxed out

```
TPM_RC
SessionContextSave (
    TPM_HANDLE           handle,        // IN: session handle
    CONTEXT_COUNTER     *contextID      // OUT: assigned contextID
    )
{
    UINT32                      contextIndex;
    CONTEXT_SLOT                slotIndex;

    pAssert(SessionIsLoaded(handle));
```

```
349
350         // check to see if the gap is already maxed out
351         // Need to have a saved session
352         if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
353               // if the oldest saved session has the same value as the low bits
354               // of the contextCounter, then the GAP is maxed out.
355            && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
356             return TPM_RC_CONTEXT_GAP;
357
358         // if the caller wants the context counter, set it
359         if(contextID != NULL)
360             *contextID = gr.contextCounter;
361
362         pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);
363
364         contextIndex = handle & HR_HANDLE_MASK;
365
366         // Extract the session slot number referenced by the contextArray
367         // because we are going to overwrite this with the low order
368         // contextID value.
369         slotIndex = gr.contextArray[contextIndex] - 1;
370
371         // Set the contextID for the contextArray
372         gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
373
374         // Increment the counter
375         gr.contextCounter++;
376
377         // In the unlikely event that the 64-bit context counter rolls over...
378         if(gr.contextCounter == 0)
379         {
380             // back it up
381             gr.contextCounter--;
382             // return an error
383             return TPM_RC_TOO_MANY_CONTEXTS;
384         }
385         // if the low-order bits wrapped, need to advance the value to skip over
386         // the values used to indicate that a session is loaded
387         if(((CONTEXT_SLOT)gr.contextCounter) == 0)
388             gr.contextCounter += MAX_LOADED_SESSIONS + 1;
389
390         // If no other sessions are saved, this is now the oldest.
391         if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
392             s_oldestSavedSession = contextIndex;
393
394         // Mark the session slot as unoccupied
395         s_sessions[slotIndex].occupied = FALSE;
396
397         // and indicate that there is an additional open slot
398         s_freeSessionSlots++;
399
400         return TPM_RC_SUCCESS;
401     }
```

8.8.6.4 SessionContextLoad()

This function is used to load a session from saved context. The session handle must be for a saved context.

If the gap is at a maximum, then the only session that can be loaded is the oldest session, otherwise TPM_RC_CONTEXT_GAP is returned.

This function requires that *handle* references a valid saved session.

Error Returns	Meaning
TPM_RC_SESSION_MEMORY	no free session slots
TPM_RC_CONTEXT_GAP	the gap count is maximum and this is not the oldest saved context

```
402 TPM_RC
403 SessionContextLoad(
404     SESSION         *session,       // IN: session structure from saved context
405     TPM_HANDLE      *handle         // IN/OUT: session handle
406     )
407 {
408     UINT32              contextIndex;
409     CONTEXT_SLOT        slotIndex;
410 
411     pAssert(   HandleGetType(*handle) == TPM_HT_POLICY_SESSION
412             || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
413 
414     // Don't bother looking if no openings
415     if(s_freeSessionSlots == 0)
416         return TPM_RC_SESSION_MEMORY;
417 
418     // Find a free session slot to load the session
419     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
420         if(s_sessions[slotIndex].occupied == FALSE) break;
421 
422     // if no spot found, then this is an internal error
423     pAssert (slotIndex < MAX_LOADED_SESSIONS);
424 
425     contextIndex = *handle & HR_HANDLE_MASK;   // extract the index
426 
427     // If there is only one slot left, and the gap is at maximum, the only session
428     // context that we can safely load is the oldest one.
429     if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
430        && s_freeSessionSlots == 1
431        && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
432        && contextIndex != s_oldestSavedSession
433       )
434         return TPM_RC_CONTEXT_GAP;
435 
436     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
437 
438     // set the contextArray value to point to the session slot where
439     // the context is loaded
440     gr.contextArray[contextIndex] = slotIndex + 1;
441 
442     // if this was the oldest context, find the new oldest
443     if(contextIndex == s_oldestSavedSession)
444         ContextIdSetOldest();
445 
446     // Copy session data to session slot
447     s_sessions[slotIndex].session = *session;
448 
449     // Set session slot as occupied
450     s_sessions[slotIndex].occupied = TRUE;
451 
452     // Reduce the number of open spots
453     s_freeSessionSlots--;
454 
455     return TPM_RC_SUCCESS;
456 }
```

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.

```
457  void
458  SessionFlush(
459      TPM_HANDLE       handle          // IN: loaded or saved session handle
460      )
461  {
462      CONTEXT_SLOT         slotIndex;
463      UINT32               contextIndex;   // Index into contextArray
464
465      pAssert(    (   HandleGetType(handle) == TPM_HT_POLICY_SESSION
466                   || HandleGetType(handle) == TPM_HT_HMAC_SESSION
467                  )
468              && (SessionIsLoaded(handle)  || SessionIsSaved(handle))
469             );
470
471      // Flush context ID of this session
472      // Convert handle to an index into the contextArray
473      contextIndex = handle & HR_HANDLE_MASK;
474
475      pAssert(contextIndex < sizeof(gr.contextArray)/sizeof(gr.contextArray[0]));
476
477      // Get the current contents of the array
478      slotIndex = gr.contextArray[contextIndex];
479
480      // Mark context array entry as available
481      gr.contextArray[contextIndex] = 0;
482
483      // Is this a saved session being flushed
484      if(slotIndex > MAX_LOADED_SESSIONS)
485      {
486          // Flushing the oldest session?
487          if(contextIndex == s_oldestSavedSession)
488              // If so, find a new value for oldest.
489              ContextIdSetOldest();
490      }
491      else
492      {
493          // Adjust slot index to point to session array index
494          slotIndex -= 1;
495
496          // Free session array index
497          s_sessions[slotIndex].occupied = FALSE;
498          s_freeSessionSlots++;
499      }
500
501      return;
502  }
```

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.

```
503  void
504  SessionComputeBoundEntity(
```

```
505         TPMI_DH_ENTITY   entityHandle,  // IN: handle of entity
506         TPM2B_NAME      *bind           // OUT: binding value
507         )
508     {
509         TPM2B_AUTH          auth;
510         INT16               overlap;
511
512         // Get name
513         bind->t.size = EntityGetName(entityHandle, &bind->t.name);
514
515     //    // The bound value of a reserved handle is the handle itself
516     //    if(bind->t.size == sizeof(TPM_HANDLE)) return;
517
518         // For all the other entities, concatenate the auth value to the name.
519         // Get a local copy of the auth value because some overlapping
520         // may be necessary.
521         auth.t.size = EntityGetAuthValue(entityHandle, &auth.t.buffer);
522         pAssert(auth.t.size <= sizeof(TPMU_HA));
523
524         // Figure out if there will be any overlap
525         overlap = bind->t.size + auth.t.size - sizeof(bind->t.name);
526
527         // There is overlap if the combined sizes are greater than will fit
528         if(overlap > 0)
529         {
530             // The overlap area is at the end of the Name
531             BYTE    *result = &bind->t.name[bind->t.size - overlap];
532             int     i;
533
534             // XOR the auth value into the Name for the overlap area
535             for(i = 0; i < overlap; i++)
536                 result[i] ^= auth.t.buffer[i];
537         }
538         else
539         {
540             // There is no overlap
541             overlap = 0;
542         }
543         //copy the remainder of the authData to the end of the name
544         MemoryCopy(&bind->t.name[bind->t.size], &auth.t.buffer[overlap],
545                    auth.t.size - overlap, sizeof(bind->t.name) - bind->t.size);
546
547         // Increase the size of the bind data by the size of the auth - the overlap
548         bind->t.size += auth.t.size-overlap;
549
550         return;
551     }
```

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.

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

```
563         session->u2.policyDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
564         return;
565     }
```

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.

```
566     void
567     SessionResetPolicyData(
568         SESSION         *session        // IN: the session to reset
569         )
570     {
571         session->commandCode = 0;       // No command
572
573         // No locality selected
574         MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
575
576         // The cpHash size to zero
577         session->u1.cpHash.b.size = 0;
578
579         // No timeout
580         session->timeOut = 0;
581
582         // Reset the pcrCounter
583         session->pcrCounter = 0;
584
585         // Reset the policy hash
586         MemorySet(&session->u2.policyDigest.t.buffer, 0,
587                   session->u2.policyDigest.t.size);
588
589         // Reset the session attributes
590         MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
591
592         // set the policy attribute
593         session->attributes.isPolicy = SET;
594     }
```

8.8.6.9 SessionCapGetLoaded()

This function returns a list of handles of loaded session, started from input *handle*

Handle must be in valid loaded session handle range, but does not have to point to a loaded session.

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

```
595     TPMI_YES_NO
596     SessionCapGetLoaded(
597         TPMI_SH_POLICY   handle,        // IN: start handle
598         UINT32           count,         // IN: count of returned handle
599         TPML_HANDLE     *handleList     // OUT: list of handle
600         )
601     {
602         TPMI_YES_NO     more = NO;
603         UINT32          i;
604
605         pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
606
607         // Initialize output handle list
```

```
608         handleList->count = 0;
609
610         // The maximum count of handles we may return is MAX_CAP_HANDLES
611         if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
612
613         // Iterate session context ID slots to get loaded session handles
614         for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
615         {
616             // If session is active
617             if(gr.contextArray[i] != 0)
618             {
619                 // If session is loaded
620                 if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
621                 {
622                     if(handleList->count < count)
623                     {
624                         SESSION         *session;
625
626                         // If we have not filled up the return list, add this
627                         // session handle to it
628                         // assume that this is going to be an HMAC session
629                         handle = i + HMAC_SESSION_FIRST;
630                         session = SessionGet(handle);
631                         if(session->attributes.isPolicy)
632                             handle = i + POLICY_SESSION_FIRST;
633                         handleList->handle[handleList->count] = handle;
634                         handleList->count++;
635                     }
636                     else
637                     {
638                         // If the return list is full but we still have loaded object
639                         // available, report this and stop iterating
640                         more = YES;
641                         break;
642                     }
643                 }
644             }
645         }
646
647         return more;
648
649     }
```

8.8.6.10 SessionCapGetSaved()

This function returns a list of handles for saved session, starting at *handle*.

Handle must be in a valid handle range, but does not have to point to a saved session

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

```
650     TPMI_YES_NO
651     SessionCapGetSaved(
652         TPMI_SH_HMAC     handle,         // IN: start handle
653         UINT32           count,          // IN: count of returned handle
654         TPML_HANDLE      *handleList     // OUT: list of handle
655         )
656     {
657         TPMI_YES_NO      more = NO;
658         UINT32           i;
659
```

```
660         pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);
661
662         // Initialize output handle list
663         handleList->count = 0;
664
665         // The maximum count of handles we may return is MAX_CAP_HANDLES
666         if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
667
668         // Iterate session context ID slots to get loaded session handles
669         for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
670         {
671             // If session is active
672             if(gr.contextArray[i] != 0)
673             {
674                 // If session is saved
675                 if (gr.contextArray[i] > MAX_LOADED_SESSIONS)
676                 {
677                     if(handleList->count < count)
678                     {
679                         // If we have not filled up the return list, add this
680                         // session handle to it
681                         handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
682                         handleList->count++;
683                     }
684                     else
685                     {
686                         // If the return list is full but we still have loaded object
687                         // available, report this and stop iterating
688                         more = YES;
689                         break;
690                     }
691                 }
692             }
693         }
694
695         return more;
696
697     }
```

8.8.6.11 SessionCapGetLoadedNumber()

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

```
698     UINT32
699     SessionCapGetLoadedNumber(
700         void
701         )
702     {
703         return MAX_LOADED_SESSIONS - s_freeSessionSlots;
704     }
```

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.

```
705     UINT32
706     SessionCapGetLoadedAvail(
707         void
708         )
```

```
709 {
710     return s_freeSessionSlots;
711 }
```

8.8.6.13 SessionCapGetActiveNumber()

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

```
712 UINT32
713 SessionCapGetActiveNumber(
714     void
715     )
716 {
717     UINT32              i;
718     UINT32              num = 0;
719 
720     // Iterate the context array to find the number of non-zero slots
721     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
722     {
723         if(gr.contextArray[i] != 0) num++;
724     }
725 
726     return num;
727 }
```

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.

```
728 UINT32
729 SessionCapGetActiveAvail(
730     void
731     )
732 {
733     UINT32              i;
734     UINT32              num = 0;
735 
736     // Iterate the context array to find the number of zero slots
737     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
738     {
739         if(gr.contextArray[i] == 0) num++;
740     }
741 
742     return num;
743 }
```

8.9 Time.c

8.9.1 Introduction

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

8.9.2 Includes

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

8.9.3 Functions

8.9.3.1 TimePowerOn()

This function initialize time info at _TPM_Init().

```
3   void
4   TimePowerOn(
5       void
6       )
7   {
8       TPM_SU          orderlyShutDown;
9
10      // Read orderly data info from NV memory
11      NvReadReserved(NV_ORDERLY_DATA, &go);
12
13      // Read orderly shut down state flag
14      NvReadReserved(NV_ORDERLY, &orderlyShutDown);
15
16      // If the previous cycle is orderly shut down, the value of the safe bit
17      // the same as previously saved.  Otherwise, it is not safe.
18      if(orderlyShutDown == SHUTDOWN_NONE)
19          go.clockSafe= NO;
20      else
21          go.clockSafe = YES;
22
23      // Set the initial state of the DRBG
24      CryptDrbgGetPutState(PUT_STATE);
25
26      // Clear time since TPM power on
27      g_time = 0;
28
29      return;
30  }
```

8.9.3.2 TimeStartup()

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

```
31  void
32  TimeStartup(
33      STARTUP_TYPE     type           // IN: start up type
34      )
35  {
36      if(type == SU_RESUME)
37      {
38          // Resume sequence
39          gr.restartCount++;
40      }
41      else
42      {
43          if(type == SU_RESTART)
44          {
45              // Hibernate sequence
46              gr.clearCount++;
47              gr.restartCount++;
48          }
49          else
50          {
51              // Reset sequence
52              // Increase resetCount
53              gp.resetCount++;
```

```
54
55              // Write resetCount to NV
56              NvWriteReserved(NV_RESET_COUNT, &gp.resetCount);
57              gp.totalResetCount++;
58
59              // We do not expect the total reset counter overflow during the life
60              // time of TPM.  if it ever happens, TPM will be put to failure mode
61              // and there is no way to recover it.
62              // The reason that there is no recovery is that we don't increment
63              // the NV totalResetCount when incrementing would make it 0. When the
64              // TPM starts up again, the old value of totalResetCount will be read
65              // and we will get right back to here with the increment failing.
66              if(gp.totalResetCount == 0)
67                  FAIL(FATAL_ERROR_INTERNAL);
68
69              // Write total reset counter to NV
70              NvWriteReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
71
72              // Reset restartCount
73              gr.restartCount = 0;
74          }
75      }
76
77      return;
78  }
```

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.

```
79  void
80  TimeUpdateToCurrent(
81      void
82      )
83  {
84      UINT64          oldClock;
85      UINT64          elapsed;
86  #define CLOCK_UPDATE_MASK  ((1ULL << NV_CLOCK_UPDATE_INTERVAL)- 1)
87
88      // Can't update time during the dark interval or when rate limiting.
89      if(NvIsAvailable() != TPM_RC_SUCCESS)
90          return;
91
92      // Save the old clock value
93      oldClock = go.clock;
94
95      // Update the time info to current
96      elapsed = _plat__ClockTimeElapsed();
97      go.clock += elapsed;
98      g_time += elapsed;
99
100     // Check to see if the update has caused a need for an nvClock update
101     // CLOCK_UPDATE_MASK is measured by second, while the value in go.clock is
102     // recorded by millisecond.  Align the clock value to second before the bit
```

```
103         // operations
104         if( ((go.clock/1000) | CLOCK_UPDATE_MASK)
105                 > ((oldClock/1000) | CLOCK_UPDATE_MASK))
106         {
107             // Going to update the time state so the safe flag
108             // should be set
109             go.clockSafe = YES;
110 
111             // Get the DRBG state before updating orderly data
112             CryptDrbgGetPutState(GET_STATE);
113 
114             NvWriteReserved(NV_ORDERLY_DATA, &go);
115         }
116 
117         // Call self healing logic for dictionary attack parameters
118         DASelfHeal();
119 
120         return;
121     }
```

8.9.3.4 TimeSetAdjustRate()

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

```
122     void
123     TimeSetAdjustRate(
124         TPM_CLOCK_ADJUST     adjust          // IN: adjust constant
125         )
126     {
127         switch(adjust)
128         {
129             case TPM_CLOCK_COARSE_SLOWER:
130                 _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
131                 break;
132             case TPM_CLOCK_COARSE_FASTER:
133                 _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
134                 break;
135             case TPM_CLOCK_MEDIUM_SLOWER:
136                 _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
137                 break;
138             case TPM_CLOCK_MEDIUM_FASTER:
139                 _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
140                 break;
141             case TPM_CLOCK_FINE_SLOWER:
142                 _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
143                 break;
144             case TPM_CLOCK_FINE_FASTER:
145                 _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
146                 break;
147             case TPM_CLOCK_NO_CHANGE:
148                 break;
149             default:
150                 pAssert(FALSE);
151                 break;
152         }
153 
154         return;
155     }
```

8.9.3.5 TimeGetRange()

This function is used to access TPMS_TIME_INFO. The TPMS_TIME_INFO structure is treaded as an array of bytes, and a byte offset and length determine what bytes are returned.

Error Returns	Meaning
TPM_RC_RANGE	invalid data range

```
156   TPM_RC
157   TimeGetRange(
158       UINT16           offset,        // IN: offset in TPMS_TIME_INFO
159       UINT16           size,          // IN: size of data
160       TIME_INFO       *dataBuffer     // OUT: result buffer
161       )
162   {
163       TPMS_TIME_INFO      timeInfo;
164       UINT16              infoSize;
165       BYTE                infoData[sizeof(TPMS_TIME_INFO)];
166       BYTE                *buffer;
167
168       // Fill TPMS_TIME_INFO structure
169       timeInfo.time = g_time;
170       TimeFillInfo(&timeInfo.clockInfo);
171
172       // Marshal TPMS_TIME_INFO to canonical form
173       buffer = infoData;
174       infoSize = TPMS_TIME_INFO_Marshal(&timeInfo, &buffer, NULL);
175
176       // Check if the input range is valid
177       if(offset + size > infoSize) return TPM_RC_RANGE;
178
179       // Copy info data to output buffer
180       MemoryCopy(dataBuffer, infoData + offset, size, sizeof(TIME_INFO));
181
182       return TPM_RC_SUCCESS;
183   }
```

8.9.3.6 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```
184   void
185   TimeFillInfo(
186       TPMS_CLOCK_INFO     *clockInfo
187       )
188   {
189       clockInfo->clock = go.clock;
190       clockInfo->resetCount = gp.resetCount;
191       clockInfo->restartCount = gr.restartCount;
192
193       // If NV is not available, clock stopped advancing and the value reported is
194       // not "safe".
195       if(NvIsAvailable() == TPM_RC_SUCCESS)
196           clockInfo->safe = go.clockSafe;
197       else
198           clockInfo->safe = NO;
199
200       return;
201   }
```

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

```
#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, 1, 0}},
#endif

    {TPM_ALG_KEYEDHASH,     {0, 0, 1, 1, 0, 1, 1, 0, 0}},

#ifdef TPM_ALG_XOR
    {TPM_ALG_XOR,           {0, 1, 1, 0, 0, 0, 0, 0, 0}},
#endif

#ifdef TPM_ALG_SHA256
    {TPM_ALG_SHA256,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA384
    {TPM_ALG_SHA384,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA512
    {TPM_ALG_SHA512,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_WHIRLPOOL512
    {TPM_ALG_WHIRLPOOL512,  {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SM3_256
    {TPM_ALG_SM3_256,       {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
```

```
52  #ifdef TPM_ALG_SM4
53      {TPM_ALG_SM4,          {0, 1, 0, 0, 0, 0, 0, 0, 0}},
54  #endif
55  #ifdef TPM_ALG_RSASSA
56      {TPM_ALG_RSASSA,        {1, 0, 0, 0, 0, 1, 0, 0, 0}},
57  #endif
58  #ifdef TPM_ALG_RSAES
59      {TPM_ALG_RSAES,         {1, 0, 0, 0, 0, 0, 1, 0, 0}},
60  #endif
61  #ifdef TPM_ALG_RSAPSS
62      {TPM_ALG_RSAPSS,        {1, 0, 0, 0, 0, 1, 0, 0, 0}},
63  #endif
64  #ifdef TPM_ALG_OAEP
65      {TPM_ALG_OAEP,          {1, 0, 0, 0, 0, 0, 1, 0, 0}},
66  #endif
67  #ifdef TPM_ALG_ECDSA
68      {TPM_ALG_ECDSA,         {1, 0, 0, 0, 0, 1, 0, 1, 0}},
69  #endif
70  #ifdef TPM_ALG_ECDH
71      {TPM_ALG_ECDH,          {1, 0, 0, 0, 0, 0, 0, 1, 0}},
72  #endif
73  #ifdef TPM_ALG_ECDAA
74      {TPM_ALG_ECDAA,         {1, 0, 0, 0, 0, 1, 0, 0, 0}},
75  #endif
76  #ifdef TPM_ALG_ECSCHNORR
77      {TPM_ALG_ECSCHNORR,     {1, 0, 0, 0, 0, 1, 0, 0, 0}},
78  #endif
79  #ifdef TPM_ALG_KDF1_SP800_56a
80      {TPM_ALG_KDF1_SP800_56a,{0, 0, 1, 0, 0, 0, 0, 1, 0}},
81  #endif
82  #ifdef TPM_ALG_KDF2
83      {TPM_ALG_KDF2,          {0, 0, 1, 0, 0, 0, 0, 1, 0}},
84  #endif
85  #ifdef TPM_ALG_KDF1_SP800_108
86      {TPM_ALG_KDF1_SP800_108,{0, 0, 1, 0, 0, 0, 0, 1, 0}},
87  #endif
88  #ifdef TPM_ALG_ECC
89      {TPM_ALG_ECC,           {1, 0, 0, 1, 0, 0, 0, 0, 0}},
90  #endif
91
92      {TPM_ALG_SYMCIPHER,     {0, 0, 0, 1, 0, 0, 0, 0, 0}},
93
94  #ifdef TPM_ALG_CTR
95      {TPM_ALG_CTR,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
96  #endif
97  #ifdef TPM_ALG_OFB
98      {TPM_ALG_OFB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
99  #endif
100 #ifdef TPM_ALG_CBC
101     {TPM_ALG_CBC,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
102 #endif
103 #ifdef TPM_ALG_CFB
104     {TPM_ALG_CFB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
105 #endif
106 #ifdef TPM_ALG_ECB
107     {TPM_ALG_ECB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
108 #endif
109 };
```

9.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2_GetCapability() to return a list of the implemented algorithms.

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

```
110   TPMI_YES_NO
111   AlgorithmCapGetImplemented(
112       TPM_ALG_ID                   algID,     // IN: the starting algorithm ID
113       UINT32                       count,     // IN: count of returned algorithms
114       TPML_ALG_PROPERTY           *algList    // OUT: algorithm list
115   )
116   {
117       TPMI_YES_NO     more = NO;
118       UINT32          i;
119       UINT32          algNum;
120
121       // initialize output algorithm list
122       algList->count = 0;
123
124       // The maximum count of algorithms we may return is MAX_CAP_ALGS.
125       if(count > MAX_CAP_ALGS)
126           count = MAX_CAP_ALGS;
127
128       // Compute how many algorithms are defined in s_algorithms array.
129       algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
130
131       // Scan the implemented algorithm list to see if there is a match to 'algID'.
132       for(i = 0; i < algNum; i++)
133       {
134           // If algID is less than the starting algorithm ID, skip it
135           if(s_algorithms[i].algID < algID)
136               continue;
137           if(algList->count < count)
138           {
139               // If we have not filled up the return list, add more algorithms
140               // to it
141               algList->algProperties[algList->count].alg = s_algorithms[i].algID;
142               algList->algProperties[algList->count].algProperties =
143                   s_algorithms[i].attributes;
144               algList->count++;
145           }
146           else
147           {
148               // If the return list is full but we still have algorithms
149               // available, report this and stop scanning.
150               more = YES;
151               break;
152           }
153
154       }
155
156       return more;
157
158   }
159   LIB_EXPORT
160   void
161   AlgorithmGetImplementedVector(
162       ALGORITHM_VECTOR    *implemented    // OUT: the implemented bits are SET
163       )
164   {
165       int                      index;
166
167       // Nothing implemented until we say it is
168       MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));
```

```
169
170         for(index =  (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
171             index >= 0;
172             index--)
173                 SET_BIT(s_algorithms[index].algID, *implemented);
174         return;
175     }
```

9.2 Bits.c

9.2.1 Introduction

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

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

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

9.2.2 Includes

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

9.2.3 Functions

9.2.3.1 BitIsSet()

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

Return Value	Meaning
TRUE	bit is set
FALSE	bit is not set

```
2   BOOL
3   BitIsSet(
4       unsigned int     bitNum,        // IN: number of the bit in 'bArray'
5       BYTE            *bArray,        // IN: array containing the bit
6       unsigned int     arraySize      // IN: size in bytes of 'bArray'
7       )
8   {
9       pAssert(arraySize > (bitNum >> 3));
10      return((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
11  }
```

9.2.3.2 BitSet()

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

```
12  void
13  BitSet(
14      unsigned int     bitNum,        // IN: number of the bit in 'bArray'
15      BYTE            *bArray,        // IN: array containing the bit
16      unsigned int     arraySize      // IN: size in bytes of 'bArray'
17      )
18  {
19      pAssert(arraySize > bitNum/8);
20      bArray[bitNum >> 3] |= (1 << (bitNum & 7));
```

```
}
```

9.2.3.3 BitClear()

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

```
void
BitClear(
    unsigned int     bitNum,        // IN: number of the bit in 'bArray'.
    BYTE            *bArray,        // IN: array containing the bit
    unsigned int     arraySize      // IN: size in bytes of 'bArray'
    )
{
    pAssert(arraySize > bitNum/8);
    bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
}
```

9.3 CommandAttributeData.c

This is the command code attribute array for GetCapability(). Both this array and *s_commandAttributes* provides command code attributes, but tuned for different purpose

```
static const TPMA_CC    s_ccAttr [] = {
        {0x011f, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_UndefineSpaceSpecial
        {0x0120, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_EvictControl
        {0x0121, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_HierarchyControl
        {0x0122, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_UndefineSpace
        {0x0123, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
        {0x0124, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_ChangeEPS
        {0x0125, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_ChangePPS
        {0x0126, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_Clear
        {0x0127, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClearControl
        {0x0128, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClockSet
        {0x0129, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_HierarchyChangeAuth
        {0x012a, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_DefineSpace
        {0x012b, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Allocate
        {0x012c, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_SetAuthPolicy
        {0x012d, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PP_Commands
        {0x012e, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetPrimaryPolicy
        {0x012f, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_FieldUpgradeStart
        {0x0130, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClockRateAdjust
        {0x0131, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_CreatePrimary
        {0x0132, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_GlobalWriteLock
        {0x0133, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_GetCommandAuditDigest
        {0x0134, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Increment
        {0x0135, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_SetBits
        {0x0136, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Extend
        {0x0137, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Write
        {0x0138, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_WriteLock
        {0x0139, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_DictionaryAttackLockReset
        {0x013a, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_DictionaryAttackParameters
        {0x013b, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_ChangeAuth
        {0x013c, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Event
        {0x013d, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Reset
        {0x013e, 0, 0, 0, 1, 1, 0, 0, 0},     // TPM_CC_SequenceComplete
        {0x013f, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetAlgorithmSet
        {0x0140, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetCommandCodeAuditStatus
        {0x0141, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_FieldUpgradeData
        {0x0142, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_IncrementalSelfTest
        {0x0143, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_SelfTest
        {0x0144, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_Startup
        {0x0145, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_Shutdown
        {0x0146, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_StirRandom
```

```
42          {0x0147, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_ActivateCredential
43          {0x0148, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Certify
44          {0x0149, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_PolicyNV
45          {0x014a, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_CertifyCreation
46          {0x014b, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Duplicate
47          {0x014c, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_GetTime
48          {0x014d, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_GetSessionAuditDigest
49          {0x014e, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Read
50          {0x014f, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_ReadLock
51          {0x0150, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_ObjectChangeAuth
52          {0x0151, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_PolicySecret
53          {0x0152, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Rewrap
54          {0x0153, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Create
55          {0x0154, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ECDH_ZGen
56          {0x0155, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_HMAC
57          {0x0156, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Import
58          {0x0157, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_Load
59          {0x0158, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Quote
60          {0x0159, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_RSA_Decrypt
61          {0x015a, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
62          {0x015b, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_HMAC_Start
63          {0x015c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_SequenceUpdate
64          {0x015d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Sign
65          {0x015e, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Unseal
66          {0x015f, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
67          {0x0160, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_PolicySigned
68          {0x0161, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_ContextLoad
69          {0x0162, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ContextSave
70          {0x0163, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ECDH_KeyGen
71          {0x0164, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_EncryptDecrypt
72          {0x0165, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_FlushContext
73          {0x0166, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
74          {0x0167, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_LoadExternal
75          {0x0168, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_MakeCredential
76          {0x0169, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_ReadPublic
77          {0x016a, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyAuthorize
78          {0x016b, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyAuthValue
79          {0x016c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCommandCode
80          {0x016d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCounterTimer
81          {0x016e, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCpHash
82          {0x016f, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyLocality
83          {0x0170, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyNameHash
84          {0x0171, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyOR
85          {0x0172, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyTicket
86          {0x0173, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ReadPublic
87          {0x0174, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_RSA_Encrypt
88          {0x0175, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
89          {0x0176, 0, 0, 0, 0, 2, 1, 0, 0},     // TPM_CC_StartAuthSession
90          {0x0177, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_VerifySignature
91          {0x0178, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_ECC_Parameters
92          {0x0179, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_FirmwareRead
93          {0x017a, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetCapability
94          {0x017b, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetRandom
95          {0x017c, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetTestResult
96          {0x017d, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_Hash
97          {0x017e, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_PCR_Read
98          {0x017f, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPCR
99          {0x0180, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyRestart
100         {0x0181, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_ReadClock
101         {0x0182, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Extend
102         {0x0183, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_SetAuthValue
103         {0x0184, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_NV_Certify
104         {0x0185, 0, 1, 0, 1, 2, 0, 0, 0},     // TPM_CC_EventSequenceComplete
105         {0x0186, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_HashSequenceStart
106         {0x0187, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPhysicalPresence
107         {0x0188, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyDuplicationSelect
```

```
108             {0x0189, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyGetDigest
109             {0x018a, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_TestParms
110             {0x018b, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Commit
111             {0x018c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPassword
112             {0x018d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ZGen_2Phase
113             {0x018e, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_EC_Ephemeral
114             {0x018f, 0, 0, 0, 0, 1, 0, 0, 0}      // TPM_CC_PolicyNvWritten
115     };
116     typedef  UINT16              _ATTR_;
117     #define  NOT_IMPLEMENTED     (_ATTR_)(0)
118     #define  ENCRYPT_2           (_ATTR_)(1 <<  0)
119     #define  ENCRYPT_4           (_ATTR_)(1 <<  1)
120     #define  DECRYPT_2           (_ATTR_)(1 <<  2)
121     #define  DECRYPT_4           (_ATTR_)(1 <<  3)
122     #define  HANDLE_1_USER       (_ATTR_)(1 <<  4)
123     #define  HANDLE_1_ADMIN      (_ATTR_)(1 <<  5)
124     #define  HANDLE_1_DUP        (_ATTR_)(1 <<  6)
125     #define  HANDLE_2_USER       (_ATTR_)(1 <<  7)
126     #define  PP_COMMAND          (_ATTR_)(1 <<  8)
127     #define  IS_IMPLEMENTED      (_ATTR_)(1 <<  9)
128     #define  NO_SESSIONS         (_ATTR_)(1 << 10)
129     #define  NV_COMMAND          (_ATTR_)(1 << 11)
130     #define  PP_REQUIRED         (_ATTR_)(1 << 12)
131     #define  R_HANDLE            (_ATTR_)(1 << 13)
```

This is the command code attribute structure.

```
132     typedef UINT16 COMMAND_ATTRIBUTES;
133     static const COMMAND_ATTRIBUTES    s_commandAttributes [] = {
134         (_ATTR_)(CC_NV_UndefineSpaceSpecial    *
        (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),              // 0x011f
135         (_ATTR_)(CC_EvictControl               *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0120
136         (_ATTR_)(CC_HierarchyControl           *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0121
137         (_ATTR_)(CC_NV_UndefineSpace           *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0122
138         (_ATTR_)                                  (NOT_IMPLEMENTED),
        // 0x0123 - Not assigned
139         (_ATTR_)(CC_ChangeEPS                  *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0124
140         (_ATTR_)(CC_ChangePPS                  *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0125
141         (_ATTR_)(CC_Clear                      *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0126
142         (_ATTR_)(CC_ClearControl               *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0127
143         (_ATTR_)(CC_ClockSet                   *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0128
144         (_ATTR_)(CC_HierarchyChangeAuth        *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                   // 0x0129
145         (_ATTR_)(CC_NV_DefineSpace             *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                   // 0x012a
146         (_ATTR_)(CC_PCR_Allocate               *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x012b
147         (_ATTR_)(CC_PCR_SetAuthPolicy          *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                   // 0x012c
148         (_ATTR_)(CC_PP_Commands                *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),                            // 0x012d
149         (_ATTR_)(CC_SetPrimaryPolicy           *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                   // 0x012e
150         (_ATTR_)(CC_FieldUpgradeStart          *
        (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),                  // 0x012f
151         (_ATTR_)(CC_ClockRateAdjust            *
        (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0130
```

```
152     (_ATTR_)(CC_CreatePrimary              *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)), // 0x0131
153     (_ATTR_)(CC_NV_GlobalWriteLock         *
    (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0132
154     (_ATTR_)(CC_GetCommandAuditDigest      *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),      // 0x0133
155     (_ATTR_)(CC_NV_Increment               * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x0134
156     (_ATTR_)(CC_NV_SetBits                 * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x0135
157     (_ATTR_)(CC_NV_Extend                  *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x0136
158     (_ATTR_)(CC_NV_Write                   *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x0137
159     (_ATTR_)(CC_NV_WriteLock               * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x0138
160     (_ATTR_)(CC_DictionaryAttackLockReset  * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x0139
161     (_ATTR_)(CC_DictionaryAttackParameters * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x013a
162     (_ATTR_)(CC_NV_ChangeAuth              *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),                             // 0x013b
163     (_ATTR_)(CC_PCR_Event                  *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x013c
164     (_ATTR_)(CC_PCR_Reset                  * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x013d
165     (_ATTR_)(CC_SequenceComplete           *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x013e
166     (_ATTR_)(CC_SetAlgorithmSet            * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x013f
167     (_ATTR_)(CC_SetCommandCodeAuditStatus  *
    (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                             // 0x0140
168     (_ATTR_)(CC_FieldUpgradeData           * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x0141
169     (_ATTR_)(CC_IncrementalSelfTest        * (IS_IMPLEMENTED)),
    // 0x0142
170     (_ATTR_)(CC_SelfTest                   * (IS_IMPLEMENTED)),
    // 0x0143
171     (_ATTR_)(CC_Startup                    * (IS_IMPLEMENTED+NO_SESSIONS)),
    // 0x0144
172     (_ATTR_)(CC_Shutdown                   * (IS_IMPLEMENTED)),
    // 0x0145
173     (_ATTR_)(CC_StirRandom                 * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x0146
174     (_ATTR_)(CC_ActivateCredential         *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),     // 0x0147
175     (_ATTR_)(CC_Certify                    *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),     // 0x0148
176     (_ATTR_)(CC_PolicyNV                   *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x0149
177     (_ATTR_)(CC_CertifyCreation            *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x014a
178     (_ATTR_)(CC_Duplicate                  *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),                     // 0x014b
179     (_ATTR_)(CC_GetTime                    *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),      // 0x014c
180     (_ATTR_)(CC_GetSessionAuditDigest      *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),      // 0x014d
181     (_ATTR_)(CC_NV_Read                    *
    (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                              // 0x014e
182     (_ATTR_)(CC_NV_ReadLock                * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x014f
183     (_ATTR_)(CC_ObjectChangeAuth           *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),                   // 0x0150
184     (_ATTR_)(CC_PolicySecret               *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0151
```

```
185         (_ATTR_)(CC_Rewrap                      *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0152
186         (_ATTR_)(CC_Create                      *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0153
187         (_ATTR_)(CC_ECDH_ZGen                   *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0154
188         (_ATTR_)(CC_HMAC                        *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0155
189         (_ATTR_)(CC_Import                      *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0156
190         (_ATTR_)(CC_Load                        *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),           // 0x0157
191         (_ATTR_)(CC_Quote                       *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0158
192         (_ATTR_)(CC_RSA_Decrypt                 *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                    // 0x0159
193         (_ATTR_)                                  (NOT_IMPLEMENTED),
    // 0x015a - Not assigned
194         (_ATTR_)(CC_HMAC_Start                  *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),                     // 0x015b
195         (_ATTR_)(CC_SequenceUpdate              *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x015c
196         (_ATTR_)(CC_Sign                        *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                              // 0x015d
197         (_ATTR_)(CC_Unseal                      *
    (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                              // 0x015e
198         (_ATTR_)                                  (NOT_IMPLEMENTED),
    // 0x015f - Not assigned
199         (_ATTR_)(CC_PolicySigned                * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
    // 0x0160
200         (_ATTR_)(CC_ContextLoad                 * (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
    // 0x0161
201         (_ATTR_)(CC_ContextSave                 * (IS_IMPLEMENTED+NO_SESSIONS)),
    // 0x0162
202         (_ATTR_)(CC_ECDH_KeyGen                 * (IS_IMPLEMENTED+ENCRYPT_2)),
    // 0x0163
203         (_ATTR_)(CC_EncryptDecrypt              *
    (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                              // 0x0164
204         (_ATTR_)(CC_FlushContext                * (IS_IMPLEMENTED+NO_SESSIONS)),
    // 0x0165
205         (_ATTR_)                                  (NOT_IMPLEMENTED),
    // 0x0166 - Not assigned
206         (_ATTR_)(CC_LoadExternal                *
    (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),                         // 0x0167
207         (_ATTR_)(CC_MakeCredential              * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
    // 0x0168
208         (_ATTR_)(CC_NV_ReadPublic               * (IS_IMPLEMENTED+ENCRYPT_2)),
    // 0x0169
209         (_ATTR_)(CC_PolicyAuthorize             * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x016a
210         (_ATTR_)(CC_PolicyAuthValue             * (IS_IMPLEMENTED)),
    // 0x016b
211         (_ATTR_)(CC_PolicyCommandCode           * (IS_IMPLEMENTED)),
    // 0x016c
212         (_ATTR_)(CC_PolicyCounterTimer          * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x016d
213         (_ATTR_)(CC_PolicyCpHash                * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x016e
214         (_ATTR_)(CC_PolicyLocality              * (IS_IMPLEMENTED)),
    // 0x016f
215         (_ATTR_)(CC_PolicyNameHash              * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x0170
216         (_ATTR_)(CC_PolicyOR                    * (IS_IMPLEMENTED)),
    // 0x0171
217         (_ATTR_)(CC_PolicyTicket                * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x0172
```

```
218         (_ATTR_)(CC_ReadPublic                * (IS_IMPLEMENTED+ENCRYPT_2)),
    // 0x0173
219         (_ATTR_)(CC_RSA_Encrypt               * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
    // 0x0174
220         (_ATTR_)                                (NOT_IMPLEMENTED),
    // 0x0175 - Not assigned
221         (_ATTR_)(CC_StartAuthSession          *
    (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),                          // 0x0176
222         (_ATTR_)(CC_VerifySignature           * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x0177
223         (_ATTR_)(CC_ECC_Parameters            * (IS_IMPLEMENTED)),
    // 0x0178
224         (_ATTR_)(CC_FirmwareRead              * (IS_IMPLEMENTED+ENCRYPT_2)),
    // 0x0179
225         (_ATTR_)(CC_GetCapability             * (IS_IMPLEMENTED)),
    // 0x017a
226         (_ATTR_)(CC_GetRandom                 * (IS_IMPLEMENTED+ENCRYPT_2)),
    // 0x017b
227         (_ATTR_)(CC_GetTestResult             * (IS_IMPLEMENTED+ENCRYPT_2)),
    // 0x017c
228         (_ATTR_)(CC_Hash                      * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
    // 0x017d
229         (_ATTR_)(CC_PCR_Read                  * (IS_IMPLEMENTED)),
    // 0x017e
230         (_ATTR_)(CC_PolicyPCR                 * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x017f
231         (_ATTR_)(CC_PolicyRestart             * (IS_IMPLEMENTED)),
    // 0x0180
232         (_ATTR_)(CC_ReadClock                 * (IS_IMPLEMENTED+NO_SESSIONS)),
    // 0x0181
233         (_ATTR_)(CC_PCR_Extend                * (IS_IMPLEMENTED+HANDLE_1_USER)),
    // 0x0182
234         (_ATTR_)(CC_PCR_SetAuthValue          *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                               // 0x0183
235         (_ATTR_)(CC_NV_Certify                *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),       // 0x0184
236         (_ATTR_)(CC_EventSequenceComplete     *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),                 // 0x0185
237         (_ATTR_)(CC_HashSequenceStart         * (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
    // 0x0186
238         (_ATTR_)(CC_PolicyPhysicalPresence    * (IS_IMPLEMENTED)),
    // 0x0187
239         (_ATTR_)(CC_PolicyDuplicationSelect   * (IS_IMPLEMENTED+DECRYPT_2)),
    // 0x0188
240         (_ATTR_)(CC_PolicyGetDigest           * (IS_IMPLEMENTED+ENCRYPT_2)),
    // 0x0189
241         (_ATTR_)(CC_TestParms                 * (IS_IMPLEMENTED)),
    // 0x018a
242         (_ATTR_)(CC_Commit                    *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                     // 0x018b
243         (_ATTR_)(CC_PolicyPassword            * (IS_IMPLEMENTED)),
    // 0x018c
244         (_ATTR_)(CC_ZGen_2Phase               *
    (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                     // 0x018d
245         (_ATTR_)(CC_EC_Ephemeral              * (IS_IMPLEMENTED+ENCRYPT_2)),
    // 0x018e
246         (_ATTR_)(CC_PolicyNvWritten           * (IS_IMPLEMENTED))
    // 0x018f
247     };
```

9.4 CommandCodeAttributes.c

9.4.1 Introduction

This file contains the functions for testing various command properties.

9.4.2 Includes and Defines

```
#include     "Tpm.h"
#include     "InternalRoutines.h"
typedef UINT16          ATTRIBUTE_TYPE;
```

The following file is produced from the command tables in part 3 of the specification. It defines the attributes for each of the commands.

NOTE: This file is currently produced by an automated process. Files produced from Part 2 or Part 3 tables through automated processes are not included in the specification so that their is no ambiguity about the table containing the information being the normative definition.

```
#include     "CommandAttributeData.c"
```

9.4.3 Command Attribute Functions

9.4.3.1 CommandAuthRole()

This function returns the authorization role required of a handle.

Return Value	Meaning
AUTH_NONE	no authorization is required
AUTH_USER	user role authorization is required
AUTH_ADMIN	admin role authorization is required
AUTH_DUP	duplication role authorization is required

```
AUTH_ROLE
CommandAuthRole(
    TPM_CC      commandCode,        // IN: command code
    UINT32      handleIndex         // IN: handle index (zero based)
    )
{
    if(handleIndex > 1)
        return AUTH_NONE;
    if(handleIndex == 0) {
        ATTRIBUTE_TYPE  properties = s_commandAttributes[commandCode - TPM_CC_FIRST];
        if(properties & HANDLE_1_USER) return AUTH_USER;
        if(properties & HANDLE_1_ADMIN) return AUTH_ADMIN;
        if(properties & HANDLE_1_DUP) return AUTH_DUP;
        return AUTH_NONE;
    }
    if(s_commandAttributes[commandCode - TPM_CC_FIRST] & HANDLE_2_USER) return
AUTH_USER;
    return AUTH_NONE;
}
```

9.4.3.2 CommandIsImplemented()

This function indicates if a command is implemented.

Return Value	Meaning
TRUE	if the command is implemented
FALSE	if the command is not implemented

```
BOOL
CommandIsImplemented(
    TPM_CC            commandCode    // IN: command code
    )
{
    if(commandCode < TPM_CC_FIRST || commandCode > TPM_CC_LAST)
        return FALSE;
    if((s_commandAttributes[commandCode - TPM_CC_FIRST] & IS_IMPLEMENTED))
        return TRUE;
    else
        return FALSE;
}
```

9.4.3.3 CommandGetAttribute()

return a TPMA_CC structure for the given command code

```
TPMA_CC
CommandGetAttribute(
    TPM_CC            commandCode    // IN: command code
    )
{
    UINT32      size = sizeof(s_ccAttr) / sizeof(s_ccAttr[0]);
    UINT32      i;
    for(i = 0; i < size; i++) {
        if(s_ccAttr[i].commandIndex == (UINT16) commandCode)
            return s_ccAttr[i];
    }

    // This function should be called in the way that the command code
    // attribute is available.
    FAIL(FATAL_ERROR_INTERNAL);
}
```

9.4.3.4 EncryptSize()

This function returns the size of the decrypt size field. This function returns 0 if encryption is not allowed

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
int
EncryptSize(
    TPM_CC            commandCode    // IN: commandCode
    )
{
    COMMAND_ATTRIBUTES  ca = s_commandAttributes[commandCode - TPM_CC_FIRST];
    if(ca & ENCRYPT_2)
        return 2;
    if(ca & ENCRYPT_4)
        return 4;
    return 0;
```

```
}
```

9.4.3.5 DecryptSize()

This function returns the size of the decrypt size field. This function returns 0 if decryption is not allowed

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
int
DecryptSize(
    TPM_CC            commandCode    // IN: commandCode
    )
{
    COMMAND_ATTRIBUTES  ca = s_commandAttributes[commandCode - TPM_CC_FIRST];

    if(ca & DECRYPT_2)
        return 2;
    if(ca & DECRYPT_4)
        return 4;
    return 0;
}
```

9.4.3.6 IsSessionAllowed()

This function indicates if the command is allowed to have sessions.

This function must not be called if the command is not known to be implemented.

Return Value	Meaning
TRUE	session is allowed with this command
FALSE	session is not allowed with this command

```
BOOL
IsSessionAllowed(
    TPM_CC            commandCode    // IN: the command to be checked
    )
{
    if(s_commandAttributes[commandCode - TPM_CC_FIRST] & NO_SESSIONS)
        return FALSE;
    else
        return TRUE;
}
```

9.4.3.7 IsHandleInResponse()

```
BOOL
IsHandleInResponse(
    TPM_CC            commandCode
    )
{
    if(s_commandAttributes[commandCode - TPM_CC_FIRST] & R_HANDLE)
        return TRUE;
    else
        return FALSE;
```

```
95  }
```

9.4.3.8 IsWriteOperation()

Checks to see if an operation will write to NV memory

```
96  BOOL
97  IsWriteOperation(
98      TPM_CC            command         // IN: Command to check
99      )
100 {
101     switch (command)
102     {
103         case TPM_CC_NV_Write:
104         case TPM_CC_NV_Increment:
105         case TPM_CC_NV_SetBits:
106         case TPM_CC_NV_Extend:
107         // Nv write lock counts as a write operation for authorization purposes.
108         // We check to see if the NV is write locked before we do the authorization
109         // If it is locked, we fail the command early.
110         case TPM_CC_NV_WriteLock:
111             return TRUE;
112         default:
113             break;
114     }
115     return FALSE;
116 }
```

9.4.3.9 IsReadOperation()

Checks to see if an operation will write to NV memory

```
117 BOOL
118 IsReadOperation(
119     TPM_CC            command         // IN: Command to check
120     )
121 {
122     switch (command)
123     {
124         case TPM_CC_NV_Read:
125         case TPM_CC_PolicyNV:
126         case TPM_CC_NV_Certify:
127         // Nv read lock counts as a read operation for authorization purposes.
128         // We check to see if the NV is read locked before we do the authorization
129         // If it is locked, we fail the command early.
130         case TPM_CC_NV_ReadLock:
131             return TRUE;
132         default:
133             break;
134     }
135     return FALSE;
136 }
```

9.4.3.10 CommandCapGetCCList()

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

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

```
137 TPMI_YES_NO
138 CommandCapGetCCList(
139     TPM_CC            commandCode,   // IN: start command code
140     UINT32            count,         // IN: maximum count for number of entries in
141                                      //     'commandList'
142     TPML_CCA         *commandList    // OUT: list of TPMA_CC
143     )
144 {
145     TPMI_YES_NO       more = NO;
146     UINT32            i;
147
148     // initialize output handle list count
149     commandList->count = 0;
150
151     // The maximum count of commands that may be return is MAX_CAP_CC.
152     if(count > MAX_CAP_CC) count = MAX_CAP_CC;
153
154     // If the command code is smaller than TPM_CC_FIRST, start from TPM_CC_FIRST
155     if(commandCode < TPM_CC_FIRST) commandCode = TPM_CC_FIRST;
156
157     // Collect command attributes
158     for(i = commandCode; i <= TPM_CC_LAST; i++)
159     {
160         if(CommandIsImplemented(i))
161         {
162             if(commandList->count < count)
163             {
164                 // If the list is not full, add the attributes for this command.
165                 commandList->commandAttributes[commandList->count]
166                     = CommandGetAttribute(i);
167                 commandList->count++;
168             }
169             else
170             {
171                 // If the list is full but there are more commands to report,
172                 // indicate this and return.
173                 more = YES;
174                 break;
175             }
176         }
177     }
178     return more;
179 }
```

9.5 DRTM.c

9.5.1 Description

This file contains functions that simulate the DRTM events. Its primary purpose is to isolate the name space of the simulator from the name space of the TPM. This is only an issue with the parameters to _TPM_Hash_Data().

9.5.2 Includes

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

9.5.3 Functions

9.5.3.1 Signal_Hash_Start()

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

```
LIB_EXPORT void
Signal_Hash_Start(
    void
    )
{
    _TPM_Hash_Start();
    return;
}
```

9.5.3.2 Signal_Hash_Data()

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

```
LIB_EXPORT void
Signal_Hash_Data(
    unsigned int     size,
    unsigned char   *buffer
    )
{
    _TPM_Hash_Data(size, buffer);
    return;
}
```

9.5.3.3 Signal_Hash_End()

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

```
LIB_EXPORT void
Signal_Hash_End(
    void
    )
{
    _TPM_Hash_End();
    return;
}
```

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

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

9.6.3 Functions

9.6.3.1 EntityGetLoadStatus()

This function will indicate if the entity associated with a handle is present in TPM memory. If the handle is a persistent object handle, and the object exists, the persistent object is moved from NV memory into a RAM object slot and the persistent handle is replaced with the transient object handle for the slot.

Error Returns	Meaning
TPM_RC_HANDLE	handle type does not match
TPM_RC_REFERENCE_H0	entity is not present
TPM_RC_HIERARCHY	entity belongs to a disabled hierarchy
TPM_RC_OBJECT_MEMORY	handle is an evict object but there is no space to load it to RAM

```
TPM_RC
EntityGetLoadStatus(
    TPM_HANDLE      *handle,        // IN/OUT: handle of the entity
    TPM_CC           commandCode    // IN: the commmandCode
    )
{
    TPM_RC          result = TPM_RC_SUCCESS;

    switch(HandleGetType(*handle))
    {
        // For handles associated with hierarchies, the entity is present
        // only if the associated enable is SET.
        case TPM_HT_PERMANENT:
            switch(*handle)
            {
                case TPM_RH_OWNER:
                    if(!gc.shEnable)
                        result = TPM_RC_HIERARCHY;
                    break;

#ifdef  VENDOR_PERMANENT
                case VENDOR_PERMANENT:
#endif
                case TPM_RH_ENDORSEMENT:
                    if(!gc.ehEnable)
                        result = TPM_RC_HIERARCHY;
                    break;
                case TPM_RH_PLATFORM:
                    if(!g_phEnable)
                        result = TPM_RC_HIERARCHY;
                    break;
                    // null handle, PW session handle and lockout
                    // handle are always available
                case TPM_RH_NULL:
                case TPM_RS_PW:
                case TPM_RH_LOCKOUT:
                    break;
                default:
                    // handling of the manufacture_specific handles
                    if(     ((TPM_RH)*handle >= TPM_RH_AUTH_00)
                        &&  ((TPM_RH)*handle <= TPM_RH_AUTH_FF))
                        // use the value that would have been returned from
                        // unmarshaling if it did the handle filtering
                            result = TPM_RC_VALUE;
                    else
                        pAssert(FALSE);
                    break;
```

```
49              }
50              break;
51          case TPM_HT_TRANSIENT:
52              // For a transient object, check if the handle is associated
53              // with a loaded object.
54              if(!ObjectIsPresent(*handle))
55                  result = TPM_RC_REFERENCE_H0;
56              break;
57          case TPM_HT_PERSISTENT:
58              // Persistent object
59              // Copy the persistent object to RAM and replace the handle with the
60              // handle of the assigned slot.  A TPM_RC_OBJECT_MEMORY,
61              // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
62              // ObjectLoadEvict()
63              result = ObjectLoadEvict(handle, commandCode);
64              break;
65          case TPM_HT_HMAC_SESSION:
66              // For an HMAC session, see if the session is loaded
67              // and if the session in the session slot is actually
68              // an HMAC session.
69              if(SessionIsLoaded(*handle))
70              {
71                  SESSION             *session;
72                  session = SessionGet(*handle);
73                  // Check if the session is a HMAC session
74                  if(session->attributes.isPolicy == SET)
75                      result = TPM_RC_HANDLE;
76              }
77              else
78                  result = TPM_RC_REFERENCE_H0;
79              break;
80          case TPM_HT_POLICY_SESSION:
81              // For a policy session, see if the session is loaded
82              // and if the session in the session slot is actually
83              // a policy session.
84              if(SessionIsLoaded(*handle))
85              {
86                  SESSION             *session;
87                  session = SessionGet(*handle);
88                  // Check if the session is a policy session
89                  if(session->attributes.isPolicy == CLEAR)
90                      result = TPM_RC_HANDLE;
91              }
92              else
93                  result = TPM_RC_REFERENCE_H0;
94              break;
95          case TPM_HT_NV_INDEX:
96              // For an NV Index, use the platform-specific routine
97              // to search the IN Index space.
98              result = NvIndexIsAccessible(*handle, commandCode);
99              break;
100         case TPM_HT_PCR:
101             // Any PCR handle that is unmarshaled successfully referenced
102             // a PCR that is defined.
103             break;
104         default:
105             // Any other handle type is a defect in the unmarshaling code.
106             pAssert(FALSE);
107             break;
108     }
109     return result;
110 }
```

9.6.3.2 EntityGetAuthValue()

This function is used to access the *authValue* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authValue* should have been verified by IsAuthValueAvailable().

This function copies the authorization value of the entity to *auth*.

Return value is the number of octets copied to *auth*.

```
111  UINT16
112  EntityGetAuthValue(
113      TPMI_DH_ENTITY   handle,        // IN: handle of entity
114      AUTH_VALUE      *auth           // OUT: authValue of the entity
115      )
116  {
117      TPM2B_AUTH       authValue = {0};
118
119      switch(HandleGetType(handle))
120      {
121          case TPM_HT_PERMANENT:
122              switch(handle)
123              {
124                  case TPM_RH_OWNER:
125                      // ownerAuth for TPM_RH_OWNER
126                      authValue = gp.ownerAuth;
127                      break;
128                  case TPM_RH_ENDORSEMENT:
129                      // endorsementAuth for TPM_RH_ENDORSEMENT
130                      authValue = gp.endorsementAuth;
131                      break;
132                  case TPM_RH_PLATFORM:
133                      // platformAuth for TPM_RH_PLATFORM
134                      authValue = gc.platformAuth;
135                      break;
136                  case TPM_RH_LOCKOUT:
137                      // lockoutAuth for TPM_RH_LOCKOUT
138                      authValue = gp.lockoutAuth;
139                      break;
140                  case TPM_RH_NULL:
141                      // nullAuth for TPM_RH_NULL. Return 0 directly here
142                      return 0;
143                      break;
144  #ifdef  VENDOR_PERMANENT
145                  case VENDOR_PERMANENT:
146                      // vendor auth value
147                      authValue = g_platformUniqueDetails;
148  #endif
149                  default:
150                      // If any other permanent handle is present it is
151                      // a code defect.
152                      pAssert(FALSE);
153                      break;
154              }
155              break;
156          case TPM_HT_TRANSIENT:
157              // authValue for an object
158              // A persistent object would have been copied into RAM
159              // and would have an transient object handle here.
160              {
161                  OBJECT          *object;
162                  object = ObjectGet(handle);
163                  // special handling if this is a sequence object
164                  if(ObjectIsSequence(object))
```

```
165                 {
166                     authValue = ((HASH_OBJECT *)object)->auth;
167                 }
168                 else
169                 {
170                     // Auth value is available only when the private portion of
171                     // the object is loaded.  The check should be made before
172                     // this function is called
173                     pAssert(object->attributes.publicOnly == CLEAR);
174                     authValue = object->sensitive.authValue;
175                 }
176             }
177             break;
178         case TPM_HT_NV_INDEX:
179             // authValue for an NV index
180             {
181                 NV_INDEX        nvIndex;
182                 NvGetIndexInfo(handle, &nvIndex);
183                 authValue = nvIndex.authValue;
184             }
185             break;
186         case TPM_HT_PCR:
187             // authValue for PCR
188             PCRGetAuthValue(handle, &authValue);
189             break;
190         default:
191             // If any other handle type is present here, then there is a defect
192             // in the unmarshaling code.
193             pAssert(FALSE);
194             break;
195     }
196
197     // Copy the authValue
198     pAssert(authValue.t.size <= sizeof(authValue.t.buffer));
199     MemoryCopy(auth, authValue.t.buffer, authValue.t.size, sizeof(TPMU_HA));
200
201     return authValue.t.size;
202 }
```

9.6.3.3 EntityGetAuthPolicy()

This function is used to access the *authPolicy* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authPolicy* should have been verified by IsAuthPolicyAvailable().

This function copies the authorization policy of the entity to *authPolicy*.

The return value is the hash algorithm for the policy.

```
203 TPMI_ALG_HASH
204 EntityGetAuthPolicy(
205     TPMI_DH_ENTITY   handle,        // IN: handle of entity
206     TPM2B_DIGEST    *authPolicy     // OUT: authPolicy of the entity
207     )
208 {
209     TPMI_ALG_HASH       hashAlg = TPM_ALG_NULL;
210
211     switch(HandleGetType(handle))
212     {
213         case TPM_HT_PERMANENT:
214             switch(handle)
215             {
216                 case TPM_RH_OWNER:
```

```
217                     // ownerPolicy for TPM_RH_OWNER
218                     *authPolicy = gp.ownerPolicy;
219                     hashAlg = gp.ownerAlg;
220                     break;
221                 case TPM_RH_ENDORSEMENT:
222                     // endorsementPolicy for TPM_RH_ENDORSEMENT
223                     *authPolicy = gp.endorsementPolicy;
224                     hashAlg = gp.endorsementAlg;
225                     break;
226                 case TPM_RH_PLATFORM:
227                     // platformPolicy for TPM_RH_PLATFORM
228                     *authPolicy = gc.platformPolicy;
229                     hashAlg = gc.platformAlg;
230                     break;
231                 case TPM_RH_LOCKOUT:
232                     // lockoutPolicy for TPM_RH_LOCKOUT
233                     *authPolicy = gp.lockoutPolicy;
234                     hashAlg = gp.lockoutAlg;
235                     break;
236                 default:
237                     // If any other permanent handle is present it is
238                     // a code defect.
239                     pAssert(FALSE);
240                     break;
241             }
242             break;
243         case TPM_HT_TRANSIENT:
244             // authPolicy for an object
245             {
246                 OBJECT *object = ObjectGet(handle);
247                 *authPolicy = object->publicArea.authPolicy;
248                 hashAlg = object->publicArea.nameAlg;
249             }
250             break;
251         case TPM_HT_NV_INDEX:
252             // authPolicy for a NV index
253             {
254                 NV_INDEX        nvIndex;
255                 NvGetIndexInfo(handle, &nvIndex);
256                 *authPolicy = nvIndex.publicArea.authPolicy;
257                 hashAlg = nvIndex.publicArea.nameAlg;
258             }
259             break;
260         case TPM_HT_PCR:
261             // authPolicy for a PCR
262             hashAlg = PCRGetAuthPolicy(handle, authPolicy);
263             break;
264         default:
265             // If any other handle type is present it is a code defect.
266             pAssert(FALSE);
267             break;
268     }
269     return hashAlg;
270 }
```

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.

```
271 UINT16
272 EntityGetName(
273     TPMI_DH_ENTITY   handle,        // IN: handle of entity
274     NAME             *name          // OUT: name of entity
```

```
275     )
276 {
277     UINT16          nameSize;
278 
279     switch(HandleGetType(handle))
280     {
281         case TPM_HT_TRANSIENT:
282             // Name for an object
283             nameSize = ObjectGetName(handle, name);
284             break;
285         case TPM_HT_NV_INDEX:
286             // Name for a NV index
287             nameSize = NvGetName(handle, name);
288             break;
289         default:
290             // For all other types, the handle is the Name
291             nameSize = TPM_HANDLE_Marshal(&handle, (BYTE **)&name, NULL);
292             break;
293     }
294     return nameSize;
295 }
```

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.

```
296 TPMI_RH_HIERARCHY
297 EntityGetHierarchy(
298     TPMI_DH_ENTITY   handle         // IN :handle of entity
299     )
300 {
301     TPMI_RH_HIERARCHY       hierarchy = TPM_RH_NULL;
302 
303     switch(HandleGetType(handle))
304     {
305         case TPM_HT_PERMANENT:
306             // hierarchy for a permanent handle
307             switch(handle)
308             {
309                 case TPM_RH_PLATFORM:
310                 case TPM_RH_ENDORSEMENT:
311                 case TPM_RH_NULL:
312                     hierarchy = handle;
313                     break;
314                 // all other permanent handles are associated with the owner
315                 // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
316                 default:
317                     hierarchy = TPM_RH_OWNER;
318                     break;
319             }
320             break;
321         case TPM_HT_NV_INDEX:
322             // hierarchy for NV index
323             {
324                 NV_INDEX        nvIndex;
325                 NvGetIndexInfo(handle, &nvIndex);
326                 // If only the platform can delete the index, then it is
```

```
327                 // considered to be in the platform hierarchy, otherwise it
328                 // is in the owner hierarchy.
329                 if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
330                     hierarchy = TPM_RH_PLATFORM;
331                 else
332                     hierarchy = TPM_RH_OWNER;
333             }
334             break;
335         case TPM_HT_TRANSIENT:
336             // hierarchy for an object
337             {
338                 OBJECT          *object;
339                 object = ObjectGet(handle);
340                 if(object->attributes.ppsHierarchy)
341                 {
342                     hierarchy = TPM_RH_PLATFORM;
343                 }
344                 else if(object->attributes.epsHierarchy)
345                 {
346                     hierarchy = TPM_RH_ENDORSEMENT;
347                 }
348                 else if(object->attributes.spsHierarchy)
349                 {
350                     hierarchy = TPM_RH_OWNER;
351                 }
352 
353             }
354             break;
355         case TPM_HT_PCR:
356             hierarchy = TPM_RH_OWNER;
357             break;
358         default:
359             pAssert(0);
360             break;
361     }
362     // this is unreachable but it provides a return value for the default
363     // case which makes the complier happy
364     return hierarchy;
365 }
```

9.7 Global.c

9.7.1 Description

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

9.7.2 Includes and Defines

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

9.7.3 Global Data Values

These values are visible across multiple modules.

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

```
                                        };
TPM_HANDLE          g_exclusiveAuditSession;
UINT64              g_time;
BOOL                g_pcrReConfig;
TPMI_DH_OBJECT      g_DRTMHandle;
BOOL                g_DrtmPreStartup;
BOOL                g_StartupLocality3;
BOOL                g_clearOrderly;
TPM_SU              g_prevOrderlyState;
BOOL                g_updateNV;
BOOL                g_nvOk;
TPM2B_AUTH          g_platformUniqueDetails;
STATE_CLEAR_DATA    gc;
STATE_RESET_DATA    gr;
PERSISTENT_DATA     gp;
ORDERLY_DATA        go;
```

9.7.4 Private Values

9.7.4.1 SessionProcess.c

```
#ifndef __IGNORE_STATE__        // DO NOT DEFINE THIS VALUE
```

These values do not need to be retained between commands.

```
TPM_HANDLE          s_sessionHandles[MAX_SESSION_NUM];
TPMA_SESSION        s_attributes[MAX_SESSION_NUM];
TPM_HANDLE          s_associatedHandles[MAX_SESSION_NUM];
TPM2B_NONCE         s_nonceCaller[MAX_SESSION_NUM];
TPM2B_AUTH          s_inputAuthValues[MAX_SESSION_NUM];
UINT32              s_encryptSessionIndex;
UINT32              s_decryptSessionIndex;
UINT32              s_auditSessionIndex;
TPM2B_DIGEST        s_cpHashForAudit;
UINT32              s_sessionNum;
#endif  // __IGNORE_STATE__
BOOL                s_DAPendingOnNV;
#ifdef TPM_CC_GetCommandAuditDigest
TPM2B_DIGEST        s_cpHashForCommandAudit;
#endif
```

9.7.4.2 DA.c

```
UINT64              s_selfHealTimer;
UINT64              s_lockoutTimer;
```

9.7.4.3 NV.c

```
UINT32              s_reservedAddr[NV_RESERVE_LAST];
UINT32              s_reservedSize[NV_RESERVE_LAST];
UINT32              s_ramIndexSize;
BYTE                s_ramIndex[RAM_INDEX_SPACE];
UINT32              s_ramIndexSizeAddr;
UINT32              s_ramIndexAddr;
UINT32              s_maxCountAddr;
UINT32              s_evictNvStart;
UINT32              s_evictNvEnd;
TPM_RC              s_NvStatus;
```

9.7.4.4 Object.c

```
52  OBJECT_SLOT          s_objects[MAX_LOADED_OBJECTS];
```

9.7.4.5 PCR.c

```
53  PCR                  s_pcrs[IMPLEMENTATION_PCR];
```

9.7.4.6 Session.c

```
54  SESSION_SLOT         s_sessions[MAX_LOADED_SESSIONS];
55  UINT32               s_oldestSavedSession;
56  int                  s_freeSessionSlots;
```

9.7.4.7 Manufacture.c

```
57  BOOL                 g_manufactured = FALSE;
```

9.7.4.8 Power.c

```
58  BOOL                 s_initialized = FALSE;
```

9.7.4.9 MemoryLib.c

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

```
59  #ifndef __IGNORE_STATE__         // DO NOT DEFINE THIS VALUE
60  UINT32   s_actionInputBuffer[1024];          // action input buffer
61  UINT32   s_actionOutputBuffer[1024];         // action output buffer
62  BYTE     s_responseBuffer[MAX_RESPONSE_SIZE];// response buffer
63  #endif
```

9.7.4.10 SelfTest.c

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

```
64  #ifndef SELF_TEST
65  ALGORITHM_VECTOR     g_implementedAlgorithms;
66  ALGORITHM_VECTOR     g_toTest;
67  #endif
```

9.7.4.11 TpmFail.c

```
68  jmp_buf              g_jumpBuffer;
69  BOOL                 g_forceFailureMode;
70  BOOL                 g_inFailureMode;
71  UINT32               s_failFunction;
72  UINT32               s_failLine;
73  UINT32               s_failCode;
```

9.8 Handle.c

9.8.1 Description

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

9.8.2 Includes

```
#include "Tpm.h"
#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.

```
TPM_HT
HandleGetType(
    TPM_HANDLE       handle          // IN: a handle to be checked
    )
{
    // return the upper bytes of input data
    return (TPM_HT) ((handle & HR_RANGE_MASK) >> HR_SHIFT);
}
```

9.8.3.2 NextPermanentHandle()

This function returns the permanent handle that is equal to the input value or is the next higher value. If there is no handle with the input value and there is no next higher value, it returns 0:

Return Value	Meaning

```
TPM_HANDLE
NextPermanentHandle(
    TPM_HANDLE       inHandle        // IN: the handle to check
    )
{
    // If inHandle is below the start of the range of permanent handles
    // set it to the start and scan from there
    if(inHandle < TPM_RH_FIRST)
        inHandle = TPM_RH_FIRST;
    // scan from input value untill we find an implemented permanent handle
    // or go out of range
    for(; inHandle <= TPM_RH_LAST; inHandle++)
    {
        switch (inHandle)
        {
            case TPM_RH_OWNER:
            case TPM_RH_NULL:
            case TPM_RS_PW:
            case TPM_RH_LOCKOUT:
            case TPM_RH_ENDORSEMENT:
            case TPM_RH_PLATFORM:
            case TPM_RH_PLATFORM_NV:
    #ifdef  VENDOR_PERMANENT
            case VENDOR_PERMANENT:
    #endif
                return inHandle;
```

```
37                  break;
38              default:
39                  break;
40          }
41      }
42      // Out of range on the top
43      return 0;
44  }
```

9.8.3.3 PermanentCapGetHandles()

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

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

```
45  TPMI_YES_NO
46  PermanentCapGetHandles(
47      TPM_HANDLE       handle,         // IN: start handle
48      UINT32           count,          // IN: count of returned handle
49      TPML_HANDLE     *handleList      // OUT: list of handle
50      )
51  {
52      TPMI_YES_NO     more = NO;
53      UINT32          i;
54
55      pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
56
57      // Initialize output handle list
58      handleList->count = 0;
59
60      // The maximum count of handles we may return is MAX_CAP_HANDLES
61      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
62
63      // Iterate permanent handle range
64      for(i = NextPermanentHandle(handle);
65              i != 0; i = NextPermanentHandle(i+1))
66      {
67          if(handleList->count < count)
68          {
69              // If we have not filled up the return list, add this permanent
70              // handle to it
71              handleList->handle[handleList->count] = i;
72              handleList->count++;
73          }
74          else
75          {
76              // If the return list is full but we still have permanent handle
77              // available, report this and stop iterating
78              more = YES;
79              break;
80          }
81      }
82      return more;
83  }
```

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.

```
TPMA_LOCALITY
LocalityGetAttributes(
    UINT8             locality       // IN: locality value
    )
{
    TPMA_LOCALITY            locality_attributes;
    BYTE                    *localityAsByte = (BYTE *)&locality_attributes;

    MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
    switch(locality)
    {
        case 0:
            locality_attributes.TPM_LOC_ZERO = SET;
            break;
        case 1:
            locality_attributes.TPM_LOC_ONE = SET;
            break;
        case 2:
            locality_attributes.TPM_LOC_TWO = SET;
            break;
        case 3:
            locality_attributes.TPM_LOC_THREE = SET;
            break;
        case 4:
            locality_attributes.TPM_LOC_FOUR = SET;
            break;
        default:
            pAssert(locality < 256 && locality > 31);
            *localityAsByte = locality;
            break;
    }
    return locality_attributes;
}
```

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

```
#define MANUFACTURE_C
#include "InternalRoutines.h"
#include "Global.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 re-manufactured by calling TPM_Teardown() first and then calling this function again.

Return Value	Meaning
0	success
1	manufacturing process previously performed

```
4   LIB_EXPORT int
5   TPM_Manufacture(
6       BOOL             firstTime      // IN: indicates if this is the first call from
7                                       //     main()
8       )
9   {
10      TPM_SU          orderlyShutdown;
11      UINT64          totalResetCount = 0;
12
13      // If TPM has been manufactured, return indication.
14      if(!firstTime && g_manufactured)
15          return 1;
16
17      // initialize crypto units
18      //CryptInitUnits();
19
20      //
21      s_selfHealTimer = 0;
22      s_lockoutTimer = 0;
23      s_DAPendingOnNV = FALSE;
24
25      // initialize NV
26      NvInit();
27
28  #ifdef _DRBG_STATE_SAVE
29      // Initialize the drbg. This needs to come before the install
30      // of the hierarchies
31      if(!_cpri__Startup())               // Have to start the crypto units first
32          FAIL(FATAL_ERROR_INTERNAL);
33      _cpri__DrbgGetPutState(PUT_STATE, 0, NULL);
34  #endif
35
36      // default configuration for PCR
37      PCRSimStart();
38
39      // initialize pre-installed hierarchy data
40      // This should happen after NV is initialized because hierarchy data is
41      // stored in NV.
42      HierarchyPreInstall_Init();
43
44      // initialize dictionary attack parameters
45      DAPreInstall_Init();
46
47      // initialize PP list
48      PhysicalPresencePreInstall_Init();
49
50      // initialize command audit list
51      CommandAuditPreInstall_Init();
52
53      // first start up is required to be Startup(CLEAR)
```

```
54          orderlyShutdown = TPM_SU_CLEAR;
55          NvWriteReserved(NV_ORDERLY, &orderlyShutdown);
56
57          // initialize the firmware version
58          gp.firmwareV1 = FIRMWARE_V1;
59      #ifdef FIRMWARE_V2
60          gp.firmwareV2 = FIRMWARE_V2;
61      #else
62          gp.firmwareV2 = 0;
63      #endif
64          NvWriteReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
65          NvWriteReserved(NV_FIRMWARE_V2, &gp.firmwareV2);
66
67          // initialize the total reset counter to 0
68          NvWriteReserved(NV_TOTAL_RESET_COUNT, &totalResetCount);
69
70          // initialize the clock stuff
71          go.clock = 0;
72          go.clockSafe = YES;
73
74      #ifdef _DRBG_STATE_SAVE
75          // initialize the current DRBG state in NV
76
77          _cpri__DrbgGetPutState(GET_STATE, sizeof(go.drbgState), (BYTE *)&go.drbgState);
78      #endif
79
80          NvWriteReserved(NV_ORDERLY_DATA, &go);
81
82          // Commit NV writes.  Manufacture process is an artificial process existing
83          // only in simulator environment and it is not defined in the specification
84          // that what should be the expected behavior if the NV write fails at this
85          // point.  Therefore, it is assumed the NV write here is always success and
86          // no return code of this function is checked.
87          NvCommit();
88
89          g_manufactured = TRUE;
90
91          return 0;
92      }
```

9.10.3.2 TPM_TearDown()

This function prepares the TPM for re-manufacture. It should not be implemented in anything other than a simulated TPM.

In this implementation, all that is needs is to stop the cryptographic units and set a flag to indicate that the TPM can be re-manufactured. This should be all that is necessary to start the manufacturing process again.

Return Value	Meaning
0	success
1	TPM not previously manufactured

```
93      LIB_EXPORT int
94      TPM_TearDown(
95          void
96          )
97      {
98          // stop crypt units
99          CryptStopUnits();
100
101         g_manufactured = FALSE;
```

```
102         return 0;
103     }
```

9.11 Marshal.c

9.11.1 Introduction

This file contains the marshaling and unmarshaling code.

The marshaling and unmarshaling code and function prototypes are not listed, as the code is repetitive, long, and not very useful to read. Examples of a few unmarshaling routines are provided. Most of the others are similar.

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets ("<>") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

9.11.2 Unmarshal and Marshal a Value

In TPM 2.0 Part 2, a TPMI_DI_OBJECT is defined by this table:

Table xxx — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```
1   TPM_RC
2   TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
3                            bool flag)
4   {
5       TPM_RC    result;
6       result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       if (*target == TPM_RH_NULL) {
10          if(flag)
11              return TPM_RC_SUCCESS;
12          else
13              return TPM_RC_VALUE;
14      }
15      if((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
16          if((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST))
17              return TPM_RC_VALUE;
18      return TPM_RC_SUCCESS;
19  }
```


and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```
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 TPM 2.0 Part 2, a TPMU_PUBLIC_PARMS union is defined by:

Table xxx — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S>

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign | encrypt | neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys
NOTE The Description column indicates which of TPMA_OBJECT.*decrypt* or TPMA_OBJECT.*sign* may be set. "+" indicates that both may be set but one shall be set. "|" indicates the optional settings.			

From this table, the following unmarshaling code is generated.

```
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:

```
1   UINT16
2   TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
3                             UINT32 selector)
4   {
5       switch(selector) {
6   #ifdef TPM_ALG_KEYEDHASH
7           case TPM_ALG_KEYEDHASH:
8               return TPMS_KEYEDHASH_PARMS_Marshal(
9                           (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
10  #endif
11  #ifdef TPM_ALG_SYMCIPHER
12          case TPM_ALG_SYMCIPHER:
13              return TPMT_SYM_DEF_OBJECT_Marshal(
14                          (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
15  #endif
16  #ifdef TPM_ALG_RSA
17          case TPM_ALG_RSA:
18              return TPMS_RSA_PARMS_Marshal(
19                           (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
20  #endif
21  #ifdef TPM_ALG_ECC
22          case TPM_ALG_ECC:
23              return TPMS_ECC_PARMS_Marshal(
24                           (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
25  #endif
26      }
27      assert(1);
28      return 0;
29  }
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

9.11.4 Unmarshal and Marshal a Structure

In TPM 2.0 Part 2, the TPMT_PUBLiC structure is defined by:

Table xxx — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	“algorithm” associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```
1   TPM_RC
2   TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, bool flag)
3   {
4       TPM_RC    result;
5       result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                          buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                       buffer, size, flag);
11      if(result != TPM_RC_SUCCESS)
12          return result;
13      result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                     buffer, size);
15      if(result != TPM_RC_SUCCESS)
16          return result;
17      result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                      buffer, size);
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                           buffer, size, (UINT32)target->type);
24      if(result != TPM_RC_SUCCESS)
25          return result;
26
27      result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                        buffer, size, (UINT32)target->type);
29      if(result != TPM_RC_SUCCESS)
30          return result;
31
32      return TPM_RC_SUCCESS;
33  }
```

The marshaling code for the TPMT_PUBLIC structure is:

```
1   UINT16
2   TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
3   {
4       UINT16    result = 0;
5       result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
6                                      (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
7       result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
8                                     (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
9   ;
10      result = (UINT16)(result + TPMA_OBJECT_Marshal(
11                               (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
12
13      result = (UINT16)(result + TPM2B_DIGEST_Marshal(
14                                  (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));
15
16      result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
17                               (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
18                                                                    (UINT32)source->type));
19
20      result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
21                                    (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
22                                                                    (UINT32)source->type));
23
24      return result;
25  }
```

9.11.5 Unmarshal and Marshal an Array

In TPM 2.0 Part 2, the TPML_DIGEST is defined by:

Table xxx — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```
1   TPM_RC
2   TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8
9       if( (target->count < 2))     // This check is triggered by the {2:} notation
10                                   // on 'count'
11          return TPM_RC_SIZE;
12
13      if((target->count) > 8)      // This check is triggered by the {:8} notation
14                                   // on 'digests'.
15          return TPM_RC_SIZE;
16
17      result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
18                                            buffer, size, (INT32)(target->count));
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      return TPM_RC_SUCCESS;
23  }
```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```
1   TPM_RC
2   TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                                INT32 count)
4   {
5       TPM_RC    result;
6       INT32 i;
7       for(i = 0; i < count; i++) {
8           result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9           if(result != TPM_RC_SUCCESS)
10              return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
14
```

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
1   UINT16
2   TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
3   {
4       UINT16    result = 0;
5       result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
6                                                                                 size));
7       result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
8                                       (TPM2B_DIGEST *)(source->digests), buffer, size,
9                                       (INT32)(source->count)));
10
11      return result;
12  }
```

The marshaling code for the array is:

```
1   TPM_RC
2   TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                                INT32 count)
4   {
5       TPM_RC    result;
6       INT32 i;
7       for(i = 0; i < count; i++) {
8           result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9           if(result != TPM_RC_SUCCESS)
10              return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
```

9.11.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 10.11.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

Table xxx — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

```
1   TPM_RC
2   TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8
9       // if size equal to 0, the rest of the structure is a zero buffer.  Stop
    processing
10      if(target->t.size == 0)
11          return TPM_RC_SUCCESS;
12
13      if((target->t.size) > 1024)     // This check is triggered by the {:1024} notation
14                                      // on 'buffer'
15          return TPM_RC_SIZE;
16
17      result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
18                                    (INT32)(target->t.size));
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      return TPM_RC_SUCCESS;
23  }
```

Which use these structure definitions:

```
1   typedef struct {
2       UINT16        size;
3       BYTE          buffer[1];
4   } TPM2B;
5
6   typedef struct {
7       UINT16        size;
8       BYTE          buffer[1024];
9   } EVENT_2B;
10
11  typedef union {
12      EVENT_2B      t;    // The type-specific union member
13      TPM2B         b;    // The generic union member
14  } TPM2B_EVENT;
```

9.12 MemoryLib.c

9.12.1 Description

This file contains a set of miscellaneous memory manipulation routines. Many of the functions have the same semantics as functions defined in string.h. Those functions are not used in the TPM in order to avoid namespace contamination.

9.12.2 Includes and Data Definitions

```
#define MEMORY_LIB_C
#include "InternalRoutines.h"
```

These buffers are set aside to hold command and response values. In this implementation, it is not guaranteed that the code will stop accessing the *s_actionInputBuffer* before starting to put values in the *s_actionOutputBuffer* so different buffers are required. However, the *s_actionInputBuffer* and *s_responseBuffer* are not needed at the same time and they could be the same buffer.

9.12.3 Functions on BYTE Arrays

9.12.3.1 MemoryMove()

This function moves data from one place in memory to another. No safety checks of any type are performed. If source and data buffer overlap, then the move is done as if an intermediate buffer were used.

NOTE: This function is used by MemoryCopy(), MemoryCopy2B(), and MemoryConcat2b() and requires that the caller know the maximum size of the destination buffer so that there is no possibility of buffer overrun.

```
LIB_EXPORT void
MemoryMove(
    void            *destination,   // OUT: move destination
    const void      *source,        // IN: move source
    UINT32           size,          // IN: number of octets to moved
    UINT32           dSize          // IN: size of the receive buffer
    )
{
    const BYTE *p = (BYTE *)source;
    BYTE *q = (BYTE *)destination;

    if(destination == NULL || source == NULL)
        return;

    pAssert(size <= dSize);
    // if the destination buffer has a lower address than the
    // source, then moving bytes in ascending order is safe.
    dSize -= size;

    if (p>q || (p+size <= q))
    {
        while(size--)
            *q++ = *p++;
    }
    // If the destination buffer has a higher address than the
    // source, then move bytes from the end to the beginning.
    else if (p < q)
    {
        p += size;
        q += size;
```

```
33          while (size--)
34              *--q = *--p;
35      }
36
37      // If the source and destination address are the same, nothing to move.
38      return;
39  }
```

9.12.3.2 MemoryCopy()

This function moves data from one place in memory to another. No safety checks of any type are performed. If the destination and source overlap, then the results are unpredictable. void *MemoryCopy*(

void	*destination, // OUT: copy destination
void	*source, // IN: copy source
UINT32	size, // IN: number of octets being copied
UINT32	*dSize* // IN: size of the receive buffer

MemoryMove(destination, source, size, *dSize*);

```
40  //%#define MemoryCopy(destination, source, size, destSize)        \
41  //%    MemoryMove((destination), (source), (size), (destSize))
```

9.12.3.3 MemoryEqual()

This function indicates if two buffers have the same values in the indicated number of bytes.

Return Value	Meaning
TRUE	all octets are the same
FALSE	all octets are not the same

```
42  LIB_EXPORT BOOL
43  MemoryEqual(
44      const void      *buffer1,       // IN: compare buffer1
45      const void      *buffer2,       // IN: compare buffer2
46      UINT32           size           // IN: size of bytes being compared
47      )
48  {
49      BOOL         equal = TRUE;
50      const BYTE  *b1, *b2;
51
52      b1 = (BYTE *)buffer1;
53      b2 = (BYTE *)buffer2;
54
55      // Compare all bytes so that there is no leakage of information
56      // due to timing differences.
57      for(; size > 0; size--)
58          equal = (*b1++ == *b2++) && equal;
59
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.

```
LIB_EXPORT INT16
MemoryCopy2B(
    TPM2B           *dest,          // OUT: receiving TPM2B
    const TPM2B     *source,        // IN: source TPM2B
    UINT16           dSize          // IN: size of the receiving buffer
    )
{

    if(dest == NULL)
        return 0;
    if(source == NULL)
        dest->size = 0;
    else
    {
        dest->size = source->size;
        MemoryMove(dest->buffer, source->buffer, dest->size, dSize);
    }
    return dest->size;
}
```

9.12.3.5 MemoryConcat2B()

This function will concatenate the buffer contents of a TPM2B to an the buffer contents of another TPM2B and adjust the size accordingly *(a* := *(a* | *b)*).

```
LIB_EXPORT void
MemoryConcat2B(
    TPM2B           *aInOut,        // IN/OUT: destination 2B
    TPM2B           *bIn,           // IN: second 2B
    UINT16           aSize          // IN: The size of aInOut.buffer (max values for
                                    //     aInOut.size)
    )
{
    MemoryMove(&aInOut->buffer[aInOut->size],
               bIn->buffer,
               bIn->size,
               aSize - aInOut->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.

Return Value	Meaning
TRUE	size and buffer contents are the same
FALSE	size or buffer contents are not the same

```
LIB_EXPORT BOOL
Memory2BEqual(
    const TPM2B     *aIn,           // IN: compare value
    const TPM2B     *bIn            // IN: compare value
    )
{
    if(aIn->size != bIn->size)
        return FALSE;
```

```
104
105     return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
106 }
```

9.12.3.7 MemorySet()

This function will set all the octets in the specified memory range to the specified octet value.

NOTE: the **dSize** parameter forces the caller to know how big the receiving buffer is to make sure that there is no possibility that the caller will inadvertently run over the end of the buffer.

```
107 LIB_EXPORT void
108 MemorySet(
109     void            *destination,   // OUT: memory destination
110     char             value,         // IN: fill value
111     UINT32           size           // IN: number of octets to fill
112     )
113 {
114     char *p = (char *)destination;
115     while (size--)
116         *p++ = value;
117     return;
118 }
```

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.

```
119 BYTE *
120 MemoryGetActionInputBuffer(
121     UINT32           size           // Size, in bytes, required for the input
122                                     // unmarshaling
123     )
124 {
125     BYTE        *buf = NULL;
126
127     if(size > 0)
128     {
129         // In this implementation, a static buffer is set aside for action output.
130         // Other implementations may apply additional optimization based on command
131         // code or other factors.
132         UINT32      *p = s_actionInputBuffer;
133         buf = (BYTE *)p;
134         pAssert(size < sizeof(s_actionInputBuffer));
135
136         // size of an element in the buffer
137 #define SZ      sizeof(s_actionInputBuffer[0])
138
139         for(size = (size + SZ - 1) / SZ; size > 0; size--)
140             *p++ = 0;
141 #undef SZ
142     }
143     return buf;
144 }
```

9.12.3.9 MemoryGetActionOutputBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```
145 void *
146 MemoryGetActionOutputBuffer(
147     TPM_CC            command        // Command that requires the buffer
148     )
149 {
150     // In this implementation, a static buffer is set aside for action output.
151     // Other implementations may apply additional optimization based on the command
152     // code or other factors.
153     command = 0;        // Unreferenced parameter
154     return s_actionOutputBuffer;
155 }
```

9.12.3.10 MemoryGetResponseBuffer()

This function returns the address into which the command response is marshaled from values in the action output buffer.

```
156 BYTE *
157 MemoryGetResponseBuffer(
158     TPM_CC            command        // Command that requires the buffer
159     )
160 {
161     // In this implementation, a static buffer is set aside for responses.
162     // Other implementation may apply additional optimization based on the command
163     // code or other factors.
164     command = 0;        // Unreferenced parameter
165     return s_responseBuffer;
166 }
```

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.

```
167 UINT16
168 MemoryRemoveTrailingZeros (
169     TPM2B_AUTH      *auth           // IN/OUT: value to adjust
170     )
171 {
172     BYTE        *a = &auth->t.buffer[auth->t.size-1];
173     for(; auth->t.size > 0; auth->t.size--)
174     {
175         if(*a--)
176             break;
177     }
178     return auth->t.size;
179 }
```

9.13 Power.c

9.13.1 Description

This file contains functions that receive the simulated power state transitions of the TPM.

9.13.2 Includes and Data Definitions

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

9.13.3 Functions

9.13.3.1 TPMInit()

This function is used to process a power on event.

```
void
TPMInit(
    void
    )
{
    // Set state as not initialized. This means that Startup is required
    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
TPMRegisterStartup(
    void
    )
{
    s_initialized = TRUE;

    return;
}
```

9.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2_Startup() has completed successfully after a _TPM_Init()).

Return Value	Meaning
TRUE	TPM has been initialized
FALSE	TPM has not been initialized

```
BOOL
TPMIsStarted(
    void
    )
{
    return s_initialized;
}
```

9.14 PropertyCap.c

9.14.1 Description

This file contains the functions that are used for accessing the TPM_CAP_TPM_PROPERTY values.

9.14.2 Includes

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

9.14.3 Functions

9.14.3.1 PCRGetProperty()

This function accepts a property selection and, if so, sets *value* to the value of the property.

All the fixed values are vendor dependent or determined by a platform-specific specification. The values in the table below are examples and should be changed by the vendor.

Return Value	Meaning
TRUE	referenced property exists and *value* set
FALSE	referenced property does not exist

```
static BOOL
TPMPropertyIsDefined(
    TPM_PT           property,      // IN: property
    UINT32          *value          // OUT: property value
    )
{
    switch(property)
    {
        case TPM_PT_FAMILY_INDICATOR:
            // from the title page of the specification
            // For this specification, the value is "2.0".
            *value = TPM_SPEC_FAMILY;
            break;
        case TPM_PT_LEVEL:
            // from the title page of the specification
            *value = TPM_SPEC_LEVEL;
            break;
        case TPM_PT_REVISION:
            // from the title page of the specification
            *value = TPM_SPEC_VERSION;
            break;
        case TPM_PT_DAY_OF_YEAR:
            // computed from the date value on the title page of the specification
            *value = TPM_SPEC_DAY_OF_YEAR;
            break;
        case TPM_PT_YEAR:
            // from the title page of the specification
            *value = TPM_SPEC_YEAR;
            break;
        case TPM_PT_MANUFACTURER:
            // vendor ID unique to each TPM manufacturer
            *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
            break;
        case TPM_PT_VENDOR_STRING_1:
            // first four characters of the vendor ID string
            *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
            break;
        case TPM_PT_VENDOR_STRING_2:
            // second four characters of the vendor ID string
#ifdef VENDOR_STRING_2
            *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
#else
            *value = 0;
#endif
```

```
46              break;
47          case TPM_PT_VENDOR_STRING_3:
48              // third four characters of the vendor ID string
49  #ifdef VENDOR_STRING_3
50              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
51  #else
52              *value = 0;
53  #endif
54              break;
55          case TPM_PT_VENDOR_STRING_4:
56              // fourth four characters of the vendor ID string
57  #ifdef VENDOR_STRING_4
58              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
59  #else
60              *value = 0;
61  #endif
62              break;
63          case TPM_PT_VENDOR_TPM_TYPE:
64              // vendor-defined value indicating the TPM model
65              *value = 1;
66              break;
67          case TPM_PT_FIRMWARE_VERSION_1:
68              // more significant 32-bits of a vendor-specific value
69              *value = gp.firmwareV1;
70              break;
71          case TPM_PT_FIRMWARE_VERSION_2:
72              // less significant 32-bits of a vendor-specific value
73              *value = gp.firmwareV2;
74              break;
75          case TPM_PT_INPUT_BUFFER:
76              // maximum size of TPM2B_MAX_BUFFER
77              *value = MAX_DIGEST_BUFFER;
78              break;
79          case TPM_PT_HR_TRANSIENT_MIN:
80              // minimum number of transient objects that can be held in TPM
81              // RAM
82              *value = MAX_LOADED_OBJECTS;
83              break;
84          case TPM_PT_HR_PERSISTENT_MIN:
85              // minimum number of persistent objects that can be held in
86              // TPM NV memory
87              // In this implementation, there is no minimum number of
88              // persistent objects.
89              *value = MIN_EVICT_OBJECTS;
90              break;
91          case TPM_PT_HR_LOADED_MIN:
92              // minimum number of authorization sessions that can be held in
93              // TPM RAM
94              *value = MAX_LOADED_SESSIONS;
95              break;
96          case TPM_PT_ACTIVE_SESSIONS_MAX:
97              // number of authorization sessions that may be active at a time
98              *value = MAX_ACTIVE_SESSIONS;
99              break;
100         case TPM_PT_PCR_COUNT:
101             // number of PCR implemented
102             *value = IMPLEMENTATION_PCR;
103             break;
104         case TPM_PT_PCR_SELECT_MIN:
105             // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
106             *value = PCR_SELECT_MIN;
107             break;
108         case TPM_PT_CONTEXT_GAP_MAX:
109             // maximum allowed difference (unsigned) between the contextID
110             // values of two saved session contexts
111             *value = (1 << (sizeof(CONTEXT_SLOT) * 8)) - 1;
```

```
112             break;
113         case TPM_PT_NV_COUNTERS_MAX:
114             // maximum number of NV indexes that are allowed to have the
115             // TPMA_NV_COUNTER attribute SET
116             // In this implementation, there is no limitation on the number
117             // of counters, except for the size of the NV Index memory.
118             *value = 0;
119             break;
120         case TPM_PT_NV_INDEX_MAX:
121             // maximum size of an NV index data area
122             *value = MAX_NV_INDEX_SIZE;
123             break;
124         case TPM_PT_MEMORY:
125             // a TPMA_MEMORY indicating the memory management method for the TPM
126         {
127             TPMA_MEMORY         attributes = {0};
128             attributes.sharedNV = SET;
129             attributes.objectCopiedToRam = SET;
130
131             // Note: Different compilers may require a different method to cast
132             // a bit field structure to a UINT32.
133             *value = * (UINT32 *) &attributes;
134             break;
135         }
136         case TPM_PT_CLOCK_UPDATE:
137             // interval, in seconds, between updates to the copy of
138             // TPMS_TIME_INFO .clock in NV
139             *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
140             break;
141         case TPM_PT_CONTEXT_HASH:
142             // algorithm used for the integrity hash on saved contexts and
143             // for digesting the fuData of TPM2_FirmwareRead()
144             *value = CONTEXT_INTEGRITY_HASH_ALG;
145             break;
146         case TPM_PT_CONTEXT_SYM:
147             // algorithm used for encryption of saved contexts
148             *value = CONTEXT_ENCRYPT_ALG;
149             break;
150         case TPM_PT_CONTEXT_SYM_SIZE:
151             // size of the key used for encryption of saved contexts
152             *value = CONTEXT_ENCRYPT_KEY_BITS;
153             break;
154         case TPM_PT_ORDERLY_COUNT:
155             // maximum difference between the volatile and non-volatile
156             // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
157             *value = MAX_ORDERLY_COUNT;
158             break;
159         case TPM_PT_MAX_COMMAND_SIZE:
160             // maximum value for 'commandSize'
161             *value = MAX_COMMAND_SIZE;
162             break;
163         case TPM_PT_MAX_RESPONSE_SIZE:
164             // maximum value for 'responseSize'
165             *value = MAX_RESPONSE_SIZE;
166             break;
167         case TPM_PT_MAX_DIGEST:
168             // maximum size of a digest that can be produced by the TPM
169             *value = sizeof(TPMU_HA);
170             break;
171         case TPM_PT_MAX_OBJECT_CONTEXT:
172             // maximum size of a TPMS_CONTEXT that will be returned by
173             // TPM2_ContextSave for object context
174             *value = 0;
175
176             // adding sequence, saved handle and hierarchy
177             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
```

```
178                     sizeof(TPMI_RH_HIERARCHY);
179             // add size field in TPM2B_CONTEXT
180             *value += sizeof(UINT16);
181
182             // add integrity hash size
183             *value += sizeof(UINT16) +
184                     CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
185
186             // Add fingerprint size, which is the same as sequence size
187             *value += sizeof(UINT64);
188
189             // Add OBJECT structure size
190             *value += sizeof(OBJECT);
191             break;
192         case TPM_PT_MAX_SESSION_CONTEXT:
193             // the maximum size of a TPMS_CONTEXT that will be returned by
194             // TPM2_ContextSave for object context
195             *value = 0;
196
197             // adding sequence, saved handle and hierarchy
198             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
199                     sizeof(TPMI_RH_HIERARCHY);
200             // Add size field in TPM2B_CONTEXT
201             *value += sizeof(UINT16);
202
203             // Add integrity hash size
204             *value += sizeof(UINT16) +
205                     CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
206             // Add fingerprint size, which is the same as sequence size
207             *value += sizeof(UINT64);
208
209             // Add SESSION structure size
210             *value += sizeof(SESSION);
211             break;
212         case TPM_PT_PS_FAMILY_INDICATOR:
213             // platform specific values for the TPM_PT_PS parameters from
214             // the relevant platform-specific specification
215             // In this reference implementation, all of these values are 0.
216             *value = 0;
217             break;
218         case TPM_PT_PS_LEVEL:
219             // level of the platform-specific specification
220             *value = 0;
221             break;
222         case TPM_PT_PS_REVISION:
223             // specification Revision times 100 for the platform-specific
224             // specification
225             *value = 0;
226             break;
227         case TPM_PT_PS_DAY_OF_YEAR:
228             // platform-specific specification day of year using TCG calendar
229             *value = 0;
230             break;
231         case TPM_PT_PS_YEAR:
232             // platform-specific specification year using the CE
233             *value = 0;
234             break;
235         case TPM_PT_SPLIT_MAX:
236             // number of split signing operations supported by the TPM
237             *value = 0;
238     #ifdef TPM_ALG_ECC
239             *value = sizeof(gr.commitArray) * 8;
240     #endif
241             break;
242         case TPM_PT_TOTAL_COMMANDS:
243             // total number of commands implemented in the TPM
```

```
244             // Since the reference implementation does not have any
245             // vendor-defined commands, this will be the same as the
246             // number of library commands.
247         {
248             UINT32 i;
249             *value = 0;
250 
251             // calculate implemented command numbers
252             for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
253             {
254                 if(CommandIsImplemented(i)) (*value)++;
255             }
256             break;
257         }
258         case TPM_PT_LIBRARY_COMMANDS:
259             // number of commands from the TPM library that are implemented
260         {
261             UINT32 i;
262             *value = 0;
263 
264             // calculate implemented command numbers
265             for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
266             {
267                 if(CommandIsImplemented(i)) (*value)++;
268             }
269             break;
270         }
271         case TPM_PT_VENDOR_COMMANDS:
272             // number of vendor commands that are implemented
273             *value = 0;
274             break;
275         case TPM_PT_PERMANENT:
276             // TPMA_PERMANENT
277         {
278             TPMA_PERMANENT          flags = {0};
279             if(gp.ownerAuth.t.size != 0)
280                 flags.ownerAuthSet = SET;
281             if(gp.endorsementAuth.t.size != 0)
282                 flags.endorsementAuthSet = SET;
283             if(gp.lockoutAuth.t.size != 0)
284                 flags.lockoutAuthSet = SET;
285             if(gp.disableClear)
286                 flags.disableClear = SET;
287             if(gp.failedTries >= gp.maxTries)
288                 flags.inLockout = SET;
289             // In this implementation, EPS is always generated by TPM
290             flags.tpmGeneratedEPS = SET;
291 
292             // Note: Different compilers may require a different method to cast
293             // a bit field structure to a UINT32.
294             *value = * (UINT32 *) &flags;
295             break;
296         }
297         case TPM_PT_STARTUP_CLEAR:
298             // TPMA_STARTUP_CLEAR
299         {
300             TPMA_STARTUP_CLEAR      flags = {0};
301             if(g_phEnable)
302                 flags.phEnable = SET;
303             if(gc.shEnable)
304                 flags.shEnable = SET;
305             if(gc.ehEnable)
306                 flags.ehEnable = SET;
307             if(gc.phEnableNV)
308                 flags.phEnableNV = SET;
309             if(g_prevOrderlyState != SHUTDOWN_NONE)
```

```
310                 flags.orderly = SET;
311
312             // Note: Different compilers may require a different method to cast
313             // a bit field structure to a UINT32.
314             *value = * (UINT32 *) &flags;
315             break;
316         }
317         case TPM_PT_HR_NV_INDEX:
318             // number of NV indexes currently defined
319             *value = NvCapGetIndexNumber();
320             break;
321         case TPM_PT_HR_LOADED:
322             // number of authorization sessions currently loaded into TPM
323             // RAM
324             *value = SessionCapGetLoadedNumber();
325             break;
326         case TPM_PT_HR_LOADED_AVAIL:
327             // number of additional authorization sessions, of any type,
328             // that could be loaded into TPM RAM
329             *value = SessionCapGetLoadedAvail();
330             break;
331         case TPM_PT_HR_ACTIVE:
332             // number of active authorization sessions currently being
333             // tracked by the TPM
334             *value = SessionCapGetActiveNumber();
335             break;
336         case TPM_PT_HR_ACTIVE_AVAIL:
337             // number of additional authorization sessions, of any type,
338             // that could be created
339             *value = SessionCapGetActiveAvail();
340             break;
341         case TPM_PT_HR_TRANSIENT_AVAIL:
342             // estimate of the number of additional transient objects that
343             // could be loaded into TPM RAM
344             *value = ObjectCapGetTransientAvail();
345             break;
346         case TPM_PT_HR_PERSISTENT:
347             // number of persistent objects currently loaded into TPM
348             // NV memory
349             *value = NvCapGetPersistentNumber();
350             break;
351         case TPM_PT_HR_PERSISTENT_AVAIL:
352             // number of additional persistent objects that could be loaded
353             // into NV memory
354             *value = NvCapGetPersistentAvail();
355             break;
356         case TPM_PT_NV_COUNTERS:
357             // number of defined NV indexes that have NV TPMA_NV_COUNTER
358             // attribute SET
359             *value = NvCapGetCounterNumber();
360             break;
361         case TPM_PT_NV_COUNTERS_AVAIL:
362             // number of additional NV indexes that can be defined with their
363             // TPMA_NV_COUNTER attribute SET
364             *value = NvCapGetCounterAvail();
365             break;
366         case TPM_PT_ALGORITHM_SET:
367             // region code for the TPM
368             *value = gp.algorithmSet;
369             break;
370
371         case TPM_PT_LOADED_CURVES:
372     #ifdef TPM_ALG_ECC
373             // number of loaded ECC curves
374             *value = CryptCapGetEccCurveNumber();
375     #else // TPM_ALG_ECC
```

```
376             *value = 0;
377     #endif // TPM_ALG_ECC
378             break;
379
380         case TPM_PT_LOCKOUT_COUNTER:
381             // current value of the lockout counter
382             *value = gp.failedTries;
383             break;
384         case TPM_PT_MAX_AUTH_FAIL:
385             // number of authorization failures before DA lockout is invoked
386             *value = gp.maxTries;
387             break;
388         case TPM_PT_LOCKOUT_INTERVAL:
389             // number of seconds before the value reported by
390             // TPM_PT_LOCKOUT_COUNTER is decremented
391             *value = gp.recoveryTime;
392             break;
393         case TPM_PT_LOCKOUT_RECOVERY:
394             // number of seconds after a lockoutAuth failure before use of
395             // lockoutAuth may be attempted again
396             *value = gp.lockoutRecovery;
397             break;
398         case TPM_PT_AUDIT_COUNTER_0:
399             // high-order 32 bits of the command audit counter
400             *value = (UINT32) (gp.auditCounter >> 32);
401             break;
402         case TPM_PT_AUDIT_COUNTER_1:
403             // low-order 32 bits of the command audit counter
404             *value = (UINT32) (gp.auditCounter);
405             break;
406         default:
407             // property is not defined
408             return FALSE;
409             break;
410     }
411
412     return TRUE;
413 }
```

9.14.3.2 TPMCapGetProperties()

This function is used to get the TPM_PT values. The search of properties will start at *property* and continue until *propertyList* has as many values as will fit, or the last property has been reported, or the list has as many values as requested in *count*.

Return Value	Meaning
YES	more properties are available
NO	no more properties to be reported

```
414 TPMI_YES_NO
415 TPMCapGetProperties(
416     TPM_PT                      property,       // IN: the starting TPM property
417     UINT32                      count,          // IN: maximum number of returned
418                                                 //     propertie
419     TPML_TAGGED_TPM_PROPERTY    *propertyList   // OUT: property list
420     )
421 {
422     TPMI_YES_NO     more = NO;
423     UINT32          i;
424
425     // initialize output property list
426     propertyList->count = 0;
```

```
427
428         // maximum count of properties we may return is MAX_PCR_PROPERTIES
429         if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;
430
431         // If property is less than PT_FIXED, start from PT_FIXED.
432         if(property < PT_FIXED) property = PT_FIXED;
433
434         // Scan through the TPM properties of the requested group.
435         // The size of TPM property group is PT_GROUP * 2 for fix and
436         // variable groups.
437         for(i = property; i <= PT_FIXED + PT_GROUP * 2; i++)
438         {
439             UINT32          value;
440             if(TPMPropertyIsDefined((TPM_PT) i, &value))
441             {
442                 if(propertyList->count < count)
443                 {
444
445                     // If the list is not full, add this property
446                     propertyList->tpmProperty[propertyList->count].property =
447                         (TPM_PT) i;
448                     propertyList->tpmProperty[propertyList->count].value = value;
449                     propertyList->count++;
450                 }
451                 else
452                 {
453                     // If the return list is full but there are more properties
454                     // available, set the indication and exit the loop.
455                     more = YES;
456                     break;
457                 }
458             }
459         }
460         return more;
461     }
```

9.15 TpmFail.c

9.15.1 Includes, Defines, and Types

```
1   #define     TPM_FAIL_C
2   #include    "InternalRoutines.h"
3   #include    <assert.h>
```

On MS C compiler, can save the alignment state and set the alignment to 1 for the duration of the TPM_Types.h include. This will avoid a lot of alignment warnings from the compiler for the unaligned structures. The alignment of the structures is not important as this function does not use any of the structures in TPM_Types.h and only include it for the #defines of the capabilities, properties, and command code values.

```
4   #pragma pack(push, 1)
5   #include "TPM_Types.h"
6   #pragma pack (pop)
7   #include "swap.h"
```

9.15.2 Typedefs

These defines are used primarily for sizing of the local response buffer.

```
8   #pragma pack(push,1)
9   typedef struct {
```

```
10      TPM_ST          tag;
11      UINT32          size;
12      TPM_RC          code;
13  } HEADER;
14  typedef struct {
15      UINT16      size;
16      struct {
17          UINT32      function;
18          UINT32      line;
19          UINT32      code;
20      } values;
21      TPM_RC      returnCode;
22  } GET_TEST_RESULT_PARAMETERS;
23  typedef struct {
24      TPMI_YES_NO                 moreData;
25      TPM_CAP                     capability; // Always TPM_CAP_TPM_PROPERTIES
26      TPML_TAGGED_TPM_PROPERTY    tpmProperty; // a single tagged property
27  } GET_CAPABILITY_PARAMETERS;
28  typedef struct {
29      HEADER header;
30      GET_TEST_RESULT_PARAMETERS  getTestResult;
31  } TEST_RESPONSE;
32  typedef struct {
33      HEADER header;
34      GET_CAPABILITY_PARAMETERS   getCap;
35  } CAPABILITY_RESPONSE;
36  typedef union {
37      TEST_RESPONSE           test;
38      CAPABILITY_RESPONSE     cap;
39  } RESPONSES;
40  #pragma pack(pop)
```

Buffer to hold the responses. This may be a little larger than required due to padding that a compiler might add.

NOTE: This is not in Global.c because of the specialized data definitions above. Since the data contained in this structure is not relevant outside of the execution of a single command (when the TPM is in failure mode. There is no compelling reason to move all the typedefs to Global.h and this structure to Global.c.

```
41  #ifndef __IGNORE_STATE__ // Don't define this value
42  static BYTE response[sizeof(RESPONSES)];
43  #endif
```

9.15.3 Local Functions

9.15.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```
44  static INT32
45  MarshalUint16(
46      UINT16          integer,
47      BYTE            **buffer
48      )
49  {
50      return UINT16_Marshal(&integer, buffer, NULL);
51  }
```

9.15.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```
52  static INT32
53  MarshalUint32(
54      UINT32           integer,
55      BYTE            **buffer
56      )
57  {
58      return UINT32_Marshal(&integer, buffer, NULL);
59  }
```

9.15.3.3 UnmarshalHeader()

Funtion to unmarshal the 10-byte command header.

```
60  static BOOL
61  UnmarshalHeader(
62      HEADER          *header,
63      BYTE            **buffer,
64      INT32           *size
65      )
66  {
67      UINT32 usize;
68      TPM_RC ucode;
69      if(     UINT16_Unmarshal(&header->tag, buffer, size) != TPM_RC_SUCCESS
70          ||  UINT32_Unmarshal(&usize, buffer, size) != TPM_RC_SUCCESS
71          ||  UINT32_Unmarshal(&ucode, buffer, size) != TPM_RC_SUCCESS
72          )
73          return FALSE;
74      header->size = usize;
75      header->code = ucode;
76      return TRUE;
77  }
```

9.15.4 Public Functions

9.15.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```
78  LIB_EXPORT void
79  SetForceFailureMode(
80      void
81      )
82  {
83      g_forceFailureMode = TRUE;
84      return;
85  }
```

9.15.4.2 TpmFail()

This function is called by TPM.lib when a failure occurs. It will set up the failure values to be returned on TPM2_GetTestResult().

```
86  void
87  TpmFail(
88      const char                  *function,
89      int line,           int      code
90      )
91  {
92      // Save the values that indicate where the error occurred.
93      // On a 64-bit machine, this may truncate the address of the string
```

```
94      // of the function name where the error occurred.
95      s_failFunction = *(UINT32*)&function;
96      s_failLine = line;
97      s_failCode = code;
98
99      // if asserts are enabled, then do an assert unless the failure mode code
100     // is being tested
101     assert(g_forceFailureMode);
102
103     // Clear this flag
104     g_forceFailureMode = FALSE;
105
106     // Jump to the failure mode code.
107     // Note: only get here if asserts are off or if we are testing failure mode
108     longjmp(&g_jumpBuffer[0], 1);
109 }
```

9.15.5 TpmFailureMode

This function is called by the interface code when the platform is in failure mode.

```
110 void
111 TpmFailureMode (
112     unsigned int     inRequestSize,     // IN: command buffer size
113     unsigned char   *inRequest,         // IN: command buffer
114     unsigned int    *outResponseSize,   // OUT: response buffer size
115     unsigned char   **outResponse       // OUT: response buffer
116     )
117 {
118     BYTE            *buffer;
119     UINT32           marshalSize;
120     UINT32           capability;
121     HEADER           header;    // unmarshaled command header
122     UINT32           pt;    // unmarshaled property type
123     UINT32           count; // unmarshaled property count
124
125     // If there is no command buffer, then just return TPM_RC_FAILURE
126     if(inRequestSize == 0 || inRequest == NULL)
127         goto FailureModeReturn;
128
129     // If the header is not correct for TPM2_GetCapability() or
130     // TPM2_GetTestResult() then just return the in failure mode response;
131     buffer = inRequest;
132     if(!UnmarshalHeader(&header, &inRequest, (INT32 *)&inRequestSize))
133         goto FailureModeReturn;
134     if(   header.tag  != TPM_ST_NO_SESSIONS
135        || header.size < 10)
136        goto FailureModeReturn;
137
138     switch (header.code) {
139     case TPM_CC_GetTestResult:
140
141         // make sure that the command size is correct
142         if(header.size != 10)
143             goto FailureModeReturn;
144         buffer = &response[10];
145         marshalSize =  MarshalUint16(3 * sizeof(UINT32), &buffer);
146         marshalSize += MarshalUint32(s_failFunction, &buffer);
147         marshalSize += MarshalUint32(s_failLine, &buffer);
148         marshalSize += MarshalUint32(s_failCode, &buffer);
149         if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
150             marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
151         else
152             marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
```

```
153             break;
154
155         case TPM_CC_GetCapability:
156             // make sure that the size of the command is exactly the size
157             // returned for the capability, property, and count
158             if(     header.size!= (10 + (3 * sizeof(UINT32)))
159                     // also verify that this is requesting TPM properties
160                 ||      (UINT32_Unmarshal(&capability, &inRequest,
161                                           (INT32 *)&inRequestSize)
162                     !=  TPM_RC_SUCCESS)
163                 || (capability != TPM_CAP_TPM_PROPERTIES)
164                 ||      (UINT32_Unmarshal(&pt, &inRequest, (INT32 *)&inRequestSize)
165                     != TPM_RC_SUCCESS)
166                 ||      (UINT32_Unmarshal(&count, &inRequest, (INT32 *)&inRequestSize)
167                     != TPM_RC_SUCCESS)
168                 )
169
170                 goto FailureModeReturn;
171
172             // If in failure mode because of an unrecoverable read error, and the
173             // property is 0 and the count is 0, then this is an indication to
174             // re-manufacture the TPM. Do the re-manufacture but stay in failure
175             // mode until the TPM is reset.
176             // Note: this behavior is not required by the specification and it is
177             // OK to leave the TPM permanently bricked due to an unrecoverable NV
178             // error.
179             if( count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
180             {
181                 g_manufactured = FALSE;
182                 TPM_Manufacture(0);
183             }
184
185             if(count > 0)
186                 count = 1;
187             else if(pt > TPM_PT_FIRMWARE_VERSION_2)
188                 count = 0;
189             if(pt < TPM_PT_MANUFACTURER)
190                 pt = TPM_PT_MANUFACTURER;
191
192             // set up for return
193             buffer = &response[10];
194             // if the request was for a PT less than the last one
195             // then we indicate more, otherwise, not.
196             if(pt < TPM_PT_FIRMWARE_VERSION_2)
197                 *buffer++ = YES;
198             else
199                 *buffer++ = NO;
200
201             marshalSize = 1;
202
203             // indicate the capability type
204             marshalSize += MarshalUint32(capability, &buffer);
205             // indicate the number of values that are being returned (0 or 1)
206             marshalSize += MarshalUint32(count, &buffer);
207             // indicate the property
208             marshalSize += MarshalUint32(pt, &buffer);
209
210             if(count > 0)
211                 switch (pt) {
212                 case TPM_PT_MANUFACTURER:
213                 // the vendor ID unique to each TPM manufacturer
214 #ifdef  MANUFACTURER
215                 pt = *(UINT32*)MANUFACTURER;
216 #else
217                 pt = 0;
218 #endif
```

```
219             break;
220         case TPM_PT_VENDOR_STRING_1:
221             // the first four characters of the vendor ID string
222 #ifdef  VENDOR_STRING_1
223             pt = *(UINT32*)VENDOR_STRING_1;
224 #else
225             pt = 0;
226 #endif
227             break;
228         case TPM_PT_VENDOR_STRING_2:
229             // the second four characters of the vendor ID string
230 #ifdef  VENDOR_STRING_2
231             pt = *(UINT32*)VENDOR_STRING_2;
232 #else
233             pt = 0;
234 #endif
235             break;
236         case TPM_PT_VENDOR_STRING_3:
237             // the third four characters of the vendor ID string
238 #ifdef  VENDOR_STRING_3
239             pt = *(UINT32*)VENDOR_STRING_3;
240 #else
241             pt = 0;
242 #endif
243             break;
244         case TPM_PT_VENDOR_STRING_4:
245             // the fourth four characters of the vendor ID string
246 #ifdef  VENDOR_STRING_4
247             pt = *(UINT32*)VENDOR_STRING_4;
248 #else
249             pt = 0;
250 #endif
251
252             break;
253         case TPM_PT_VENDOR_TPM_TYPE:
254             // vendor-defined value indicating the TPM model
255             // We just make up a number here
256             pt = 1;
257             break;
258         case TPM_PT_FIRMWARE_VERSION_1:
259             // the more significant 32-bits of a vendor-specific value
260             // indicating the version of the firmware
261 #ifdef  FIRMWARE_V1
262             pt = FIRMWARE_V1;
263 #else
264             pt = 0;
265 #endif
266             break;
267         default: // TPM_PT_FIRMWARE_VERSION_2:
268             // the less significant 32-bits of a vendor-specific value
269             // indicating the version of the firmware
270 #ifdef  FIRMWARE_V2
271             pt = FIRMWARE_V2;
272 #else
273             pt = 0;
274 #endif
275             break;
276         }
277         marshalSize += MarshalUint32(pt, &buffer);
278         break;
279     default: // default for switch (cc)
280         goto FailureModeReturn;
281     }
282     // Now do the header
283     buffer = response;
284     marshalSize = marshalSize + 10; // Add the header size to the
```

```
285                                     // stuff already marshaled
286         MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
287         MarshalUint32(marshalSize, &buffer);  // responseSize
288         MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
289
290         *outResponseSize = marshalSize;
291         *outResponse = (unsigned char *)&response;
292         return;
293
294     FailureModeReturn:
295
296         buffer = response;
297
298         marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
299         marshalSize += MarshalUint32(10, &buffer);
300         marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
301
302         *outResponseSize = marshalSize;
303         *outResponse = (unsigned char *)response;
304         return;
305     }
```

10 Cryptographic Functions

10.1 Introduction

The files in this section provide cryptographic support for the other functions in the TPM and the interface to the Crypto Engine.

10.2 CryptUtil.c

10.2.1 Includes

```
#include     "TPM_Types.h"
#include     "CryptoEngine.h"     // types shared by CryptUtil and CryptoEngine.
                                  // Includes the function prototypes for the
                                  // CryptoEngine functions.
#include     "Global.h"
#include     "InternalRoutines.h"
#include     "MemoryLib_fp.h"
//#include     "CryptSelfTest_fp.h"
```

10.2.2 TranslateCryptErrors()

This function converts errors from the cryptographic library into TPM_RC_VALUES.

Error Returns	Meaning
TPM_RC_VALUE	CRYPT_FAIL
TPM_RC_NO_RESULT	CRYPT_NO_RESULT
TPM_RC_SCHEME	CRYPT_SCHEME
TPM_RC_VALUE	CRYPT_PARAMETER
TPM_RC_SIZE	CRYPT_UNDERFLOW
TPM_RC_ECC_POINT	CRYPT_POINT
TPM_RC_CANCELLED	CRYPT_CANCEL

```
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_VALUE;
    case CRYPT_UNDERFLOW:
        return TPM_RC_SIZE;
    case CRYPT_POINT:
        return TPM_RC_ECC_POINT;
    case CRYPT_CANCEL:
```

```
31          return TPM_RC_CANCELED;
32      default: // Other unknown warnings
33          return TPM_RC_FAILURE;
34      }
35  }
```

10.2.3 Random Number Generation Functions

```
36  #ifdef TPM_ALG_NULL //%
37  #ifdef _DRBG_STATE_SAVE //%
```

10.2.3.1 CryptDrbgGetPutState()

Read or write the current state from the DRBG in the *cryptoEngine*.

```
38  void
39  CryptDrbgGetPutState(
40      GET_PUT          direction      // IN: Get from or put to DRBG
41      )
42  {
43      _cpri__DrbgGetPutState(direction,
44                             sizeof(go.drbgState),
45                             (BYTE *)&go.drbgState);
46  }
47  #else   //% 00
48  //%#define CryptDrbgGetPutState(ignored)  // If not doing state save, turn this
49  //%                                       // into a null macro
50  #endif  //%
```

10.2.3.2 CryptStirRandom()

Stir random entropy

```
51  void
52  CryptStirRandom(
53      UINT32           entropySize,   // IN: size of entropy buffer
54      BYTE            *buffer         // IN: entropy buffer
55      )
56  {
57      // RNG self testing code may be inserted here
58
59      // Call crypto engine random number stirring function
60      _cpri__StirRandom(entropySize, buffer);
61
62      return;
63  }
```

10.2.3.3 CryptGenerateRandom()

This is the interface to _cpri__GenerateRandom().

```
64  UINT16
65  CryptGenerateRandom(
66      UINT16           randomSize,    // IN: size of random number
67      BYTE            *buffer         // OUT: buffer of random number
68      )
69  {
70      UINT16              result;
71      pAssert(randomSize <= MAX_RSA_KEY_BYTES || randomSize <= PRIMARY_SEED_SIZE);
72      if(randomSize == 0)
```

```
73          return 0;
74
75      // Call crypto engine random number generation
76      result = _cpri__GenerateRandom(randomSize, buffer);
77      if(result != randomSize)
78          FAIL(FATAL_ERROR_INTERNAL);
79
80      return result;
81  }
82  #endif //TPM_ALG_NULL //%
```

10.2.4 Hash/HMAC Functions

10.2.4.1 CryptGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```
83  #ifdef TPM_ALG_KEYEDHASH        //% 1
84  TPM_ALG_ID
85  CryptGetContextAlg(
86      void            *state          // IN: the context to check
87      )
88  {
89      HASH_STATE  *context = (HASH_STATE *)state;
90      return _cpri__GetContextAlg(&context->state);
91  }
```

10.2.4.2 CryptStartHash()

This function starts a hash and return the size, in bytes, of the digest.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
92  UINT16
93  CryptStartHash(
94      TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
95      HASH_STATE      *hashState      // OUT: the state of hash stack. It will be used
96                                      //      in hash update and completion
97      )
98  {
99      CRYPT_RESULT       retVal = 0;
100
101     pAssert(hashState != NULL);
102
103     TEST_HASH(hashAlg);
104
105     hashState->type = HASH_STATE_EMPTY;
106
107     // Call crypto engine start hash function
108     if((retVal = _cpri__StartHash(hashAlg, FALSE, &hashState->state)) > 0)
109         hashState->type = HASH_STATE_HASH;
110
111     return retVal;
112 }
```

10.2.4.3 CryptStartHashSequence()

Start a hash stack for a sequence object and return the size, in bytes, of the digest. This call uses the form of the hash state that requires context save and restored.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
113  UINT16
114  CryptStartHashSequence(
115      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
116      HASH_STATE      *hashState       // OUT: the state of hash stack. It will be used
117                                       //      in hash update and completion
118      )
119  {
120      CRYPT_RESULT   retVal = 0;
121
122      pAssert(hashState != NULL);
123
124      TEST_HASH(hashAlg);
125
126      hashState->type = HASH_STATE_EMPTY;
127
128      // Call crypto engine start hash function
129      if((retVal = _cpri__StartHash(hashAlg, TRUE, &hashState->state)) > 0)
130          hashState->type = HASH_STATE_HASH;
131
132      return retVal;
133
134  }
```

10.2.4.4 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

```
135  UINT16
136  CryptStartHMAC(
137      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
138      UINT16           keySize,        // IN: the size of HMAC key in byte
139      BYTE            *key,            // IN: HMAC key
140      HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
141                                       //      in HMAC update and completion
142      )
143  {
144      HASH_STATE      *hashState = (HASH_STATE *)hmacState;
145      CRYPT_RESULT     retVal;
146
147      // This has to come before the pAssert in case we all calling this function
148      // during testing. If so, the first instance will have no arguments but the
149      // hash algorithm. The call from the test routine will have arguments. When
150      // the second call is done, then we return to the test dispatcher.
151      TEST_HASH(hashAlg);
```

```
152
153         pAssert(hashState != NULL);
154
155         hashState->type = HASH_STATE_EMPTY;
156
157         if((retVal =  _cpri__StartHMAC(hashAlg, FALSE, &hashState->state, keySize, key,
158                                        &hmacState->hmacKey.b)) > 0)
159             hashState->type = HASH_STATE_HMAC;
160
161         return retVal;
162     }
```

10.2.4.5 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

```
163     UINT16
164     CryptStartHMACSequence(
165         TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
166         UINT16           keySize,        // IN: the size of HMAC key in byte
167         BYTE            *key,            // IN: HMAC key
168         HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
169                                          //      in HMAC update and completion
170         )
171     {
172         HASH_STATE      *hashState = (HASH_STATE *)hmacState;
173         CRYPT_RESULT    retVal;
174
175         TEST_HASH(hashAlg);
176
177         hashState->type = HASH_STATE_EMPTY;
178
179         if((retVal =  _cpri__StartHMAC(hashAlg, TRUE, &hashState->state,
180                                        keySize, key, &hmacState->hmacKey.b)) > 0)
181             hashState->type = HASH_STATE_HMAC;
182
183         return retVal;
184     }
```

10.2.4.6 CryptStartHMAC2B()

This function starts an HMAC and returns the size of the digest that will be produced.

This function is provided to support the most common use of starting an HMAC with a TPM2B key.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
185  LIB_EXPORT UINT16
186  CryptStartHMAC2B(
187      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
188      TPM2B           *key,            // IN: HMAC key
189      HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
190                                       //      in HMAC update and completion
191      )
192  {
193      return CryptStartHMAC(hashAlg, key->size, key->buffer, hmacState);
194  }
```

10.2.4.7 CryptStartHMACSequence2B()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

This function is provided to support the most common use of starting an HMAC with a TPM2B key.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
195  UINT16
196  CryptStartHMACSequence2B(
197      TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
198      TPM2B           *key,            // IN: HMAC key
199      HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
200                                       //      in HMAC update and completion
201      )
202  {
203      return CryptStartHMACSequence(hashAlg, key->size, key->buffer, hmacState);
204  }
```

10.2.4.8 CryptUpdateDigest()

This function updates a digest (hash or HMAC) with an array of octets.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```
205  LIB_EXPORT void
206  CryptUpdateDigest(
207      void            *digestState,    // IN: the state of hash stack
208      UINT32           dataSize,       // IN: the size of data
209      BYTE            *data            // IN: data to be hashed
210      )
211  {
212      HASH_STATE      *hashState = (HASH_STATE *)digestState;
213
214      pAssert(digestState != NULL);
215
216      if(hashState->type != HASH_STATE_EMPTY && data != NULL && dataSize != 0)
217      {
```

```
218             // Call crypto engine update hash function
219             _cpri__UpdateHash(&hashState->state, dataSize, data);
220         }
221         return;
222     }
```

10.2.4.9 CryptUpdateDigest2B()

This function updates a digest (hash or HMAC) with a TPM2B.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```
223     LIB_EXPORT void
224     CryptUpdateDigest2B(
225         void            *digestState,   // IN: the digest state
226         TPM2B           *bIn            // IN: 2B containing the data
227         )
228     {
229         // Only compute the digest if a pointer to the 2B is provided.
230         // In CryptUpdateDigest(), if size is zero or buffer is NULL, then no change
231         // to the digest occurs. This function should not provide a buffer if bIn is
232         // not provided.
233         if(bIn != NULL)
234             CryptUpdateDigest(digestState, bIn->size, bIn->buffer);
235         return;
236     }
```

10.2.4.10 CryptUpdateDigestInt()

This function is used to include an integer value to a hash stack. The function marshals the integer into its canonical form before calling CryptUpdateHash().

```
237     LIB_EXPORT void
238     CryptUpdateDigestInt(
239         void            *state,         // IN: the state of hash stack
240         UINT32           intSize,       // IN: the size of 'intValue' in byte
241         void            *intValue       // IN: integer value to be hashed
242         )
243     {
244
245     #if BIG_ENDIAN_TPM == YES
246         pAssert(    intValue != NULL && (intSize == 1 || intSize == 2
247                 ||  intSize == 4 || intSize == 8));
248         CryptUpdateHash(state, inSize, (BYTE *)intValue);
249     #else
250
251         BYTE        marshalBuffer[8];
252         // Point to the big end of an little-endian value
253         BYTE        *p = &((BYTE *)intValue)[intSize - 1];
254         // Point to the big end of an big-endian value
255         BYTE        *q = marshalBuffer;
256
257         pAssert(intValue != NULL);
258         switch (intSize)
259         {
260         case 8:
261             *q++ = *p--;
262             *q++ = *p--;
263             *q++ = *p--;
264             *q++ = *p--;
265         case 4:
266             *q++ = *p--;
```

```
267             *q++ = *p--;
268         case 2:
269             *q++ = *p--;
270         case 1:
271             *q = *p;
272             // Call update the hash
273             CryptUpdateDigest(state, intSize, marshalBuffer);
274             break;
275         default:
276             FAIL(0);
277         }
278
279 #endif
280     return;
281 }
```

10.2.4.11 CryptCompleteHash()

This function completes a hash sequence and returns the digest.

This function can be called to complete either an HMAC or hash sequence. The state type determines if the context type is a hash or HMAC. If an HMAC, then the call is forwarded to CryptCompleteHash().

If **digestSize** is smaller than the digest size of hash/HMAC algorithm, the most significant bytes of required size will be returned

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
282 LIB_EXPORT UINT16
283 CryptCompleteHash(
284     void            *state,         // IN: the state of hash stack
285     UINT16           digestSize,    // IN: size of digest buffer
286     BYTE            *digest         // OUT: hash digest
287     )
288 {
289     HASH_STATE      *hashState = (HASH_STATE *)state;     // local value
290
291     // If the session type is HMAC, then could forward this to
292     // the HMAC processing and not cause an error. However, if no
293     // function calls this routine to forward it, then we can't get
294     // test coverage. The decision is to assert if this is called with
295     // the type == HMAC and fix anything that makes the wrong call.
296     pAssert(hashState->type == HASH_STATE_HASH);
297
298     // Set the state to empty so that it doesn't get used again
299     hashState->type = HASH_STATE_EMPTY;
300
301     // Call crypto engine complete hash function
302     return      _cpri__CompleteHash(&hashState->state, digestSize, digest);
303 }
```

10.2.4.12 CryptCompleteHash2B()

This function is the same as CypteCompleteHash() but the digest is placed in a TPM2B. This is the most common use and this is provided for specification clarity. 'digest.size' should be set to indicate the number of bytes to place in the buffer

Return Value	Meaning
>=0	the number of bytes placed in 'digest.buffer'

```
304   LIB_EXPORT UINT16
305   CryptCompleteHash2B(
306       void            *state,         // IN: the state of hash stack
307       TPM2B           *digest         // IN: the size of the buffer Out: requested
308                                       //     number of byte
309       )
310   {
311       UINT16              retVal = 0;
312
313       if(digest != NULL)
314           retVal = CryptCompleteHash(state, digest->size, digest->buffer);
315
316       return retVal;
317   }
```

10.2.4.13 CryptHashBlock()

Hash a block of data and return the results. If the digest is larger than *retSize*, it is truncated and with the least significant octets dropped.

Return Value	Meaning
>=0	the number of bytes placed in *ret*

```
318   LIB_EXPORT UINT16
319   CryptHashBlock(
320       TPM_ALG_ID       algId,         // IN: the hash algorithm to use
321       UINT16           blockSize,     // IN: size of the data block
322       BYTE            *block,         // IN: address of the block to hash
323       UINT16           retSize,       // IN: size of the return buffer
324       BYTE            *ret            // OUT: address of the buffer
325       )
326   {
327       TEST_HASH(algId);
328
329       return _cpri__HashBlock(algId, blockSize, block, retSize, ret);
330   }
```

10.2.4.14 CryptCompleteHMAC()

This function completes a HMAC sequence and returns the digest. If *digestSize* is smaller than the digest size of the HMAC algorithm, the most significant bytes of required size will be returned.

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
331   LIB_EXPORT UINT16
332   CryptCompleteHMAC(
333       HMAC_STATE      *hmacState,     // IN: the state of HMAC stack
334       UINT32           digestSize,    // IN: size of digest buffer
335       BYTE            *digest         // OUT: HMAC digest
336       )
337   {
338       HASH_STATE      *hashState;
339
340       pAssert(hmacState != NULL);
```

```
341         hashState = &hmacState->hashState;
342
343         pAssert(hashState->type == HASH_STATE_HMAC);
344
345         hashState->type = HASH_STATE_EMPTY;
346
347         return _cpri__CompleteHMAC(&hashState->state, &hmacState->hmacKey.b,
348                                    digestSize, digest);
349
350     }
```

10.2.4.15 CryptCompleteHMAC2B()

This function is the same as CryptCompleteHMAC() but the HMAC result is returned in a TPM2B which is the most common use.

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
351     LIB_EXPORT UINT16
352     CryptCompleteHMAC2B(
353         HMAC_STATE      *hmacState,     // IN: the state of HMAC stack
354         TPM2B           *digest         // OUT: HMAC
355         )
356     {
357         UINT16                retVal = 0;
358         if(digest != NULL)
359             retVal = CryptCompleteHMAC(hmacState, digest->size, digest->buffer);
360         return retVal;
361     }
```

10.2.4.16 CryptHashStateImportExport()

This function is used to prepare a hash state context for LIB_EXPORT or to import it into the internal format. It is used by TPM2_ContextSave() and TPM2_ContextLoad() via SequenceDataImportExport(). This is just a pass-through function to the crypto library.

```
362     void
363     CryptHashStateImportExport(
364         HASH_STATE      *internalFmt,   // IN: state to LIB_EXPORT
365         HASH_STATE      *externalFmt,   // OUT: exported state
366         IMPORT_EXPORT    direction
367         )
368     {
369         _cpri__ImportExportHashState(&internalFmt->state,
370                                      (EXPORT_HASH_STATE *)&externalFmt->state,
371                                      direction);
372     }
```

10.2.4.17 CryptGetHashDigestSize()

This function returns the digest size in bytes for a hash algorithm.

Return Value	Meaning
0	digest size for TPM_ALG_NULL
> 0	digest size

```
373     LIB_EXPORT UINT16
```

```
374 CryptGetHashDigestSize(
375     TPM_ALG_ID        hashAlg         // IN: hash algorithm
376     )
377 {
378     return _cpri__GetDigestSize(hashAlg);
379 }
```

10.2.4.18 CryptGetHashBlockSize()

Get the digest size in byte of a hash algorithm.

Return Value	Meaning
0	block size for TPM_ALG_NULL
> 0	block size

```
380 LIB_EXPORT UINT16
381 CryptGetHashBlockSize(
382     TPM_ALG_ID        hash            // IN: hash algorithm to look up
383     )
384 {
385     return _cpri__GetHashBlockSize(hash);
386 }
```

10.2.4.19 CryptGetHashAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and an *index* value of 2 will return the last implemented hash. All other index values will return TPM_ALG_NULL.

Return Value	Meaning
TPM_ALG_xxx()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
387 LIB_EXPORT TPM_ALG_ID
388 CryptGetHashAlgByIndex(
389     UINT32            index           // IN: the index
390     )
391 {
392     return _cpri__GetHashAlgByIndex(index);
393 }
```

10.2.4.20 CryptSignHMAC()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

Error Returns	Meaning

```
394 static TPM_RC
395 CryptSignHMAC(
396     OBJECT              *signKey,       // IN: HMAC key sign the hash
397     TPMT_SIG_SCHEME     *scheme,        // IN: signing scheme
398     TPM2B_DIGEST        *hashData,      // IN: hash to be signed
399     TPMT_SIGNATURE      *signature      // OUT: signature
400     )
401 {
```

```
402         HMAC_STATE       hmacState;
403         UINT32           digestSize;
404
405         // HMAC algorithm self testing code may be inserted here
406
407         digestSize = CryptStartHMAC2B(scheme->details.hmac.hashAlg,
408                                       &signKey->sensitive.sensitive.bits.b,
409                                       &hmacState);
410
411         // The hash algorithm must be a valid one.
412         pAssert(digestSize > 0);
413
414         CryptUpdateDigest2B(&hmacState, &hashData->b);
415
416         CryptCompleteHMAC(&hmacState, digestSize,
417                           (BYTE *) &signature->signature.hmac.digest);
418
419         // Set HMAC algorithm
420         signature->signature.hmac.hashAlg = scheme->details.hmac.hashAlg;
421
422         return TPM_RC_SUCCESS;
423     }
```

10.2.4.21 CryptHMACVerifySignature()

This function will verify a signature signed by a HMAC key.

Error Returns	Meaning
TPM_RC_SIGNATURE	if invalid input or signature is not genuine

```
424     static TPM_RC
425     CryptHMACVerifySignature(
426         OBJECT          *signKey,       // IN: HMAC key signed the hash
427         TPM2B_DIGEST    *hashData,      // IN: digest being verified
428         TPMT_SIGNATURE  *signature      // IN: signature to be verified
429         )
430     {
431         HMAC_STATE          hmacState;
432         TPM2B_DIGEST        digestToCompare;
433
434         digestToCompare.t.size = CryptStartHMAC2B(signature->signature.hmac.hashAlg,
435                                   &signKey->sensitive.sensitive.bits.b, &hmacState);
436
437         CryptUpdateDigest2B(&hmacState, &hashData->b);
438
439         CryptCompleteHMAC2B(&hmacState, &digestToCompare.b);
440
441         // Compare digest
442         if(MemoryEqual(digestToCompare.t.buffer,
443                        (BYTE *) &signature->signature.hmac.digest,
444                        digestToCompare.t.size))
445             return TPM_RC_SUCCESS;
446         else
447             return TPM_RC_SIGNATURE;
448
449     }
```

10.2.4.22 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Error Returns	Meaning
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```
450  static TPM_RC
451  CryptGenerateKeyedHash(
452      TPMT_PUBLIC             *publicArea,        // IN/OUT: the public area template
453                                                  //     for the new key.
454      TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
455      TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
456      TPM_ALG_ID               kdfHashAlg,        // IN: algorithm for the KDF
457      TPM2B_SEED              *seed,              // IN: the seed
458      TPM2B_NAME              *name               // IN: name of the object
459      )
460  {
461      TPMT_KEYEDHASH_SCHEME   *scheme;
462      TPM_ALG_ID               hashAlg;
463      UINT16                   hashBlockSize;
464
465      scheme = &publicArea->parameters.keyedHashDetail.scheme;
466
467      pAssert(publicArea->type == TPM_ALG_KEYEDHASH);
468
469      // Pick the limiting hash algorithm
470      if(scheme->scheme == TPM_ALG_NULL)
471          hashAlg = publicArea->nameAlg;
472      else if(scheme->scheme == TPM_ALG_XOR)
473          hashAlg = scheme->details.xor.hashAlg;
474      else
475          hashAlg = scheme->details.hmac.hashAlg;
476      hashBlockSize =  CryptGetHashBlockSize(hashAlg);
477
478      // if this is a signing or a decryption key, then then the limit
479      // for the data size is the block size of the hash. This limit
480      // is set because larger values have lower entropy because of the
481      // HMAC function.
482      if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR)
483      {
484          if(    (   publicArea->objectAttributes.decrypt
485                  ||  publicArea->objectAttributes.sign)
486              && sensitiveCreate->data.t.size > hashBlockSize)
487
488              return TPM_RC_SIZE;
489      }
490      else
491      {
492          // If the TPM is going to generate the data, then set the size to be the
493          // size of the digest of the algorithm
494          sensitive->sensitive.sym.t.size = CryptGetHashDigestSize(hashAlg);
495          sensitiveCreate->data.t.size = 0;
496      }
497
498      // Fill in the sensitive area
499      CryptGenerateNewSymmetric(sensitiveCreate, sensitive, kdfHashAlg,
500                                seed, name);
501
502      // Create unique area in public
503      CryptComputeSymmetricUnique(publicArea->nameAlg,
504                                  sensitive, &publicArea->unique.sym);
505
506      return TPM_RC_SUCCESS;
507  }
```

10.2.4.23 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.

```
508  //%#define CryptKDFa(hashAlg, key, label, contextU, contextV,    \
509  //%                  sizeInBits, keyStream, counterInOut)       \
510  //%         TEST_HASH(hashAlg);                                 \
511  //%        _cpri__KDFa(                                         \
512  //%                    ((TPM_ALG_ID)hashAlg),                   \
513  //%                    ((TPM2B *)key),                          \
514  //%                    ((const char *)label),                   \
515  //%                    ((TPM2B *)contextU),                     \
516  //%                    ((TPM2B *)contextV),                     \
517  //%                    ((UINT32)sizeInBits),                    \
518  //%                    ((BYTE *)keyStream),                     \
519  //%                    ((UINT32 *)counterInOut),                \
520  //%                    ((BOOL) FALSE)                           \
521  //%                   )
522  //%
```

10.2.4.24 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*.

```
523  //%#define CryptKDFaOnce(hashAlg, key, label, contextU, contextV,    \
524  //%                   sizeInBits, keyStream, counterInOut)       \
525  //%         TEST_HASH(hashAlg);                                 \
526  //%        _cpri__KDFa(                                         \
527  //%                    ((TPM_ALG_ID)hashAlg),                   \
528  //%                    ((TPM2B *)key),                          \
529  //%                    ((const char *)label),                   \
530  //%                    ((TPM2B *)contextU),                     \
531  //%                    ((TPM2B *)contextV),                     \
532  //%                    ((UINT32)sizeInBits),                    \
533  //%                    ((BYTE *)keyStream),                     \
534  //%                    ((UINT32 *)counterInOut),                \
535  //%                    ((BOOL) TRUE)                            \
536  //%                   )
537  //%
```

10.2.4.25 KDFa()

This function is used by functions outside of CryptUtil() to access _cpri_KDFa().

```
538  void
539  KDFa(
540      TPM_ALG_ID       hash,          // IN: hash algorithm used in HMAC
541      TPM2B           *key,           // IN: HMAC key
542      const char      *label,         // IN: a null-terminated label for KDF
543      TPM2B           *contextU,      // IN: context U
544      TPM2B           *contextV,      // IN: context V
545      UINT32           sizeInBits,    // IN: size of generated key in bit
546      BYTE            *keyStream,     // OUT: key buffer
547      UINT32          *counterInOut   // IN/OUT: caller may provide the iteration
548                                      //     counter for incremental operations to
```

```
549                                     //     avoid large intermediate buffers.
550     )
551 {
552     CryptKDFa(hash, key, label, contextU, contextV, sizeInBits,
553               keyStream, counterInOut);
554 }
```

10.2.4.26 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().

```
555 //%#define CryptKDFe(hashAlg, Z, label, partyUInfo, partyVInfo,          \
556 //%                  sizeInBits, keyStream)                             \
557 //% TEST_HASH(hashAlg);                                                 \
558 //% _cpri__KDFe(                                                        \
559 //%             ((TPM_ALG_ID)hashAlg),                                  \
560 //%             ((TPM2B *)Z),                                           \
561 //%             ((const char *)label),                                  \
562 //%             ((TPM2B *)partyUInfo),                                  \
563 //%             ((TPM2B *)partyVInfo),                                  \
564 //%             ((UINT32)sizeInBits),                                   \
565 //%             ((BYTE *)keyStream)                                     \
566 //%            )
567 //%
568 #endif //TPM_ALG_KEYEDHASH    //% 1
```

10.2.5 RSA Functions

10.2.5.1 BuildRSA()

Function to set the cryptographic elements of an RSA key into a structure to simplify the interface to _cpri__ RSA function. This can/should be eliminated by building this structure into the object structure.

```
569 #ifdef TPM_ALG_RSA          //% 2
570 static void
571 BuildRSA(
572     OBJECT          *rsaKey,
573     RSA_KEY         *key
574     )
575 {
576     key->exponent = rsaKey->publicArea.parameters.rsaDetail.exponent;
577     if(key->exponent == 0)
578         key->exponent = RSA_DEFAULT_PUBLIC_EXPONENT;
579     key->publicKey = &rsaKey->publicArea.unique.rsa.b;
580
581     if(rsaKey->attributes.publicOnly || rsaKey->privateExponent.t.size == 0)
582         key->privateKey = NULL;
583     else
584         key->privateKey = &(rsaKey->privateExponent.b);
585 }
```

10.2.5.2 CryptTestKeyRSA()

This function provides the interface to _cpri__TestKeyRSA(). If both *p* and *q* are provided, *n* will be set to *p***q*.

If only *p* is provided, *q* is computed by *q* = *n*/*p*. If *n* mod *p* != 0, TPM_RC_BINDING is returned.

The key is validated by checking that a *d* can be found such that *e d* mod ((*p*-1)*(*q*-1)) = 1. If *d* is found that satisfies this requirement, it will be placed in *d*.

Error Returns	Meaning
TPM_RC_BINDING	the public and private portions of the key are not matched

```
586 TPM_RC
587 CryptTestKeyRSA(
588     TPM2B           *d,             // OUT: receives the private exponent
589     UINT32           e,             // IN: public exponent
590     TPM2B           *n,             // IN/OUT: public modulu
591     TPM2B           *p,             // IN: a first prime
592     TPM2B           *q              // IN: an optional second prime
593     )
594 {
595     CRYPT_RESULT    retVal;
596
597     TEST(ALG_NULL_VALUE);
598
599     pAssert(d != NULL && n != NULL && p != NULL);
600     // Set the exponent
601     if(e == 0)
602         e = RSA_DEFAULT_PUBLIC_EXPONENT;
603     // CRYPT_PARAMETER
604     retVal = _cpri__TestKeyRSA(d, e, n, p, q);
605     if(retVal == CRYPT_SUCCESS)
606         return TPM_RC_SUCCESS;
607     else
608         return TPM_RC_BINDING;  // convert CRYPT_PARAMETER
609 }
```

10.2.5.3 CryptGenerateKeyRSA()

This function is called to generate an RSA key from a provided seed. It calls _cpri__GenerateKeyRSA() to perform the computations. The implementation is vendor specific.

Error Returns	Meaning
TPM_RC_RANGE	the exponent value is not supported
TPM_RC_CANCELLED	key generation has been canceled
TPM_RC_VALUE	exponent is not prime or is less than 3; or could not find a prime using the provided parameters

```
610 static TPM_RC
611 CryptGenerateKeyRSA(
612     TPMT_PUBLIC         *publicArea,        // IN/OUT: The public area template for
613                                             //      the new key. The public key
614                                             //      area will be replaced by the
615                                             //      product of two primes found by
616                                             //      this function
617     TPMT_SENSITIVE      *sensitive,         // OUT: the sensitive area will be
618                                             //      updated to contain the first
619                                             //      prime and the symmetric
620                                             //      encryption key
621     TPM_ALG_ID           hashAlg,           // IN: the hash algorithm for the KDF
622     TPM2B_SEED          *seed,              // IN: Seed for the creation
623     TPM2B_NAME          *name,              // IN: Object name
624     UINT32              *counter            // OUT: last iteration of the counter
625 )
626 {
627     CRYPT_RESULT    retVal;
628     UINT32          exponent = publicArea->parameters.rsaDetail.exponent;
629
630     TEST_HASH(hashAlg);
```

```
631         TEST(ALG_NULL_VALUE);
632
633         // In this implementation, only the default exponent is allowed
634         if(exponent != 0 && exponent != RSA_DEFAULT_PUBLIC_EXPONENT)
635             return TPM_RC_RANGE;
636         exponent = RSA_DEFAULT_PUBLIC_EXPONENT;
637
638         *counter = 0;
639
640         // _cpri_GenerateKeyRSA can return CRYPT_CANCEL or CRYPT_FAIL
641         retVal = _cpri__GenerateKeyRSA(&publicArea->unique.rsa.b,
642                                        &sensitive->sensitive.rsa.b,
643                                        publicArea->parameters.rsaDetail.keyBits,
644                                        exponent,
645                                        hashAlg,
646                                        &seed->b,
647                                        "RSA key by vendor",
648                                        &name->b,
649                                        counter);
650
651         // CRYPT_CANCEL -> TPM_RC_CANCELLED; CRYPT_FAIL -> TPM_RC_VALUE
652         return TranslateCryptErrors(retVal);
653
654     }
```

10.2.5.4 CryptLoadPrivateRSA()

This function is called to generate the private exponent of an RSA key. It uses CryptTestKeyRSA().

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of *rsaKey* are not matched

```
655     TPM_RC
656     CryptLoadPrivateRSA(
657         OBJECT          *rsaKey         // IN: the RSA key object
658         )
659     {
660         TPM_RC           result;
661         TPMT_PUBLIC     *publicArea = &rsaKey->publicArea;
662         TPMT_SENSITIVE  *sensitive = &rsaKey->sensitive;
663
664         // Load key by computing the private exponent
665         // TPM_RC_BINDING
666         result = CryptTestKeyRSA(&(rsaKey->privateExponent.b),
667                                  publicArea->parameters.rsaDetail.exponent,
668                                  &(publicArea->unique.rsa.b),
669                                  &(sensitive->sensitive.rsa.b),
670                                  NULL);
671         if(result == TPM_RC_SUCCESS)
672             rsaKey->attributes.privateExp = SET;
673
674         return result;
675     }
```

10.2.5.5 CryptSelectRSAScheme()

This function is used by TPM2_RSA_Decrypt() and TPM2_RSA_Encrypt(). It sets up the rules to select a scheme between input and object default. This function assume the RSA object is loaded. If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both the object and *scheme* are not TPM_ALG_NULL, then if the schemes

are the same, the input scheme will be chosen. if the scheme are not compatible, a NULL pointer will be returned.

The return pointer may point to a TPM_ALG_NULL scheme.

```
676 TPMT_RSA_DECRYPT*
677 CryptSelectRSAScheme(
678     TPMI_DH_OBJECT       rsaHandle,     // IN: handle of sign key
679     TPMT_RSA_DECRYPT    *scheme         // IN: a sign or decrypt scheme
680     )
681 {
682     OBJECT              *rsaObject;
683     TPMT_ASYM_SCHEME    *keyScheme;
684     TPMT_RSA_DECRYPT    *retVal = NULL;
685
686     // Get sign object pointer
687     rsaObject = ObjectGet(rsaHandle);
688     keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
689
690     // if the default scheme of the object is TPM_ALG_NULL, then select the
691     // input scheme
692     if(keyScheme->scheme == TPM_ALG_NULL)
693     {
694         retVal = scheme;
695     }
696     // if the object scheme is not TPM_ALG_NULL and the input scheme is
697     // TPM_ALG_NULL, then select the default scheme of the object.
698     else if(scheme->scheme == TPM_ALG_NULL)
699     {
700         // if input scheme is NULL
701         retVal = (TPMT_RSA_DECRYPT *)keyScheme;
702     }
703     // get here if both the object scheme and the input scheme are
704     // not TPM_ALG_NULL. Need to insure that they are the same.
705     // IMPLEMENTATION NOTE: This could cause problems if future versions have
706     // schemes that have more values than just a hash algorithm. A new function
707     // (IsSchemeSame()) might be needed then.
708     else if(   keyScheme->scheme == scheme->scheme
709             && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
710     {
711         retVal = scheme;
712     }
713     // two different, incompatible schemes specified will return NULL
714     return retVal;
715 }
```

10.2.5.6 CryptDecryptRSA()

This function is the interface to _cpri__DecryptRSA(). It handles the return codes from that function and converts them from CRYPT_RESULT to TPM_RC values. The *rsaKey* parameter must reference an RSA decryption key

Error Returns	Meaning
TPM_RC_BINDING	Public and private parts of the key are not cryptographically bound.
TPM_RC_SIZE	Size of data to decrypt is not the same as the key size.
TPM_RC_VALUE	Numeric value of the encrypted data is greater than the public exponent, or output buffer is too small for the decrypted message.

```
716 TPM_RC
717 CryptDecryptRSA(
718     UINT16              *dataOutSize,   // OUT: size of plain text in byte
```

```
719         BYTE                *dataOut,       // OUT: plain text
720         OBJECT              *rsaKey,        // IN: internal RSA key
721         TPMT_RSA_DECRYPT    *scheme,        // IN: selects the padding scheme
722         UINT16               cipherInSize,  // IN: size of cipher text  in byte
723         BYTE                *cipherIn,      // IN: cipher text
724         const char          *label          // IN: a label, when needed
725         )
726     {
727         RSA_KEY         key;
728         CRYPT_RESULT    retVal = CRYPT_SUCCESS;
729         UINT32          dSize;                  // Place to put temporary value for the
730                                                 // returned data size
731         TPMI_ALG_HASH   hashAlg = TPM_ALG_NULL; // hash algorithm in the selected
732                                                 // padding scheme
733         TPM_RC          result = TPM_RC_SUCCESS;
734
735         // pointer checks
736         pAssert(    (dataOutSize != NULL) && (dataOut != NULL)
737                 &&  (rsaKey != NULL) && (cipherIn != NULL));
738
739         // The public type is a RSA decrypt key
740         pAssert(    (rsaKey->publicArea.type == TPM_ALG_RSA
741                 &&  rsaKey->publicArea.objectAttributes.decrypt == SET));
742
743         // Must have the private portion loaded.  This check is made before this
744         // function is called.
745         pAssert(rsaKey->attributes.publicOnly == CLEAR);
746
747         // decryption requires that the private modulus be present
748         if(rsaKey->attributes.privateExp == CLEAR)
749         {
750
751             // Load key by computing the private exponent
752             // CryptLoadPrivateRSA may return TPM_RC_BINDING
753             result = CryptLoadPrivateRSA(rsaKey);
754         }
755
756         // the input buffer must be the size of the key
757         if(result == TPM_RC_SUCCESS)
758         {
759             if(cipherInSize != rsaKey->publicArea.unique.rsa.t.size)
760                 result = TPM_RC_SIZE;
761             else
762             {
763                 BuildRSA(rsaKey, &key);
764
765                 // Initialize the dOutSize parameter
766                 dSize = *dataOutSize;
767
768                 // For OAEP scheme, initialize the hash algorithm for padding
769                 if(scheme->scheme == TPM_ALG_OAEP)
770                 {
771                     hashAlg = scheme->details.oaep.hashAlg;
772                     TEST_HASH(hashAlg);
773                 }
774                 // See if the padding mode needs to be tested
775                 TEST(scheme->scheme);
776
777                 // _cpri__DecryptRSA may return CRYPT_PARAMETER CRYPT_FAIL CRYPT_SCHEME
778                 retVal = _cpri__DecryptRSA(&dSize, dataOut, &key, scheme->scheme,
779                                            cipherInSize, cipherIn, hashAlg, label);
780
781                 // Scheme must have been validated when the key was loaded/imported
782                 pAssert(retVal != CRYPT_SCHEME);
783
784                 // Set the return size
```

```
785             pAssert(dSize <= UINT16_MAX);
786             *dataOutSize = (UINT16)dSize;
787
788             // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_FAIL -> TPM_RC_VALUE
789             result = TranslateCryptErrors(retVal);
790         }
791     }
792     return result;
793 }
```

10.2.5.7 CryptEncryptRSA()

This function provides the interface to _cpri__EncryptRSA(). The object referenced by **rsaKey** is required to be an RSA decryption key.

Error Returns	Meaning
TPM_RC_SCHEME	*scheme* is not supported
TPM_RC_VALUE	numeric value of *dataIn* is greater than the key modulus

```
794 TPM_RC
795 CryptEncryptRSA(
796     UINT16              *cipherOutSize, // OUT: size of cipher text in byte
797     BYTE                *cipherOut,     // OUT: cipher text
798     OBJECT              *rsaKey,        // IN: internal RSA key
799     TPMT_RSA_DECRYPT    *scheme,        // IN: selects the padding scheme
800     UINT16               dataInSize,    // IN: size of plain text in byte
801     BYTE                *dataIn,        // IN: plain text
802     const char          *label          // IN: an optional label
803     )
804 {
805     RSA_KEY              key;
806     CRYPT_RESULT         retVal;
807     UINT32               cOutSize;                  // Conversion variable
808     TPMI_ALG_HASH        hashAlg = TPM_ALG_NULL;    // hash algorithm in selected
809                                                     // padding scheme
810
811     // must have a pointer to a key and some data to encrypt
812     pAssert(rsaKey != NULL && dataIn != NULL);
813
814     // The public type is a RSA decryption key
815     pAssert(   rsaKey->publicArea.type == TPM_ALG_RSA
816             && rsaKey->publicArea.objectAttributes.decrypt == SET);
817
818     // If the cipher buffer must be provided and it must be large enough
819     // for the result
820     pAssert(   cipherOut != NULL
821             && cipherOutSize != NULL
822             && *cipherOutSize >= rsaKey->publicArea.unique.rsa.t.size);
823
824     // Only need the public key and exponent for encryption
825     BuildRSA(rsaKey, &key);
826
827     // Copy the size to the conversion buffer
828     cOutSize = *cipherOutSize;
829
830     // For OAEP scheme, initialize the hash algorithm for padding
831     if(scheme->scheme == TPM_ALG_OAEP)
832     {
833         hashAlg = scheme->details.oaep.hashAlg;
834         TEST_HASH(hashAlg);
835     }
836
```

```
837         // This is a public key operation and does not require that the private key
838         // be loaded. To verify this, need to do the full algorithm
839         TEST(scheme->scheme);
840
841         // Encrypt the data with the public exponent
842         // _cpri__EncryptRSA may return CRYPT_PARAMETER or CRYPT_SCHEME
843         retVal = _cpri__EncryptRSA(&cOutSize,cipherOut, &key, scheme->scheme,
844                                    dataInSize, dataIn, hashAlg, label);
845
846         pAssert (cOutSize <= UINT16_MAX);
847         *cipherOutSize = (UINT16)cOutSize;
848         // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_SCHEME -> TPM_RC_SCHEME
849         return TranslateCryptErrors(retVal);
850     }
```

10.2.5.8 CryptSignRSA()

This function is used to sign a digest with an RSA signing key.

Error Returns	Meaning
TPM_RC_BINDING	public and private part of *signKey* are not properly bound
TPM_RC_SCHEME	*scheme* is not supported
TPM_RC_VALUE	*hashData* is larger than the modulus of *signKey*, or the size of *hashData* does not match hash algorithm in *scheme*

```
851     static TPM_RC
852     CryptSignRSA(
853         OBJECT              *signKey,       // IN: RSA key signs the hash
854         TPMT_SIG_SCHEME     *scheme,        // IN: sign scheme
855         TPM2B_DIGEST        *hashData,      // IN: hash to be signed
856         TPMT_SIGNATURE      *sig            // OUT: signature
857         )
858     {
859         UINT32               signSize;
860         RSA_KEY              key;
861         CRYPT_RESULT         retVal;
862         TPM_RC               result = TPM_RC_SUCCESS;
863
864         pAssert(    (signKey != NULL) && (scheme != NULL)
865                     && (hashData != NULL) && (sig != NULL));
866
867         // assume that the key has private part loaded and that it is a signing key.
868         pAssert(   (signKey->attributes.publicOnly == CLEAR)
869                 && (signKey->publicArea.objectAttributes.sign == SET));
870
871         // check if the private exponent has been computed
872         if(signKey->attributes.privateExp == CLEAR)
873             // May return TPM_RC_BINDING
874             result = CryptLoadPrivateRSA(signKey);
875
876         if(result == TPM_RC_SUCCESS)
877         {
878             BuildRSA(signKey, &key);
879
880             // Make sure that the hash is tested
881             TEST_HASH(sig->signature.any.hashAlg);
882
883             // Run a test of the RSA sign
884             TEST(scheme->scheme);
885
886             // _crypi__SignRSA can return CRYPT_SCHEME and CRYPT_PARAMETER
887             retVal = _cpri__SignRSA(&signSize,
```

```
888                                     sig->signature.rsassa.sig.t.buffer,
889                                     &key,
890                                     sig->sigAlg,
891                                     sig->signature.any.hashAlg,
892                                     hashData->t.size, hashData->t.buffer);
893         pAssert(signSize <= UINT16_MAX);
894         sig->signature.rsassa.sig.t.size = (UINT16)signSize;
895
896         // CRYPT_SCHEME -> TPM_RC_SCHEME; CRYPT_PARAMTER -> TPM_RC_VALUE
897         result = TranslateCryptErrors(retVal);
898     }
899     return result;
900 }
```

10.2.5.9 CryptRSAVerifySignature()

This function is used to verify signature signed by a RSA key.

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not genuine
TPM_RC_SCHEME	signature scheme not supported

```
901 static TPM_RC
902 CryptRSAVerifySignature(
903     OBJECT          *signKey,       // IN: RSA key signed the hash
904     TPM2B_DIGEST    *digestData,    // IN: digest being signed
905     TPMT_SIGNATURE  *sig            // IN: signature to be verified
906     )
907 {
908     RSA_KEY             key;
909     CRYPT_RESULT        retVal;
910     TPM_RC              result;
911
912     // Validate parameter assumptions
913     pAssert((signKey != NULL) && (digestData != NULL) && (sig != NULL));
914
915     TEST_HASH(sig->signature.any.hashAlg);
916     TEST(sig->sigAlg);
917
918     // This is a public-key-only operation
919     BuildRSA(signKey, &key);
920
921     // Call crypto engine to verify signature
922     // _cpri_ValidateSignaturRSA may return CRYPT_FAIL or CRYPT_SCHEME
923     retVal = _cpri__ValidateSignatureRSA(&key,
924                                          sig->sigAlg,
925                                          sig->signature.any.hashAlg,
926                                          digestData->t.size,
927                                          digestData->t.buffer,
928                                          sig->signature.rsassa.sig.t.size,
929                                          sig->signature.rsassa.sig.t.buffer,
930                                          0);
931     // _cpri__ValidateSignatureRSA can return CRYPT_SUCCESS, CRYPT_FAIL, or
932     // CRYPT_SCHEME. Translate CRYPT_FAIL to TPM_RC_SIGNATURE
933     if(retVal == CRYPT_FAIL)
934         result = TPM_RC_SIGNATURE;
935     else
936         // CRYPT_SCHEME -> TPM_RC_SCHEME
937         result = TranslateCryptErrors(retVal);
938
939     return result;
940 }
```

```
941 #endif //TPM_ALG_RSA      //% 2
```

10.2.6 ECC Functions

10.2.6.1 CryptEccGetCurveDataPointer()

This function returns a pointer to an ECC_CURVE_VALUES structure that contains the parameters for the key size and schemes for a given curve.

```
942 #ifdef TPM_ALG_ECC //% 3
943 static const ECC_CURVE    *
944 CryptEccGetCurveDataPointer(
945     TPM_ECC_CURVE    curveID        // IN: id of the curve
946     )
947 {
948     return _cpri__EccGetParametersByCurveId(curveID);
949 }
```

10.2.6.2 CryptEccGetKeySizeInBits()

This function returns the size in bits of the key associated with a curve.

```
950 UINT16
951 CryptEccGetKeySizeInBits(
952     TPM_ECC_CURVE    curveID        // IN: id of the curve
953     )
954 {
955     const ECC_CURVE         *curve = CryptEccGetCurveDataPointer(curveID);
956     UINT16                   keySizeInBits = 0;
957 
958     if(curve != NULL)
959         keySizeInBits = curve->keySizeBits;
960 
961     return keySizeInBits;
962 }
```

10.2.6.3 CryptEccGetKeySizeBytes()

This macro returns the size of the ECC key in bytes. It uses CryptEccGetKeySizeInBits().

```
963 // The next lines will be placed in CyrptUtil_fp.h with the //% removed
964 //% #define CryptEccGetKeySizeInBytes(curve)           \
965 //%             ((CryptEccGetKeySizeInBits(curve)+7)/8)
```

10.2.6.4 CryptEccGetParameter()

This function returns a pointer to an ECC curve parameter. The parameter is selected by a single character designator from the set of {pnabxyh}.

```
966 LIB_EXPORT const TPM2B *
967 CryptEccGetParameter(
968     char             p,             // IN: the parameter selector
969     TPM_ECC_CURVE    curveId        // IN: the curve id
970     )
971 {
972     const ECC_CURVE     *curve = _cpri__EccGetParametersByCurveId(curveId);
973     const TPM2B         *parameter = NULL;
974 
975     if(curve != NULL)
```

```
976         {
977             switch (p)
978             {
979             case 'p':
980                 parameter = curve->curveData->p;
981                 break;
982             case 'n':
983                 parameter =  curve->curveData->n;
984                 break;
985             case 'a':
986                 parameter =  curve->curveData->a;
987                 break;
988             case 'b':
989                 parameter =  curve->curveData->b;
990                 break;
991             case 'x':
992                 parameter =  curve->curveData->x;
993                 break;
994             case 'y':
995                 parameter =  curve->curveData->y;
996                 break;
997             case 'h':
998                 parameter =  curve->curveData->h;
999                 break;
1000            default:
1001                break;
1002            }
1003        }
1004        return parameter;
1005    }
```

10.2.6.5 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```
1006    const TPMT_ECC_SCHEME *
1007    CryptGetCurveSignScheme(
1008        TPM_ECC_CURVE    curveId        // IN: The curve selector
1009        )
1010    {
1011        const ECC_CURVE         *curve = _cpri__EccGetParametersByCurveId(curveId);
1012        const TPMT_ECC_SCHEME   *scheme = NULL;
1013
1014        if(curve != NULL)
1015            scheme =  &(curve->sign);
1016        return scheme;
1017    }
```

10.2.6.6 CryptEccIsPointOnCurve()

This function will validate that an ECC point is on the curve of given *curveID*.

Return Value	Meaning
TRUE	if the point is on curve
FALSE	if the point is not on curve

```
1018    BOOL
1019    CryptEccIsPointOnCurve(
1020        TPM_ECC_CURVE    curveID,       // IN: ECC curve ID
1021        TPMS_ECC_POINT  *Q              // IN: ECC point
1022        )
```

```
{
    // Make sure that point multiply is working
    TEST(TPM_ALG_ECC);
    // Check point on curve logic by seeing if the test key is on the curve

    // 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;
}
```

10.2.6.7 CryptNewEccKey()

This function creates a random ECC key that is not derived from other parameters as is a Primary Key.

```
TPM_RC
CryptNewEccKey(
    TPM_ECC_CURVE            curveID,       // IN: ECC curve
    TPMS_ECC_POINT          *publicPoint,   // OUT: public point
    TPM2B_ECC_PARAMETER     *sensitive      // OUT: private area
    )
{
    TPM_RC               result = TPM_RC_SUCCESS;
    // _cpri__GetEphemeralECC may return CRYPT_PARAMETER
    if(_cpri__GetEphemeralEcc(publicPoint, sensitive, curveID) != CRYPT_SUCCESS)
        // Something is wrong with the key.
        result = TPM_RC_KEY;

    return result;
}
```

10.2.6.8 CryptEccPointMultiply()

This function is used to perform a point multiply *R* = [*d*]*Q*. If *Q* is not provided, the multiplication is performed using the generator point of the curve.

Error Returns	Meaning
TPM_RC_ECC_POINT	invalid optional ECC point *pIn*
TPM_RC_NO_RESULT	multiplication resulted in a point at infinity
TPM_RC_CANCELED	if a self-test was done, it might have been aborted

```
TPM_RC
CryptEccPointMultiply(
    TPMS_ECC_POINT          *pOut,          // OUT: output point
    TPM_ECC_CURVE            curveId,       // IN: curve selector
    TPM2B_ECC_PARAMETER     *dIn,           // IN: public scalar
    TPMS_ECC_POINT          *pIn            // IN: optional point
    )
{
    TPM2B_ECC_PARAMETER     *n = NULL;
    CRYPT_RESULT             retVal;

    pAssert(pOut != NULL && dIn != NULL);

    if(pIn != NULL)
    {
        n = dIn;
        dIn = NULL;
```

```
1067        }
1068        // Do a test of point multiply
1069        TEST(TPM_ALG_ECC);
1070
1071        // _cpri__EccPointMultiply may return CRYPT_POINT or CRYPT_NO_RESULT
1072        retVal = _cpri__EccPointMultiply(pOut, curveId, dIn, pIn, n);
1073
1074        // CRYPT_POINT->TPM_RC_ECC_POINT and CRYPT_NO_RESULT->TPM_RC_NO_RESULT
1075        return TranslateCryptErrors(retVal);
1076    }
```

10.2.6.9 CryptGenerateKeyECC()

This function generates an ECC key from a seed value.

The method here may not work for objects that have an order *(G)* that with a different size than a private key.

Error Returns	Meaning
TPM_RC_VALUE	hash algorithm is not supported

```
1077    static TPM_RC
1078    CryptGenerateKeyECC(
1079        TPMT_PUBLIC     *publicArea,    // IN/OUT: The public area template for the new
1080                                        //     key.
1081        TPMT_SENSITIVE  *sensitive,     // IN/OUT: the sensitive area
1082        TPM_ALG_ID       hashAlg,       // IN: algorithm for the KDF
1083        TPM2B_SEED      *seed,          // IN: the seed value
1084        TPM2B_NAME      *name,          // IN: the name of the object
1085        UINT32          *counter        // OUT: the iteration counter
1086        )
1087    {
1088        CRYPT_RESULT         retVal;
1089
1090        TEST_HASH(hashAlg);
1091        TEST(ALG_ECDSA_VALUE); // ECDSA is used to verify each key
1092
1093        // The iteration counter has no meaning for ECC key generation. The parameter
1094        // will be overloaded for those implementations that have a requirement for
1095        // doing pair-wise consistency checks on signing keys. If the counter parameter
1096        // is 0 or NULL, then no consistency check is done. If it is other than 0, then
1097        // a consistency check is run. This modification allow this code to work with
1098        // the existing versions of the CrytpoEngine and with FIPS-compliant versions
1099        // as well.
1100        *counter = (UINT32)(publicArea->objectAttributes.sign == SET);
1101
1102        // _cpri__GenerateKeyEcc only has one error return (CRYPT_PARAMETER) which means
1103        // that the hash algorithm is not supported. This should not be possible
1104        retVal = _cpri__GenerateKeyEcc(&publicArea->unique.ecc,
1105                                       &sensitive->sensitive.ecc,
1106                                       publicArea->parameters.eccDetail.curveID,
1107                                       hashAlg, &seed->b, "ECC key by vendor",
1108                                       &name->b, counter);
1109        // This will only be useful if _cpri__GenerateKeyEcc return CRYPT_CANCEL
1110        return TranslateCryptErrors(retVal);
1111    }
```

10.2.6.10 CryptSignECC()

This function is used for ECC signing operations. If the signing scheme is a split scheme, and the signing operation is successful, the commit value is retired.

Error Returns	Meaning
TPM_RC_SCHEME	unsupported *scheme*
TPM_RC_VALUE	invalid commit status (in case of a split scheme) or failed to generate r value.

```
1112  static TPM_RC
1113  CryptSignECC(
1114      OBJECT              *signKey,       // IN: ECC key to sign the hash
1115      TPMT_SIG_SCHEME     *scheme,        // IN: sign scheme
1116      TPM2B_DIGEST        *hashData,      // IN: hash to be signed
1117      TPMT_SIGNATURE      *signature      // OUT: signature
1118      )
1119  {
1120      TPM2B_ECC_PARAMETER      r;
1121      TPM2B_ECC_PARAMETER     *pr = NULL;
1122      CRYPT_RESULT             retVal;
1123
1124      // Run a test of the ECC sign and verify if it has not already been run
1125      TEST_HASH(scheme->details.any.hashAlg);
1126      TEST(scheme->scheme);
1127
1128      if(CryptIsSplitSign(scheme->scheme))
1129      {
1130          // When this code was written, the only split scheme was ECDAA
1131          // (which can also be used for U-Prove).
1132          if(!CryptGenerateR(&r,
1133                             &scheme->details.ecdaa.count,
1134                             signKey->publicArea.parameters.eccDetail.curveID,
1135                             &signKey->name))
1136              return TPM_RC_VALUE;
1137          pr = &r;
1138      }
1139      // Call crypto engine function to sign
1140      // _cpri__SignEcc may return CRYPT_SCHEME
1141      retVal = _cpri__SignEcc(&signature->signature.ecdsa.signatureR,
1142                              &signature->signature.ecdsa.signatureS,
1143                              scheme->scheme,
1144                              scheme->details.any.hashAlg,
1145                              signKey->publicArea.parameters.eccDetail.curveID,
1146                              &signKey->sensitive.sensitive.ecc,
1147                              &hashData->b,
1148                              pr
1149                              );
1150      if(CryptIsSplitSign(scheme->scheme) && retVal == CRYPT_SUCCESS)
1151          CryptEndCommit(scheme->details.ecdaa.count);
1152      // CRYPT_SCHEME->TPM_RC_SCHEME
1153      return TranslateCryptErrors(retVal);
1154  }
```

10.2.6.11 CryptECCVerifySignature()

This function is used to verify a signature created with an ECC key.

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not valid
TPM_RC_SCHEME	the signing scheme or *hashAlg* is not supported

```
1155  static TPM_RC
1156  CryptECCVerifySignature(
1157      OBJECT          *signKey,       // IN: ECC key signed the hash
```

```
1158        TPM2B_DIGEST    *digestData,    // IN: digest being signed
1159        TPMT_SIGNATURE  *signature      // IN: signature to be verified
1160        )
1161    {
1162        CRYPT_RESULT        retVal;
1163
1164        TEST_HASH(signature->signature.any.hashAlg);
1165        TEST(signature->sigAlg);
1166
1167        // This implementation uses the fact that all the defined ECC signing
1168        // schemes have the hash as the first parameter.
1169        // _cpriValidateSignatureEcc may return CRYPT_FAIL or CRYP_SCHEME
1170        retVal = _cpri__ValidateSignatureEcc(&signature->signature.ecdsa.signatureR,
1171                                             &signature->signature.ecdsa.signatureS,
1172                                             signature->sigAlg,
1173                                             signature->signature.any.hashAlg,
1174                                             signKey->publicArea.parameters.eccDetail.curveID,
1175                                             &signKey->publicArea.unique.ecc,
1176                                             &digestData->b);
1177        if(retVal == CRYPT_FAIL)
1178            return TPM_RC_SIGNATURE;
1179        // CRYPT_SCHEME->TPM_RC_SCHEME
1180        return TranslateCryptErrors(retVal);
1181    }
```

10.2.6.12 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.c*ommitArray* bit associated with the input value of *c* is SET. If not, the TPM returns FALSE and no *r* value is generated.

Return Value	Meaning
TRUE	r value computed
FALSE	no r value computed

```
1182    BOOL
1183    CryptGenerateR(
1184        TPM2B_ECC_PARAMETER     *r,             // OUT: the generated random value
1185        UINT16                  *c,             // IN/OUT: count value.
1186        TPMI_ECC_CURVE           curveID,       // IN: the curve for the value
1187        TPM2B_NAME              *name           // IN: optional name of a key to
1188                                                //     associate with 'r'
1189        )
1190    {
1191        // This holds the marshaled g_commitCounter.
1192        TPM2B_TYPE(8B, 8);
1193        TPM2B_8B                cntr = {8,{0}};
1194
1195        UINT32                   iterations;
1196        const TPM2B             *n;
1197        UINT64                   currentCount = gr.commitCounter;
1198        // This is just to suppress a compiler warning about a conditional expression
1199        // being a constant. This is because of the macro expansion of ryptKDFa
1200        TPMI_ALG_HASH            hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
1201
1202        n =  CryptEccGetParameter('n', curveID);
1203        pAssert(r != NULL && n != NULL);
1204
1205        // If this is the commit phase, use the current value of the commit counter
1206        if(c != NULL)
```

```
1207        {
1208
1209            UINT16      t1;
1210            // if the array bit is not set, can't use the value.
1211            if(!BitIsSet((*c & COMMIT_INDEX_MASK), gr.commitArray,
1212                         sizeof(gr.commitArray)))
1213                return FALSE;
1214
1215            // If it is the sign phase, figure out what the counter value was
1216            // when the commitment was made.
1217            //
1218            // When gr.commitArray has less than 64K bits, the extra
1219            // bits of 'c' are used as a check to make sure that the
1220            // signing operation is not using an out of range count value
1221            t1 = (UINT16)currentCount;
1222
1223            // If the lower bits of c are greater or equal to the lower bits of t1
1224            // then the upper bits of t1 must be one more than the upper bits
1225            // of c
1226            if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
1227                // Since the counter is behind, reduce the current count
1228                currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
1229
1230            t1 = (UINT16)currentCount;
1231            if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
1232                return FALSE;
1233            // set the counter to the value that was
1234            // present when the commitment was made
1235            currentCount = (currentCount & 0xffffffffffff0000) | *c;
1236
1237        }
1238        // Marshal the count value to a TPM2B buffer for the KDF
1239        cntr.t.size = sizeof(currentCount);
1240        UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
1241
1242        // Now can do the KDF to create the random value for the signing operation
1243        // During the creation process, we may generate an r that does not meet the
1244        // requirements of the random value.
1245        // want to generate a new r.
1246
1247        r->t.size = n->size;
1248
1249        // Arbitrary upper limit on the number of times that we can look for
1250        // a suitable random value.  The normally number of tries will be 1.
1251        for(iterations = 1; iterations < 1000000;)
1252        {
1253            BYTE    *pr = &r->b.buffer[0];
1254            int     i;
1255            CryptKDFa(hashAlg, &gr.commitNonce.b, "ECDAA Commit",
1256                      name, &cntr.b, n->size * 8, r->t.buffer, &iterations);
1257
1258            // random value must be less than the prime
1259            if(CryptCompare(r->b.size, r->b.buffer, n->size, n->buffer) >= 0)
1260                continue;
1261
1262            // in this implementation it is required that at least bit
1263            // in the upper half of the number be set
1264            for(i = n->size/2; i > 0; i--)
1265                if(*pr++ != 0)
1266                    return TRUE;
1267        }
1268        return FALSE;
1269    }
```


10.2.6.13 CryptCommit()

This function is called when the count value is committed. The gr.c*ommitArray* value associated with the current count value is SET and *g_commitCounter* is incremented. The low-order 16 bits of old value of the counter is returned.

```
1270    UINT16
1271    CryptCommit(
1272        void
1273        )
1274    {
1275        UINT16      oldCount = (UINT16)gr.commitCounter;
1276        gr.commitCounter++;
1277        BitSet(oldCount & COMMIT_INDEX_MASK, gr.commitArray, sizeof(gr.commitArray));
1278        return oldCount;
1279    }
```

10.2.6.14 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the gr.c*ommitArray* bit associated with the count value so that it can't be used again.

```
1280    void
1281    CryptEndCommit(
1282        UINT16          c               // IN: the counter value of the commitment
1283        )
1284    {
1285        BitClear((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
1286    }
```

10.2.6.15 CryptCommitCompute()

This function performs the computations for the TPM2_Commit() command. This could be a macro.

Error Returns	Meaning
TPM_RC_NO_RESULT	*K*, *L*, or *E* is the point at infinity
TPM_RC_CANCELLED	command was canceled

```
1287    TPM_RC
1288    CryptCommitCompute(
1289        TPMS_ECC_POINT          *K,             // OUT: [d]B
1290        TPMS_ECC_POINT          *L,             // OUT: [r]B
1291        TPMS_ECC_POINT          *E,             // OUT: [r]M
1292        TPM_ECC_CURVE            curveID,       // IN: The curve for the computation
1293        TPMS_ECC_POINT          *M,             // IN: M (P1)
1294        TPMS_ECC_POINT          *B,             // IN: B (x2, y2)
1295        TPM2B_ECC_PARAMETER     *d,             // IN: the private scalar
1296        TPM2B_ECC_PARAMETER     *r              // IN: the computed r value
1297        )
1298    {
1299        TEST(ALG_ECDH_VALUE);
1300        // CRYPT_NO_RESULT->TPM_RC_NO_RESULT CRYPT_CANCEL->TPM_RC_CANCELLED
1301        return TranslateCryptErrors(
1302                    _cpri__EccCommitCompute(K, L , E, curveID, M, B, d, r));
1303    }
```

10.2.6.16 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve

Return Value	Meaning
TRUE	Get parameters success
FALSE	Unsupported ECC curve ID

```
1304  BOOL
1305  CryptEccGetParameters(
1306      TPM_ECC_CURVE                curveId,       // IN: ECC curve ID
1307      TPMS_ALGORITHM_DETAIL_ECC   *parameters     // OUT: ECC parameter
1308      )
1309  {
1310      const ECC_CURVE             *curve = _cpri__EccGetParametersByCurveId(curveId);
1311      const ECC_CURVE_DATA        *data;
1312      BOOL                         found = curve != NULL;
1313
1314      if(found)
1315      {
1316
1317          data = curve->curveData;
1318
1319          parameters->curveID = curve->curveId;
1320
1321          // Key size in bit
1322          parameters->keySize = curve->keySizeBits;
1323
1324          // KDF
1325          parameters->kdf = curve->kdf;
1326
1327          // Sign
1328          parameters->sign = curve->sign;
1329
1330          // Copy p value
1331          MemoryCopy2B(&parameters->p.b, data->p, sizeof(parameters->p.t.buffer));
1332
1333          // Copy a value
1334          MemoryCopy2B(&parameters->a.b, data->a, sizeof(parameters->a.t.buffer));
1335
1336          // Copy b value
1337          MemoryCopy2B(&parameters->b.b, data->b, sizeof(parameters->b.t.buffer));
1338
1339          // Copy Gx value
1340          MemoryCopy2B(&parameters->gX.b, data->x, sizeof(parameters->gX.t.buffer));
1341
1342          // Copy Gy value
1343          MemoryCopy2B(&parameters->gY.b, data->y, sizeof(parameters->gY.t.buffer));
1344
1345          // Copy n value
1346          MemoryCopy2B(&parameters->n.b, data->n, sizeof(parameters->n.t.buffer));
1347
1348          // Copy h value
1349          MemoryCopy2B(&parameters->h.b, data->h, sizeof(parameters->h.t.buffer));
1350      }
1351      return found;
1352  }
1353  #if CC_ZGen_2Phase == YES
```

CryptEcc2PhaseKeyExchange() This is the interface to the key exchange function.

```
1354  TPM_RC
1355  CryptEcc2PhaseKeyExchange(
```

```
1356        TPMS_ECC_POINT          *outZ1,         // OUT: the computed point
1357        TPMS_ECC_POINT          *outZ2,         // OUT: optional second point
1358        TPM_ALG_ID               scheme,        // IN: the key exchange scheme
1359        TPM_ECC_CURVE            curveId,       // IN: the curve for the computation
1360        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1361        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1362        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1363        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1364        )
1365    {
1366        return (TranslateCryptErrors(_cpri__C_2_2_KeyExchange(outZ1,
1367                                                              outZ2,
1368                                                              scheme,
1369                                                              curveId,
1370                                                              dsA,
1371                                                              deA,
1372                                                              QsB,
1373                                                              QeB)));
1374    }
1375    #endif //  CC_ZGen_2Phase
1376    #endif //TPM_ALG_ECC  //% 3
```

10.2.6.17 CryptIsSchemeAnonymous()

This function is used to test a scheme to see if it is an anonymous scheme The only anonymous scheme is ECDAA. ECDAA can be used to do things like U-Prove.

```
1377    BOOL
1378    CryptIsSchemeAnonymous(
1379        TPM_ALG_ID       scheme          // IN: the scheme algorithm to test
1380        )
1381    {
1382    #ifdef TPM_ALG_ECDAA
1383        return (scheme == TPM_ALG_ECDAA);
1384    #else
1385        UNREFERENCED(scheme);
1386        return  0;
1387    #endif
1388    }
```

10.2.7 Symmetric Functions

10.2.7.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```
1389    void
1390    ParmDecryptSym(
1391        TPM_ALG_ID       symAlg,         // IN: the symmetric algorithm
1392        TPM_ALG_ID       hash,           // IN: hash algorithm for KDFa
1393        UINT16           keySizeInBits,  // IN: key key size in bit
1394        TPM2B           *key,            // IN: KDF HMAC key
1395        TPM2B           *nonceCaller,    // IN: nonce caller
1396        TPM2B           *nonceTpm,       // IN: nonce TPM
1397        UINT32           dataSize,       // IN: size of parameter buffer
1398        BYTE            *data            // OUT: buffer to be decrypted
1399        )
1400    {
1401        // KDF output buffer
1402        // It contains parameters for the CFB encryption
1403        // From MSB to LSB, they are the key and iv
1404        BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
```

```
1405        // Symmetric key size in byte
1406        UINT16          keySize = (keySizeInBits + 7) / 8;
1407        TPM2B_IV        iv;
1408
1409        iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
1410        // If there is decryption to do...
1411        if(iv.t.size > 0)
1412        {
1413            // Generate key and iv
1414            CryptKDFa(hash, key, "CFB", nonceCaller, nonceTpm,
1415                      keySizeInBits + (iv.t.size * 8), symParmString, NULL);
1416            MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size,
1417                       sizeof(iv.t.buffer));
1418
1419            CryptSymmetricDecrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB,
1420                                  symParmString, &iv, dataSize, data);
1421        }
1422        return;
1423    }
```

10.2.7.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
1424    void
1425    ParmEncryptSym(
1426        TPM_ALG_ID       symAlg,        // IN: symmetric algorithm
1427        TPM_ALG_ID       hash,          // IN: hash algorithm for KDFa
1428        UINT16           keySizeInBits, // IN: AES key size in bit
1429        TPM2B           *key,           // IN: KDF HMAC key
1430        TPM2B           *nonceCaller,   // IN: nonce caller
1431        TPM2B           *nonceTpm,      // IN: nonce TPM
1432        UINT32           dataSize,      // IN: size of parameter buffer
1433        BYTE            *data           // OUT: buffer to be encrypted
1434        )
1435    {
1436        // KDF output buffer
1437        // It contains parameters for the CFB encryption
1438        BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
1439
1440        // Symmetric key size in bytes
1441        UINT16           keySize = (keySizeInBits + 7) / 8;
1442
1443        TPM2B_IV         iv;
1444
1445        iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
1446        // See if there is any encryption to do
1447        if(iv.t.size > 0)
1448        {
1449            // Generate key and iv
1450            CryptKDFa(hash, key, "CFB", nonceTpm, nonceCaller,
1451                      keySizeInBits + (iv.t.size * 8), symParmString, NULL);
1452
1453            MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size,
1454                       sizeof(iv.t.buffer));
1455
1456            CryptSymmetricEncrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB,
1457                                  symParmString, &iv, dataSize, data);
1458        }
1459        return;
1460    }
```

10.2.7.3 CryptGenerateNewSymmetric()

This function creates the sensitive symmetric values for an HMAC or symmetric key. If the sensitive area is zero, then the sensitive creation key data is copied. If it is not zero, then the TPM will generate a random value of the selected size.

```
1461  void
1462  CryptGenerateNewSymmetric(
1463      TPMS_SENSITIVE_CREATE   *sensitiveCreate,    // IN: sensitive creation data
1464      TPMT_SENSITIVE          *sensitive,          // OUT: sensitive area
1465      TPM_ALG_ID               hashAlg,            // IN: hash algorithm for the KDF
1466      TPM2B_SEED              *seed,               // IN: seed used in creation
1467      TPM2B_NAME              *name                // IN: name of the object
1468      )
1469  {
1470      // This function is called to create a key and obfuscation value for a
1471      // symmetric key that can either be a block cipher or an XOR key. The buffer
1472      // in sensitive->sensitive will hold either. When we call the function
1473      // to copy the input value or generated value to the sensitive->sensitive
1474      // buffer we will need to have a size for the output buffer. This define
1475      // computes the maximum that it might need to be and uses that. It will always
1476      // be smaller than the largest value that will fit.
1477      #define MAX_SENSITIVE_SIZE                                                  \
1478          (MAX(sizeof(sensitive->sensitive.bits.t.buffer),                        \
1479              sizeof(sensitive->sensitive.sym.t.buffer)))
1480
1481      // set the size of the obfuscation value
1482      sensitive->seedValue.t.size = CryptGetHashDigestSize(hashAlg);
1483
1484      // If the input sensitive size is zero, then create both the sensitive data
1485      // and the obfuscation value
1486      if(sensitiveCreate->data.t.size == 0)
1487      {
1488          BYTE                    symValues[MAX(MAX_DIGEST_SIZE, MAX_SYM_KEY_BYTES)
1489                                            + MAX_DIGEST_SIZE];
1490          UINT16                  requestSize;
1491
1492          // Set the size of the request to be the size of the key and the
1493          // obfuscation value
1494          requestSize =   sensitive->sensitive.sym.t.size
1495                        + sensitive->seedValue.t.size;
1496          pAssert(requestSize <= sizeof(symValues));
1497
1498          requestSize = _cpri__GenerateSeededRandom(requestSize, symValues, hashAlg,
1499                                                    &seed->b,
1500                                                    "symmetric sensitive", &name->b,
1501                                                    NULL);
1502          pAssert(requestSize != 0);
1503
1504          // Copy the new key
1505          MemoryCopy(sensitive->sensitive.sym.t.buffer,
1506                     symValues, sensitive->sensitive.sym.t.size,
1507                     MAX_SENSITIVE_SIZE);
1508
1509          // copy the obfuscation value
1510          MemoryCopy(sensitive->seedValue.t.buffer,
1511                     &symValues[sensitive->sensitive.sym.t.size],
1512                     sensitive->seedValue.t.size,
1513                     sizeof(sensitive->seedValue.t.buffer));
1514      }
1515      else
1516      {
1517          // Copy input symmetric key to sensitive area as long as it will fit
1518          MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
1519                       MAX_SENSITIVE_SIZE);
```

```
1520
1521            // Create the obfuscation value
1522            _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
1523                                        sensitive->seedValue.t.buffer,
1524                                        hashAlg, &seed->b,
1525                                        "symmetric obfuscation", &name->b, NULL);
1526        }
1527        return;
1528    }
```

10.2.7.4 CryptGenerateKeySymmetric()

This function derives a symmetric cipher key from the provided seed.

Error Returns	Meaning
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area

```
1529    static TPM_RC
1530    CryptGenerateKeySymmetric(
1531        TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
1532                                                    //     for the new key.
1533        TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
1534        TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
1535        TPM_ALG_ID               hashAlg,           // IN: hash algorithm for the KDF
1536        TPM2B_SEED              *seed,              // IN: seed used in creation
1537        TPM2B_NAME              *name               // IN: name of the object
1538        )
1539    {
1540        // If this is not a new key, then the provided key data must be the right size
1541        if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR)
1542        {
1543            if(     (sensitiveCreate->data.t.size * 8)
1544                !=  publicArea->parameters.symDetail.sym.keyBits.sym)
1545                return TPM_RC_KEY_SIZE;
1546            // Make sure that the key size is OK.
1547            // This implementation only supports symmetric key sizes that are
1548            // multiples of 8
1549            if(publicArea->parameters.symDetail.sym.keyBits.sym % 8 != 0)
1550                return TPM_RC_KEY_SIZE;
1551        }
1552        else
1553        {
1554            // TPM is going to generate the key so set the size
1555            sensitive->sensitive.sym.t.size
1556                = publicArea->parameters.symDetail.sym.keyBits.sym / 8;
1557            sensitiveCreate->data.t.size = 0;
1558        }
1559        // Fill in the sensitive area
1560        CryptGenerateNewSymmetric(sensitiveCreate, sensitive, hashAlg,
1561                                  seed, name);
1562
1563        // Create unique area in public
1564        CryptComputeSymmetricUnique(publicArea->nameAlg,
1565                                    sensitive, &publicArea->unique.sym);
1566
1567        return TPM_RC_SUCCESS;
1568    }
```


10.2.7.5 CryptXORObfuscation()

This function implements XOR obfuscation. It should not be called if the hash algorithm is not implemented. The only return value from this function is TPM_RC_SUCCESS.

```
1569  #ifdef TPM_ALG_KEYEDHASH //% 5
1570  void
1571  CryptXORObfuscation(
1572      TPM_ALG_ID       hash,          // IN: hash algorithm for KDF
1573      TPM2B           *key,           // IN: KDF key
1574      TPM2B           *contextU,      // IN: contextU
1575      TPM2B           *contextV,      // IN: contextV
1576      UINT32           dataSize,      // IN: size of data buffer
1577      BYTE            *data           // IN/OUT: data to be XORed in place
1578      )
1579  {
1580      BYTE             mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
1581      BYTE            *pm;
1582      UINT32           i;
1583      UINT32           counter = 0;
1584      UINT16           hLen = CryptGetHashDigestSize(hash);
1585      UINT32           requestSize = dataSize * 8;
1586      INT32            remainBytes = (INT32) dataSize;
1587
1588      pAssert((key != NULL) && (data != NULL) && (hLen != 0));
1589
1590      // Call KDFa to generate XOR mask
1591      for(; remainBytes > 0; remainBytes -= hLen)
1592      {
1593          // Make a call to KDFa to get next iteration
1594          CryptKDFaOnce(hash, key, "XOR", contextU, contextV,
1595                        requestSize, mask, &counter);
1596
1597          // XOR next piece of the data
1598          pm = mask;
1599          for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
1600              *data++ ^= *pm++;
1601      }
1602      return;
1603  }
1604  #endif //TPM_ALG_KEYED_HASH //%5
```

10.2.8 Initialization and shut down

10.2.8.1 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.

```
1605  void
1606  CryptInitUnits(
1607      void
1608      )
1609  {
1610      // Initialize the vector of implemented algorithms
1611      AlgorithmGetImplementedVector(&g_implementedAlgorithms);
1612
1613      // Indicate that all test are necessary
1614      CryptInitializeToTest();
```

```
1615
1616        // Call crypto engine unit initialization
1617        // It is assumed that crypt engine initialization should always succeed.
1618        // Otherwise, TPM should go to failure mode.
1619        if(_cpri__InitCryptoUnits(&TpmFail) != CRYPT_SUCCESS)
1620            FAIL(FATAL_ERROR_INTERNAL);
1621        return;
1622    }
```

10.2.8.2 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.

```
1623    void
1624    CryptStopUnits(
1625        void
1626        )
1627    {
1628        // Call crypto engine unit stopping
1629        _cpri__StopCryptoUnits();
1630
1631        return;
1632    }
```

10.2.8.3 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.

```
1633    BOOL
1634    CryptUtilStartup(
1635        STARTUP_TYPE     type           // IN: the startup type
1636        )
1637    {
1638        // Make sure that the crypto library functions are ready.
1639        // NOTE: need to initialize the crypto before loading
1640        // the RND state may trigger a self-test which
1641        // uses the
1642        if( !_cpri__Startup())
1643            return FALSE;
1644
1645        // Initialize the state of the RNG.
1646        CryptDrbgGetPutState(PUT_STATE);
1647
1648        if(type == SU_RESET)
1649        {
1650    #ifdef TPM_ALG_ECC
1651            // Get a new  random commit nonce
1652            gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
1653            _cpri__GenerateRandom(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
1654            // Reset the counter and commit array
1655            gr.commitCounter = 0;
1656            MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
1657    #endif // TPM_ALG_ECC
1658        }
```

```
1659
1660        // If the shutdown was orderly, then the values recovered from NV will
1661        // be OK to use. If the shutdown was not orderly, then a TPM Reset was required
1662        // and we would have initialized in the code above.
1663
1664        return TRUE;
1665    }
```

10.2.9 Algorithm-Independent Functions

10.2.9.1 Introduction

These functions are used generically when a function of a general type (e.g., symmetric encryption) is required. The functions will modify the parameters as required to interface to the indicated algorithms.

10.2.9.2 CryptIsAsymAlgorithm()

This function indicates if an algorithm is an asymmetric algorithm.

Return Value	Meaning
TRUE	if it is an asymmetric algorithm
FALSE	if it is not an asymmetric algorithm

```
1666    BOOL
1667    CryptIsAsymAlgorithm(
1668        TPM_ALG_ID       algID          // IN: algorithm ID
1669        )
1670    {
1671        return (
1672    #ifdef TPM_ALG_RSA
1673                algID ==  TPM_ALG_RSA
1674    #endif
1675    #if defined TPM_ALG_RSA && defined TPM_ALG_ECC
1676                ||
1677    #endif
1678    #ifdef TPM_ALG_ECC
1679                algID == TPM_ALG_ECC
1680    #endif
1681              );
1682    }
```

10.2.9.3 CryptGetSymmetricBlockSize()

This function returns the size in octets of the symmetric encryption block used by an algorithm and key size combination.

```
1683    INT16
1684    CryptGetSymmetricBlockSize(
1685        TPMI_ALG_SYM     algorithm,     // IN: symmetric algorithm
1686        UINT16           keySize        // IN: key size in bit
1687        )
1688    {
1689        return _cpri__GetSymmetricBlockSize(algorithm, keySize);
1690    }
```

10.2.9.4 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.

```
1691  void
1692  CryptSymmetricEncrypt(
1693      BYTE                *encrypted,     // OUT: the encrypted data
1694      TPM_ALG_ID           algorithm,     // IN: algorithm for encryption
1695      UINT16               keySizeInBits, // IN: key size in bit
1696      TPMI_ALG_SYM_MODE    mode,          // IN: symmetric encryption mode
1697      BYTE                *key,           // IN: encryption key
1698      TPM2B_IV            *ivIn,          // IN/OUT: Input IV and output chaining
1699                                          //     value for the next block
1700      UINT32               dataSize,      // IN: data size in byte
1701      BYTE                *data           // IN/OUT: data buffer
1702      )
1703  {
1704
1705      TPM2B_IV             defaultIv = {0};
1706      TPM2B_IV            *iv = (ivIn != NULL) ? ivIn : &defaultIv;
1707
1708     TEST(algorithm);
1709
1710      pAssert(encrypted != NULL && key != NULL);
1711
1712      // this check can pass but the case below can fail. ALG_xx_VALUE values are
1713      // defined for all algorithms but the TPM_ALG_xx might not be.
1714      if(algorithm == ALG_AES_VALUE || algorithm == ALG_SM4_VALUE)
1715      {
1716          if(mode != TPM_ALG_ECB)
1717              defaultIv.t.size = 16;
1718          // A provided IV has to be the right size
1719          pAssert(mode == TPM_ALG_ECB || iv->t.size == 16);
1720      }
1721      switch(algorithm)
1722      {
1723  #ifdef TPM_ALG_AES
1724          case TPM_ALG_AES:
1725          {
1726              switch (mode)
1727              {
1728                  case TPM_ALG_CTR:
1729                      _cpri__AESEncryptCTR(encrypted, keySizeInBits, key,
1730                                           iv->t.buffer, dataSize, data);
1731                      break;
1732                  case TPM_ALG_OFB:
1733                      _cpri__AESEncryptOFB(encrypted, keySizeInBits, key,
1734                                           iv->t.buffer, dataSize, data);
1735                      break;
1736                  case TPM_ALG_CBC:
1737                      _cpri__AESEncryptCBC(encrypted, keySizeInBits, key,
1738                                           iv->t.buffer, dataSize, data);
1739                      break;
1740                  case TPM_ALG_CFB:
1741                      _cpri__AESEncryptCFB(encrypted, keySizeInBits, key,
1742                                           iv->t.buffer, dataSize, data);
1743                      break;
1744                  case TPM_ALG_ECB:
1745                      _cpri__AESEncryptECB(encrypted, keySizeInBits, key,
1746                                           dataSize, data);
1747                      break;
1748                  default:
1749                      pAssert(0);
1750              }
```

```
1751            }
1752            break;
1753    #endif
1754    #ifdef TPM_ALG_SM4
1755            case TPM_ALG_SM4:
1756            {
1757                switch (mode)
1758                {
1759                    case TPM_ALG_CTR:
1760                        _cpri__SM4EncryptCTR(encrypted, keySizeInBits, key,
1761                                             iv->t.buffer, dataSize, data);
1762                        break;
1763                    case TPM_ALG_OFB:
1764                        _cpri__SM4EncryptOFB(encrypted, keySizeInBits, key,
1765                                             iv->t.buffer, dataSize, data);
1766                        break;
1767                    case TPM_ALG_CBC:
1768                        _cpri__SM4EncryptCBC(encrypted, keySizeInBits, key,
1769                                             iv->t.buffer, dataSize, data);
1770                        break;
1771
1772                    case TPM_ALG_CFB:
1773                        _cpri__SM4EncryptCFB(encrypted, keySizeInBits, key,
1774                                             iv->t.buffer, dataSize, data);
1775                        break;
1776                    case TPM_ALG_ECB:
1777                        _cpri__SM4EncryptECB(encrypted, keySizeInBits, key,
1778                                             dataSize, data);
1779                        break;
1780                    default:
1781                        pAssert(0);
1782                }
1783            }
1784            break;
1785
1786    #endif
1787            default:
1788                pAssert(FALSE);
1789                break;
1790        }
1791
1792        return;
1793
1794    }
```

10.2.9.5 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.

```
1795    void
1796    CryptSymmetricDecrypt(
1797        BYTE                *decrypted,
1798        TPM_ALG_ID           algorithm,     // IN: algorithm for encryption
1799        UINT16               keySizeInBits, // IN: key size in bit
1800        TPMI_ALG_SYM_MODE    mode,          // IN: symmetric encryption mode
1801        BYTE                *key,           // IN: encryption key
1802        TPM2B_IV            *ivIn,          // IN/OUT: IV for next block
1803        UINT32               dataSize,      // IN: data size in byte
1804        BYTE                *data           // IN/OUT: data buffer
1805        )
1806    {
1807        BYTE                *iv = NULL;
1808        BYTE                 defaultIV[sizeof(TPMT_HA)];
```

```
1809
1810        TEST(algorithm);
1811
1812        if(
1813    #ifdef TPM_ALG_AES
1814              algorithm == TPM_ALG_AES
1815    #endif
1816    #if defined TPM_ALG_AES && defined TPM_ALG_SM4
1817           ||
1818    #endif
1819    #ifdef  TPM_ALG_SM4
1820              algorithm == TPM_ALG_SM4
1821    #endif
1822          )
1823        {
1824            // Both SM4 and AES have block size of 128 bits
1825            // If the iv is not provided, create a default of 0
1826            if(ivIn == NULL)
1827            {
1828                // Initialize the default IV
1829                iv = defaultIV;
1830                MemorySet(defaultIV, 0, 16);
1831            }
1832            else
1833            {
1834                // A provided IV has to be the right size
1835                pAssert(mode == TPM_ALG_ECB || ivIn->t.size == 16);
1836                iv = &(ivIn->t.buffer[0]);
1837            }
1838        }
1839
1840        switch(algorithm)
1841        {
1842    #ifdef TPM_ALG_AES
1843        case TPM_ALG_AES:
1844        {
1845
1846            switch (mode)
1847            {
1848                case TPM_ALG_CTR:
1849                    _cpri__AESDecryptCTR(decrypted, keySizeInBits, key, iv,
1850                                         dataSize, data);
1851                    break;
1852                case TPM_ALG_OFB:
1853                    _cpri__AESDecryptOFB(decrypted, keySizeInBits, key, iv,
1854                                         dataSize, data);
1855                    break;
1856                case TPM_ALG_CBC:
1857                    _cpri__AESDecryptCBC(decrypted, keySizeInBits, key, iv,
1858                                         dataSize, data);
1859                    break;
1860                case TPM_ALG_CFB:
1861                    _cpri__AESDecryptCFB(decrypted, keySizeInBits, key, iv,
1862                                         dataSize, data);
1863                    break;
1864                case TPM_ALG_ECB:
1865                    _cpri__AESDecryptECB(decrypted, keySizeInBits, key,
1866                                         dataSize, data);
1867                    break;
1868                default:
1869                    pAssert(0);
1870            }
1871            break;
1872        }
1873    #endif //TPM_ALG_AES
1874    #ifdef TPM_ALG_SM4
```

```
1875        case TPM_ALG_SM4 :
1876            switch (mode)
1877            {
1878                case TPM_ALG_CTR:
1879                    _cpri__SM4DecryptCTR(decrypted, keySizeInBits, key, iv,
1880                                         dataSize, data);
1881                    break;
1882                case TPM_ALG_OFB:
1883                    _cpri__SM4DecryptOFB(decrypted, keySizeInBits, key, iv,
1884                                         dataSize, data);
1885                    break;
1886                case TPM_ALG_CBC:
1887                    _cpri__SM4DecryptCBC(decrypted, keySizeInBits, key, iv,
1888                                         dataSize, data);
1889                    break;
1890                case TPM_ALG_CFB:
1891                    _cpri__SM4DecryptCFB(decrypted, keySizeInBits, key, iv,
1892                                         dataSize, data);
1893                    break;
1894                case TPM_ALG_ECB:
1895                    _cpri__SM4DecryptECB(decrypted, keySizeInBits, key,
1896                                         dataSize, data);
1897                    break;
1898                default:
1899                    pAssert(0);
1900            }
1901            break;
1902    #endif //TPM_ALG_SM4
1903
1904        default:
1905            pAssert(FALSE);
1906            break;
1907        }
1908        return;
1909    }
```

10.2.9.6 CryptSecretEncrypt()

This function creates a secret value and its associated secret structure using an asymmetric algorithm.

This function is used by TPM2_Rewrap() TPM2_MakeCredential(), and TPM2_Duplicate().

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a valid decryption key
TPM_RC_KEY	invalid ECC key (public point is not on the curve)
TPM_RC_SCHEME	RSA key with an unsupported padding scheme
TPM_RC_VALUE	numeric value of the data to be decrypted is greater than the RSA key modulus

```
1910    TPM_RC
1911    CryptSecretEncrypt(
1912        TPMI_DH_OBJECT           keyHandle,     // IN: encryption key handle
1913        const char              *label,         // IN: a null-terminated string as L
1914        TPM2B_DATA              *data,          // OUT: secret value
1915        TPM2B_ENCRYPTED_SECRET  *secret         // OUT: secret structure
1916        )
1917    {
1918        TPM_RC       result = TPM_RC_SUCCESS;
1919        OBJECT      *encryptKey = ObjectGet(keyHandle);  // TPM key used for encrypt
1920
1921        pAssert(data != NULL && secret != NULL);
```

```
1922
1923        // The output secret value has the size of the digest produced by the nameAlg.
1924        data->t.size = CryptGetHashDigestSize(encryptKey->publicArea.nameAlg);
1925
1926        pAssert(encryptKey->publicArea.objectAttributes.decrypt == SET);
1927
1928        switch(encryptKey->publicArea.type)
1929        {
1930    #ifdef TPM_ALG_RSA
1931            case TPM_ALG_RSA:
1932            {
1933                TPMT_RSA_DECRYPT            scheme;
1934
1935                // Use OAEP scheme
1936                scheme.scheme = TPM_ALG_OAEP;
1937                scheme.details.oaep.hashAlg = encryptKey->publicArea.nameAlg;
1938
1939                // Create secret data from RNG
1940                CryptGenerateRandom(data->t.size, data->t.buffer);
1941
1942                // Encrypt the data by RSA OAEP into encrypted secret
1943                result = CryptEncryptRSA(&secret->t.size, secret->t.secret,
1944                                         encryptKey, &scheme,
1945                                         data->t.size, data->t.buffer, label);
1946            }
1947            break;
1948    #endif //TPM_ALG_RSA
1949
1950    #ifdef TPM_ALG_ECC
1951            case TPM_ALG_ECC:
1952            {
1953                TPMS_ECC_POINT      eccPublic;
1954                TPM2B_ECC_PARAMETER eccPrivate;
1955                TPMS_ECC_POINT      eccSecret;
1956                BYTE                *buffer = secret->t.secret;
1957
1958                // Need to make sure that the public point of the key is on the
1959                // curve defined by the key.
1960                if(!_cpri__EccIsPointOnCurve(
1961                           encryptKey->publicArea.parameters.eccDetail.curveID,
1962                           &encryptKey->publicArea.unique.ecc))
1963                    result = TPM_RC_KEY;
1964                else
1965                {
1966
1967                    // Call crypto engine to create an auxiliary ECC key
1968                    // We assume crypt engine initialization should always success.
1969                    // Otherwise, TPM should go to failure mode.
1970                    CryptNewEccKey(encryptKey->publicArea.parameters.eccDetail.curveID,
1971                                   &eccPublic, &eccPrivate);
1972
1973                    // Marshal ECC public to secret structure. This will be used by the
1974                    // recipient to decrypt the secret with their private key.
1975                    secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
1976
1977                    // Compute ECDH shared secret which is R = [d]Q where d is the
1978                    // private part of the ephemeral key and Q is the public part of a
1979                    // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
1980                    // because the auxiliary ECC key is just created according to the
1981                    // parameters of input ECC encrypt key.
1982                    if(    CryptEccPointMultiply(&eccSecret,
1983                                   encryptKey->publicArea.parameters.eccDetail.curveID,
1984                                   &eccPrivate,
1985                                   &encryptKey->publicArea.unique.ecc)
1986                       != CRYPT_SUCCESS)
1987                        result = TPM_RC_KEY;
```

```
1988                else
1989
1990                    // The secret value is computed from Z using KDFe as:
1991                    // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
1992                    // Where:
1993                    //  HashID  the nameAlg of the decrypt key
1994                    //  Z    the x coordinate (Px) of the product (P) of the point
1995                    //       (Q) of the secret and the private x coordinate (de,V)
1996                    //       of the decryption key
1997                    //  Use a null-terminated string containing "SECRET"
1998                    //  PartyUInfo  the x coordinate of the point in the secret
1999                    //              (Qe,U )
2000                    //  PartyVInfo  the x coordinate of the public key (Qs,V )
2001                    //  bits    the number of bits in the digest of HashID
2002                    // Retrieve seed from KDFe
2003
2004                    CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
2005                              label, &eccPublic.x.b,
2006                              &encryptKey->publicArea.unique.ecc.x.b,
2007                              data->t.size * 8, data->t.buffer);
2008            }
2009        }
2010        break;
2011 #endif //TPM_ALG_ECC
2012
2013     default:
2014         FAIL(FATAL_ERROR_INTERNAL);
2015         break;
2016     }
2017
2018     return result;
2019 }
```

10.2.9.7 CryptSecretDecrypt()

Decrypt a secret value by asymmetric (or symmetric) algorithm This function is used for ActivateCredential() and Import for asymmetric decryption, and StartAuthSession() for both asymmetric and symmetric decryption process

Error Returns	Meaning
TPM_RC_ATTRIBUTES	RSA key is not a decryption key
TPM_RC_BINDING	Invalid RSA key (public and private parts are not cryptographically bound.
TPM_RC_ECC_POINT	ECC point in the secret is not on the curve
TPM_RC_INSUFFICIENT	failed to retrieve ECC point from the secret
TPM_RC_NO_RESULT	multiplication resulted in ECC point at infinity
TPM_RC_SIZE	data to decrypt is not of the same size as RSA key
TPM_RC_VALUE	For RSA key, numeric value of the encrypted data is greater than the modulus, or the recovered data is larger than the output buffer. For *keyedHash* or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.
TPM_RC_FAILURE	internal error

```
2020 TPM_RC
2021 CryptSecretDecrypt(
2022     TPM_HANDLE              tpmKey,         // IN: decrypt key
2023     TPM2B_NONCE            *nonceCaller,    // IN: nonceCaller.  It is needed for
2024                                             //     symmetric decryption.  For
```

```
2025                                                //     asymmetric decryption, this
2026                                                //     parameter is NULL
2027      const char              *label,          // IN: a null-terminated string as L
2028      TPM2B_ENCRYPTED_SECRET  *secret,         // IN: input secret
2029      TPM2B_DATA              *data            // OUT: decrypted secret value
2030      )
2031  {
2032      TPM_RC      result = TPM_RC_SUCCESS;
2033      OBJECT      *decryptKey = ObjectGet(tpmKey);   //TPM key used for decrypting
2034
2035      // Decryption for secret
2036      switch(decryptKey->publicArea.type)
2037      {
2038
2039  #ifdef TPM_ALG_RSA
2040          case TPM_ALG_RSA:
2041          {
2042              TPMT_RSA_DECRYPT        scheme;
2043
2044              // Use OAEP scheme
2045              scheme.scheme = TPM_ALG_OAEP;
2046              scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
2047
2048              // Set the output buffer capacity
2049              data->t.size = sizeof(data->t.buffer);
2050
2051              // Decrypt seed by RSA OAEP
2052              result = CryptDecryptRSA(&data->t.size, data->t.buffer, decryptKey,
2053                                       &scheme,
2054                                       secret->t.size, secret->t.secret,label);
2055              if(   (result == TPM_RC_SUCCESS)
2056                 && (data->t.size
2057                      > CryptGetHashDigestSize(decryptKey->publicArea.nameAlg)))
2058                  result = TPM_RC_VALUE;
2059          }
2060          break;
2061  #endif //TPM_ALG_RSA
2062
2063  #ifdef TPM_ALG_ECC
2064          case TPM_ALG_ECC:
2065          {
2066              TPMS_ECC_POINT       eccPublic;
2067              TPMS_ECC_POINT       eccSecret;
2068              BYTE                *buffer = secret->t.secret;
2069              INT32                size = secret->t.size;
2070
2071              // Retrieve ECC point from secret buffer
2072              result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
2073              if(result == TPM_RC_SUCCESS)
2074              {
2075                  result = CryptEccPointMultiply(&eccSecret,
2076                                 decryptKey->publicArea.parameters.eccDetail.curveID,
2077                                 &decryptKey->sensitive.sensitive.ecc,
2078                                 &eccPublic);
2079
2080                  if(result == TPM_RC_SUCCESS)
2081                  {
2082
2083                      // Set the size of the "recovered" secret value to be the size
2084                      // of the digest produced by the nameAlg.
2085                      data->t.size =
2086                              CryptGetHashDigestSize(decryptKey->publicArea.nameAlg);
2087
2088                      // The secret value is computed from Z using KDFe as:
2089                      // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
2090                      // Where:
```

```
2091                    //  HashID -- the nameAlg of the decrypt key
2092                    //  Z --  the x coordinate (Px) of the product (P) of the point
2093                    //        (Q) of the secret and the private x coordinate (de,V)
2094                    //        of the decryption key
2095                    //  Use -- a null-terminated string containing "SECRET"
2096                    //  PartyUInfo -- the x coordinate of the point in the secret
2097                    //              (Qe,U )
2098                    //  PartyVInfo -- the x coordinate of the public key (Qs,V )
2099                    //  bits -- the number of bits in the digest of HashID
2100                    // Retrieve seed from KDFe
2101                    CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
2102                              &eccPublic.x.b,
2103                              &decryptKey->publicArea.unique.ecc.x.b,
2104                              data->t.size * 8, data->t.buffer);
2105                }
2106            }
2107        }
2108        break;
2109  #endif //TPM_ALG_ECC
2110
2111        case TPM_ALG_KEYEDHASH:
2112            // The seed size can not be bigger than the digest size of nameAlg
2113            if(secret->t.size >
2114                    CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
2115                result = TPM_RC_VALUE;
2116            else
2117            {
2118                // Retrieve seed by XOR Obfuscation:
2119                //    seed = XOR(secret, hash, key, nonceCaller, nullNonce)
2120                //    where:
2121                //    secret  the secret parameter from the TPM2_StartAuthHMAC
2122                //            command
2123                //            which contains the seed value
2124                //    hash    nameAlg  of tpmKey
2125                //    key     the key or data value in the object referenced by
2126                //            entityHandle in the TPM2_StartAuthHMAC command
2127                //    nonceCaller the parameter from the TPM2_StartAuthHMAC command
2128                //    nullNonce   a zero-length nonce
2129                // XOR Obfuscation in place
2130                CryptXORObfuscation(decryptKey->publicArea.nameAlg,
2131                                    &decryptKey->sensitive.sensitive.bits.b,
2132                                    &nonceCaller->b, NULL,
2133                                    secret->t.size, secret->t.secret);
2134                // Copy decrypted seed
2135                MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
2136            }
2137            break;
2138        case TPM_ALG_SYMCIPHER:
2139            {
2140                TPM2B_IV                iv = {0};
2141                TPMT_SYM_DEF_OBJECT     *symDef;
2142                // The seed size can not be bigger than the digest size of nameAlg
2143                if(secret->t.size >
2144                        CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
2145                    result = TPM_RC_VALUE;
2146                else
2147                {
2148                    symDef = &decryptKey->publicArea.parameters.symDetail.sym;
2149                    iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
2150                                                           symDef->keyBits.sym);
2151                    pAssert(iv.t.size != 0);
2152                    if(nonceCaller->t.size >= iv.t.size)
2153                        MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size,
2154                                   sizeof(iv.t.buffer));
2155                    else
2156                        MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
```

```
                                        nonceCaller->t.size, sizeof(iv.t.buffer));
                    // 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, sizeof(data->t.buffer));
                }
            }
            break;
        default:
            pAssert(0);
            break;
    }
    return result;
}
```

10.2.9.8 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```
void
CryptParameterEncryption(
    TPM_HANDLE       handle,            // IN: encrypt session handle
    TPM2B           *nonceCaller,       // IN: nonce caller
    UINT16           leadingSizeInByte, // IN: the size of the leading size field in
                                        //     byte
    TPM2B_AUTH      *extraKey,          // IN: additional key material other than
                                        //     session auth
    BYTE            *buffer             // IN/OUT: parameter buffer to be encrypted
    )
{
    SESSION     *session = SessionGet(handle);  // encrypt session
    TPM2B_TYPE(SYM_KEY, (  sizeof(extraKey->t.buffer)
                         + sizeof(session->sessionKey.t.buffer)));
    TPM2B_SYM_KEY        key;               // encryption key
    UINT32               cipherSize = 0;    // size of cipher text

    pAssert(session->sessionKey.t.size + extraKey->t.size <= 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];
    }
#ifdef      TPM4B
    else if(leadingSizeInByte == 4)
    {
        // use the first four bytes to indicate the number of bytes to encrypt
        cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
        //advance pointer
        buffer = &buffer[4];
    }
#endif
    else
    {
        pAssert(FALSE);
    }
```

```
2216
2217        // Compute encryption key by concatenating sessionAuth with extra key
2218        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
2219        MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
2220
2221        if (session->symmetric.algorithm == TPM_ALG_XOR)
2222
2223            // XOR parameter encryption formulation:
2224            //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
2225            CryptXORObfuscation(session->authHashAlg, &(key.b),
2226                                &(session->nonceTPM.b),
2227                                nonceCaller, cipherSize, buffer);
2228        else
2229            ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
2230                           session->symmetric.keyBits.aes, &(key.b),
2231                           nonceCaller, &(session->nonceTPM.b),
2232                           cipherSize, buffer);
2233        return;
2234    }
```

10.2.9.9 CryptParameterDecryption()

This function does in-place decryption of a command parameter.

Error Returns	Meaning
TPM_RC_SIZE	The number of bytes in the input buffer is less than the number of bytes to be decrypted.

```
2235    TPM_RC
2236    CryptParameterDecryption(
2237        TPM_HANDLE       handle,            // IN: encrypted session handle
2238        TPM2B           *nonceCaller,       // IN: nonce caller
2239        UINT32           bufferSize,        // IN: size of parameter buffer
2240        UINT16           leadingSizeInByte, // IN: the size of the leading size field in
2241                                            //     byte
2242        TPM2B_AUTH      *extraKey,          // IN: the authValue
2243        BYTE            *buffer             // IN/OUT: parameter buffer to be decrypted
2244        )
2245    {
2246        SESSION         *session = SessionGet(handle);  // encrypt session
2247        // The HMAC key is going to be the concatenation of the session key and any
2248        // additional key material (like the authValue). The size of both of these
2249        // is the size of the buffer which can contain a TPMT_HA.
2250        TPM2B_TYPE(HMAC_KEY, (  sizeof(extraKey->t.buffer)
2251                              + sizeof(session->sessionKey.t.buffer)));
2252        TPM2B_HMAC_KEY          key;            // decryption key
2253        UINT32                  cipherSize = 0; // size of cipher text
2254
2255        pAssert(session->sessionKey.t.size + extraKey->t.size <= sizeof(key.t.buffer));
2256
2257        // Retrieve encrypted data size.
2258        if(leadingSizeInByte == 2)
2259        {
2260            // The first two bytes of the buffer are the size of the
2261            // data to be decrypted
2262            cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
2263            buffer = &buffer[2];   // advance the buffer
2264        }
2265    #ifdef  TPM4B
2266        else if(leadingSizeInByte == 4)
2267        {
2268            // the leading size is four bytes so get the four byte size field
2269            cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
```

```
2270            buffer = &buffer[4];   //advance pointer
2271        }
2272    #endif
2273        else
2274        {
2275            pAssert(FALSE);
2276        }
2277        if(cipherSize > bufferSize)
2278            return TPM_RC_SIZE;
2279
2280        // Compute decryption key by concatenating sessionAuth with extra input key
2281        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
2282        MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
2283
2284        if(session->symmetric.algorithm == TPM_ALG_XOR)
2285            // XOR parameter decryption formulation:
2286            //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
2287            // Call XOR obfuscation function
2288            CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
2289                                &(session->nonceTPM.b), cipherSize, buffer);
2290        else
2291            // Assume that it is one of the symmetric block ciphers.
2292            ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
2293                           session->symmetric.keyBits.sym,
2294                           &key.b, nonceCaller, &session->nonceTPM.b,
2295                           cipherSize, buffer);
2296
2297        return TPM_RC_SUCCESS;
2298
2299    }
```

10.2.9.10 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```
2300    void
2301    CryptComputeSymmetricUnique(
2302        TPMI_ALG_HASH    nameAlg,       // IN: object name algorithm
2303        TPMT_SENSITIVE  *sensitive,     // IN: sensitive area
2304        TPM2B_DIGEST    *unique         // OUT: unique buffer
2305        )
2306    {
2307        HASH_STATE  hashState;
2308
2309        pAssert(sensitive != NULL && unique != NULL);
2310
2311        // Compute the public value as the hash of sensitive.symkey || unique.buffer
2312        unique->t.size = CryptGetHashDigestSize(nameAlg);
2313        CryptStartHash(nameAlg, &hashState);
2314
2315        // Add obfuscation value
2316        CryptUpdateDigest2B(&hashState, &sensitive->seedValue.b);
2317
2318        // Add sensitive value
2319        CryptUpdateDigest2B(&hashState, &sensitive->sensitive.any.b);
2320
2321        CryptCompleteHash2B(&hashState, &unique->b);
2322
2323        return;
2324    }
2325    #if 0 //%
```

10.2.9.11 CryptComputeSymValue()

This function computes the *seedValue* field in asymmetric sensitive areas.

```
2326    void
2327    CryptComputeSymValue(
2328        TPM_HANDLE       parentHandle,  // IN: parent handle of the object to be created
2329        TPMT_PUBLIC     *publicArea,    // IN/OUT: the public area template
2330        TPMT_SENSITIVE  *sensitive,     // IN: sensitive area
2331        TPM2B_SEED      *seed,          // IN: the seed
2332        TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm for KDFa
2333        TPM2B_NAME      *name           // IN: object name
2334        )
2335    {
2336        TPM2B_AUTH   *proof = NULL;
2337
2338        if(CryptIsAsymAlgorithm(publicArea->type))
2339        {
2340            // Generate seedValue only when an asymmetric key is a storage key
2341            if(publicArea->objectAttributes.decrypt == SET
2342                    && publicArea->objectAttributes.restricted == SET)
2343            {
2344                // If this is a primary object in the endorsement hierarchy, use
2345                // ehProof in the creation of the symmetric seed so that child
2346                // objects in the endorsement hierarchy are voided on TPM2_Clear()
2347                // or TPM2_ChangeEPS()
2348                if(   parentHandle == TPM_RH_ENDORSEMENT
2349                   && publicArea->objectAttributes.fixedTPM == SET)
2350                    proof = &gp.ehProof;
2351            }
2352            else
2353            {
2354                sensitive->seedValue.t.size = 0;
2355                return;
2356            }
2357        }
2358
2359        // For all object types, the size of seedValue is the digest size of nameAlg
2360        sensitive->seedValue.t.size = CryptGetHashDigestSize(publicArea->nameAlg);
2361
2362        // Compute seedValue using implementation-dependent method
2363        _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
2364                                    sensitive->seedValue.t.buffer,
2365                                    hashAlg,
2366                                    &seed->b,
2367                                    "seedValue",
2368                                    &name->b,
2369                                    (TPM2B *)proof);
2370        return;
2371    }
2372    #endif //%
```

10.2.9.12 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.

Error Returns	Meaning
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area for a symmetric key
TPM_RC_RANGE	for an RSA key, the exponent is not supported
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme for a keyed hash object
TPM_RC_VALUE	exponent is not prime or could not find a prime using the provided parameters for an RSA key; unsupported name algorithm for an ECC key

```
2373    TPM_RC
2374    CryptCreateObject(
2375        TPM_HANDLE                 parentHandle,      // IN/OUT: indication of the seed
2376                                                      //     source
2377        TPMT_PUBLIC               *publicArea,        // IN/OUT: public area
2378        TPMS_SENSITIVE_CREATE     *sensitiveCreate,   // IN: sensitive creation
2379        TPMT_SENSITIVE            *sensitive          // OUT: sensitive area
2380        )
2381    {
2382        // Next value is a placeholder for a random seed that is used in
2383        // key creation when the parent is not a primary seed. It has the same
2384        // size as the primary seed.
2385
2386        TPM2B_SEED       localSeed;     // data to seed key creation if this
2387                                        // is not a primary seed
2388
2389        TPM2B_SEED      *seed = NULL;
2390        TPM_RC           result = TPM_RC_SUCCESS;
2391
2392        TPM2B_NAME       name;
2393        TPM_ALG_ID       hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
2394        OBJECT          *parent;
2395        UINT32           counter;
2396
2397        // Set the sensitive type for the object
2398        sensitive->sensitiveType = publicArea->type;
2399        ObjectComputeName(publicArea, &name);
2400
2401        // For all objects, copy the initial auth data
2402        sensitive->authValue = sensitiveCreate->userAuth;
2403
2404        // If this is a permanent handle assume that it is a hierarchy
2405        if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
2406        {
2407            seed = HierarchyGetPrimarySeed(parentHandle);
2408        }
2409        else
2410        {
2411            // If not hierarchy handle, get parent
2412            parent = ObjectGet(parentHandle);
2413            hashAlg = parent->publicArea.nameAlg;
2414
2415            // Use random value as seed for non-primary objects
2416            localSeed.t.size = PRIMARY_SEED_SIZE;
2417            CryptGenerateRandom(PRIMARY_SEED_SIZE, localSeed.t.buffer);
2418            seed = &localSeed;
2419        }
2420
2421        switch(publicArea->type)
2422        {
2423    #ifdef TPM_ALG_RSA
2424            // Create RSA key
```

```
2425        case TPM_ALG_RSA:
2426            result = CryptGenerateKeyRSA(publicArea, sensitive,
2427                                         hashAlg, seed, &name, &counter);
2428            break;
2429    #endif // TPM_ALG_RSA
2430
2431    #ifdef TPM_ALG_ECC
2432            // Create ECC key
2433        case TPM_ALG_ECC:
2434            result = CryptGenerateKeyECC(publicArea, sensitive,
2435                                         hashAlg, seed, &name, &counter);
2436            break;
2437    #endif // TPM_ALG_ECC
2438
2439            // Collect symmetric key information
2440        case TPM_ALG_SYMCIPHER:
2441            return CryptGenerateKeySymmetric(publicArea, sensitiveCreate,
2442                                             sensitive, hashAlg, seed, &name);
2443            break;
2444        case TPM_ALG_KEYEDHASH:
2445            return CryptGenerateKeyedHash(publicArea, sensitiveCreate,
2446                                          sensitive, hashAlg, seed, &name);
2447            break;
2448        default:
2449            pAssert(0);
2450            break;
2451        }
2452        if(result == TPM_RC_SUCCESS)
2453        {
2454            TPM2B_AUTH          *proof = NULL;
2455
2456            if(publicArea->objectAttributes.decrypt == SET
2457                    && publicArea->objectAttributes.restricted == SET)
2458            {
2459                // If this is a primary object in the endorsement hierarchy, use
2460                // ehProof in the creation of the symmetric seed so that child
2461                // objects in the endorsement hierarchy are voided on TPM2_Clear()
2462                // or TPM2_ChangeEPS()
2463                if(   parentHandle == TPM_RH_ENDORSEMENT
2464                   && publicArea->objectAttributes.fixedTPM == SET)
2465                    proof = &gp.ehProof;
2466
2467                // For all object types, the size of seedValue is the digest size
2468                // of its nameAlg
2469                sensitive->seedValue.t.size
2470                    = CryptGetHashDigestSize(publicArea->nameAlg);
2471
2472                // Compute seedValue using implementation-dependent method
2473                _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
2474                                            sensitive->seedValue.t.buffer,
2475                                            hashAlg,
2476                                            &seed->b,
2477                                            "seedValuea",
2478                                            &name.b,
2479                                            (TPM2B *)proof);
2480            }
2481            else
2482            {
2483                sensitive->seedValue.t.size = 0;
2484            }
2485        }
2486
2487        return result;
2488
2489    }
```

10.2.9.13 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

```
2490    BOOL
2491    CryptObjectIsPublicConsistent(
2492        TPMT_PUBLIC     *publicArea     // IN: public area
2493        )
2494    {
2495        BOOL                OK = TRUE;
2496        switch (publicArea->type)
2497        {
2498    #ifdef TPM_ALG_RSA
2499            case TPM_ALG_RSA:
2500                OK = CryptAreKeySizesConsistent(publicArea);
2501                break;
2502    #endif //TPM_ALG_RSA
2503
2504    #ifdef TPM_ALG_ECC
2505            case TPM_ALG_ECC:
2506                {
2507                    const ECC_CURVE                *curveValue;
2508
2509                    // Check that the public point is on the indicated curve.
2510                    OK = CryptEccIsPointOnCurve(
2511                                    publicArea->parameters.eccDetail.curveID,
2512                                    &publicArea->unique.ecc);
2513                    if(OK)
2514                    {
2515                        curveValue = CryptEccGetCurveDataPointer(
2516                                                publicArea->parameters.eccDetail.curveID);
2517                        pAssert(curveValue != NULL);
2518
2519                        // The input ECC curve must be a supported curve
2520                        // IF a scheme is defined for the curve, then that scheme must
2521                        // be used.
2522                        OK =    (curveValue->sign.scheme == TPM_ALG_NULL
2523                             || (   publicArea->parameters.eccDetail.scheme.scheme
2524                                 == curveValue->sign.scheme));
2525                        OK = OK && CryptAreKeySizesConsistent(publicArea);
2526                    }
2527                }
2528                break;
2529    #endif //TPM_ALG_ECC
2530
2531            default:
2532                // Symmetric object common checks
2533                // There is noting to check with a symmetric key that is public only.
2534                // Also not sure that there is anything useful to be done with it
2535                // either.
2536                break;
2537        }
2538        return OK;
2539    }
```

10.2.9.14 CryptObjectPublicPrivateMatch()

This function checks the cryptographic binding between the public and sensitive areas.

Error Returns	Meaning
TPM_RC_TYPE	the type of the public and private areas are not the same
TPM_RC_FAILURE	crypto error
TPM_RC_BINDING	the public and private areas are not cryptographically matched.

```
2540    TPM_RC
2541    CryptObjectPublicPrivateMatch(
2542        OBJECT          *object         // IN: the object to check
2543        )
2544    {
2545        TPMT_PUBLIC         *publicArea;
2546        TPMT_SENSITIVE      *sensitive;
2547        TPM_RC               result = TPM_RC_SUCCESS;
2548        BOOL                 isAsymmetric = FALSE;
2549
2550        pAssert(object != NULL);
2551        publicArea = &object->publicArea;
2552        sensitive = &object->sensitive;
2553        if(publicArea->type != sensitive->sensitiveType)
2554            return TPM_RC_TYPE;
2555
2556        switch(publicArea->type)
2557        {
2558    #ifdef TPM_ALG_RSA
2559        case TPM_ALG_RSA:
2560            isAsymmetric = TRUE;
2561            // The public and private key sizes need to be consistent
2562            if(sensitive->sensitive.rsa.t.size != publicArea->unique.rsa.t.size/2)
2563                result = TPM_RC_BINDING;
2564            else
2565            // Load key by computing the private exponent
2566                result = CryptLoadPrivateRSA(object);
2567            break;
2568    #endif
2569    #ifdef TPM_ALG_ECC
2570            // This function is called from ObjectLoad() which has already checked to
2571            // see that the public point is on the curve so no need to repeat that
2572            // check.
2573        case TPM_ALG_ECC:
2574            isAsymmetric = TRUE;
2575            if(   publicArea->unique.ecc.x.t.size
2576                    != sensitive->sensitive.ecc.t.size)
2577                result = TPM_RC_BINDING;
2578            else if(publicArea->nameAlg != TPM_ALG_NULL)
2579            {
2580                TPMS_ECC_POINT          publicToCompare;
2581                // Compute ECC public key
2582                CryptEccPointMultiply(&publicToCompare,
2583                                      publicArea->parameters.eccDetail.curveID,
2584                                      &sensitive->sensitive.ecc, NULL);
2585                // Compare ECC public key
2586                if(    (!Memory2BEqual(&publicArea->unique.ecc.x.b,
2587                                       &publicToCompare.x.b))
2588                   || (!Memory2BEqual(&publicArea->unique.ecc.y.b,
2589                                       &publicToCompare.y.b)))
2590                    result = TPM_RC_BINDING;
2591            }
2592            break;
```

```
2593    #endif
2594        case TPM_ALG_KEYEDHASH:
2595            break;
2596        case TPM_ALG_SYMCIPHER:
2597            if(   (publicArea->parameters.symDetail.sym.keyBits.sym + 7)/8
2598               != sensitive->sensitive.sym.t.size)
2599                result = TPM_RC_BINDING;
2600            break;
2601        default:
2602            // The choice here is an assert or a return of a bad type for the object
2603            pAssert(0);
2604            break;
2605        }
2606
2607        // For asymmetric keys, the algorithm for validating the linkage between
2608        // the public and private areas is algorithm dependent. For symmetric keys
2609        // the linkage is based on hashing the symKey and obfuscation values.
2610        if(   result == TPM_RC_SUCCESS && !isAsymmetric
2611           && publicArea->nameAlg != TPM_ALG_NULL)
2612        {
2613            TPM2B_DIGEST    uniqueToCompare;
2614
2615            // Compute unique for symmetric key
2616            CryptComputeSymmetricUnique(publicArea->nameAlg, sensitive,
2617                                        &uniqueToCompare);
2618            // Compare unique
2619            if(!Memory2BEqual(&publicArea->unique.sym.b,
2620                              &uniqueToCompare.b))
2621                result = TPM_RC_BINDING;
2622        }
2623        return result;
2624
2625    }
```

10.2.9.15 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

```
2626    TPMI_ALG_HASH
2627    CryptGetSignHashAlg(
2628        TPMT_SIGNATURE  *auth           // IN: signature
2629        )
2630    {
2631        pAssert(auth->sigAlg != TPM_ALG_NULL);
2632
2633        // Get authHash algorithm based on signing scheme
2634        switch(auth->sigAlg)
2635        {
2636
2637    #ifdef  TPM_ALG_RSA
2638            case TPM_ALG_RSASSA:
2639                return auth->signature.rsassa.hash;
2640
2641            case TPM_ALG_RSAPSS:
2642                return auth->signature.rsapss.hash;
2643
2644        #endif //TPM_ALG_RSA
2645
2646        #ifdef TPM_ALG_ECC
2647            case TPM_ALG_ECDSA:
2648                return auth->signature.ecdsa.hash;
2649
2650        #endif //TPM_ALG_ECC
```

```
2651
2652            case TPM_ALG_HMAC:
2653                return auth->signature.hmac.hashAlg;
2654
2655            default:
2656                return TPM_ALG_NULL;
2657        }
2658    }
```

10.2.9.16 CryptIsSplitSign()

This function us used to determine if the signing operation is a split signing operation that required a TPM2_Commit().

```
2659    BOOL
2660    CryptIsSplitSign(
2661        TPM_ALG_ID       scheme          // IN: the algorithm selector
2662        )
2663    {
2664        if(   scheme != scheme
2665    #   ifdef   TPM_ALG_ECDAA
2666           || scheme == TPM_ALG_ECDAA
2667    #   endif   // TPM_ALG_ECDAA
2668
2669          )
2670            return TRUE;
2671        return FALSE;
2672    }
```

10.2.9.17 CryptIsSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```
2673    BOOL
2674    CryptIsSignScheme(
2675        TPMI_ALG_ASYM_SCHEME     scheme
2676        )
2677    {
2678        BOOL            isSignScheme = FALSE;
2679
2680        switch(scheme)
2681        {
2682    #ifdef TPM_ALG_RSA
2683            // If RSA is implemented, then both signing schemes are required
2684        case TPM_ALG_RSASSA:
2685        case TPM_ALG_RSAPSS:
2686            isSignScheme = TRUE;
2687            break;
2688    #endif //TPM_ALG_RSA
2689
2690    #ifdef TPM_ALG_ECC
2691            // If ECC is implemented ECDSA is required
2692        case TPM_ALG_ECDSA:
2693    #ifdef  TPM_ALG_ECDAA
2694            // ECDAA is optional
2695        case TPM_ALG_ECDAA:
2696    #endif
2697    #ifdef   TPM_ALG_ECSCHNORR
2698            // Schnorr is also optional
2699        case TPM_ALG_ECSCHNORR:
2700    #endif
2701    #ifdef  TPM_ALG_SM2
2702        case TPM_ALG_SM2:
```

```
#endif
        isSignScheme = TRUE;
        break;
#endif //TPM_ALG_ECC
    default:
        break;
    }
    return isSignScheme;
}
```

10.2.9.18 CryptIsDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
BOOL
CryptIsDecryptScheme(
    TPMI_ALG_ASYM_SCHEME     scheme
    )
{
    BOOL        isDecryptScheme = FALSE;

    switch(scheme)
    {
#ifdef TPM_ALG_RSA
        // If RSA is implemented, then both decrypt schemes are required
    case TPM_ALG_RSAES:
    case TPM_ALG_OAEP:
         isDecryptScheme = 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
        isDecryptScheme = TRUE;
        break;
#endif //TPM_ALG_ECC
    default:
        break;
    }
    return isDecryptScheme;
}
```

10.2.9.19 CryptSelectSignScheme()

This function is used by the attestation and signing commands. It implements the rules for selecting the signature scheme to use in signing. This function requires that the signing key either be TPM_RH_NULL or be loaded.

If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both object and input scheme has a non-NULL scheme algorithm, if the schemes are compatible, the input scheme will be chosen.

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

```
2746  TPM_RC
2747  CryptSelectSignScheme(
2748      TPMI_DH_OBJECT       signHandle,     // IN: handle of signing key
2749      TPMT_SIG_SCHEME     *scheme          // IN/OUT: signing scheme
2750      )
2751  {
2752      OBJECT              *signObject;
2753      TPMT_SIG_SCHEME     *objectScheme;
2754      TPMT_PUBLIC         *publicArea;
2755      TPM_RC               result = TPM_RC_SUCCESS;
2756
2757      // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
2758      // of the setting of scheme
2759      if(signHandle == TPM_RH_NULL)
2760      {
2761          scheme->scheme = TPM_ALG_NULL;
2762          scheme->details.any.hashAlg = TPM_ALG_NULL;
2763      }
2764      else
2765      {
2766          // sign handle is not NULL so...
2767          // Get sign object pointer
2768          signObject = ObjectGet(signHandle);
2769          publicArea = &signObject->publicArea;
2770
2771          // is this a signing key?
2772          if(!publicArea->objectAttributes.sign)
2773              result = TPM_RC_KEY;
2774          else
2775          {
2776              // "parms" defined to avoid long code lines.
2777              TPMU_PUBLIC_PARMS   *parms = &publicArea->parameters;
2778              if(CryptIsAsymAlgorithm(publicArea->type))
2779                  objectScheme = (TPMT_SIG_SCHEME *)&parms->asymDetail.scheme;
2780              else
2781                  objectScheme = (TPMT_SIG_SCHEME *)&parms->keyedHashDetail.scheme;
2782
2783              // If the object doesn't have a default scheme, then use the
2784              // input scheme.
2785              if(objectScheme->scheme == TPM_ALG_NULL)
2786              {
2787                  // Input and default can't both be NULL
2788                  if(scheme->scheme == TPM_ALG_NULL)
2789                      result = TPM_RC_SCHEME;
2790
2791                  // Assume that the scheme is compatible with the key. If not,
2792                  // we will generate an error in the signing operation.
2793
2794              }
2795              else if(scheme->scheme == TPM_ALG_NULL)
2796              {
2797                  // input scheme is NULL so use default
2798
2799                  // First, check to see if the default requires that the caller
2800                  // provided scheme data
2801                  if(CryptIsSplitSign(objectScheme->scheme))
2802                      result = TPM_RC_SCHEME;
```

```
2803                    else
2804                    {
2805                        scheme->scheme = objectScheme->scheme;
2806                        scheme->details.any.hashAlg
2807                                    = objectScheme->details.any.hashAlg;
2808                    }
2809                }
2810                else
2811                {
2812                    // Both input and object have scheme selectors
2813                    // If the scheme and the hash are not the same then...
2814                    if(   objectScheme->scheme != scheme->scheme
2815                       || (   objectScheme->details.any.hashAlg
2816                           != scheme->details.any.hashAlg))
2817                        result = TPM_RC_SCHEME;
2818                }
2819            }
2820
2821        }
2822        return result;
2823    }
```

10.2.9.20 CryptSign()

Sign a digest with asymmetric key or HMAC. This function is called by attestation commands and the generic TPM2_Sign() command. This function checks the key scheme and digest size. It does not check if the sign operation is allowed for restricted key. It should be checked before the function is called. The function will assert if the key is not a signing key.

Error Returns	Meaning
TPM_RC_SCHEME	*signScheme* is not compatible with the signing key type
TPM_RC_VALUE	*digest* value is greater than the modulus of *signHandle* or size of *hashData* does not match hash algorithm in*signScheme* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
2824    TPM_RC
2825    CryptSign(
2826        TPMI_DH_OBJECT       signHandle,     // IN: The handle of sign key
2827        TPMT_SIG_SCHEME     *signScheme,     // IN: sign scheme.
2828        TPM2B_DIGEST        *digest,         // IN: The digest being signed
2829        TPMT_SIGNATURE      *signature       // OUT: signature
2830        )
2831    {
2832        OBJECT              *signKey = ObjectGet(signHandle);
2833        TPM_RC               result = TPM_RC_SCHEME;
2834
2835        // check if input handle is a sign key
2836        pAssert(signKey->publicArea.objectAttributes.sign == SET);
2837
2838        // Must have the private portion loaded.  This check is made during
2839        // authorization.
2840        pAssert(signKey->attributes.publicOnly == CLEAR);
2841
2842        // Initialize signature scheme
2843        signature->sigAlg = signScheme->scheme;
2844
2845        // If the signature algorithm is TPM_ALG_NULL, then we are done
2846        if(signature->sigAlg == TPM_ALG_NULL)
2847            return TPM_RC_SUCCESS;
2848
2849        // All the schemes other than TPM_ALG_NULL have a hash algorithm
```

```
2850        TEST_HASH(signScheme->details.any.hashAlg);
2851
2852        // Initialize signature hash
2853        // Note: need to do the check for alg null first because the null scheme
2854        // doesn't have a hashAlg member.
2855        signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
2856
2857        // perform sign operation based on different key type
2858        switch (signKey->publicArea.type)
2859        {
2860
2861    #ifdef TPM_ALG_RSA
2862            case TPM_ALG_RSA:
2863                result = CryptSignRSA(signKey, signScheme, digest, signature);
2864                break;
2865    #endif //TPM_ALG_RSA
2866
2867    #ifdef TPM_ALG_ECC
2868            case TPM_ALG_ECC:
2869                result = CryptSignECC(signKey, signScheme, digest, signature);
2870                break;
2871    #endif //TPM_ALG_ECC
2872            case TPM_ALG_KEYEDHASH:
2873                result = CryptSignHMAC(signKey, signScheme, digest, signature);
2874                break;
2875            default:
2876                break;
2877        }
2878
2879        return result;
2880    }
```

10.2.9.21 CryptVerifySignature()

This function is used to verify a signature. It is called by TPM2_VerifySignature() and TPM2_PolicySigned().

Since this operation only requires use of a public key, no consistency checks are necessary for the key to signature type because a caller can load any public key that they like with any scheme that they like. This routine simply makes sure that the signature is correct, whatever the type.

This function requires that *auth* is not a NULL pointer.

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SCHEME	the scheme is not supported
TPM_RC_HANDLE	an HMAC key was selected but the private part of the key is not loaded

```
2881    TPM_RC
2882    CryptVerifySignature(
2883        TPMI_DH_OBJECT   keyHandle,     // IN: The handle of sign key
2884        TPM2B_DIGEST    *digest,        // IN: The digest being validated
2885        TPMT_SIGNATURE  *signature      // IN: signature
2886        )
2887    {
2888        // NOTE: ObjectGet will either return a pointer to a loaded object or
2889        // will assert. It will never return a non-valid value. This makes it save
2890        // to initialize 'publicArea' with the return value from ObjectGet() without
2891        // checking it first.
2892        OBJECT              *authObject = ObjectGet(keyHandle);
2893        TPMT_PUBLIC         *publicArea = &authObject->publicArea;
```

```
2894        TPM_RC                result = TPM_RC_SCHEME;
2895
2896        // The input unmarshaling should prevent any input signature from being
2897        // a NULL signature, but just in case
2898        if(signature->sigAlg == TPM_ALG_NULL)
2899            return TPM_RC_SIGNATURE;
2900
2901        switch (publicArea->type)
2902        {
2903
2904    #ifdef TPM_ALG_RSA
2905        case TPM_ALG_RSA:
2906            result = CryptRSAVerifySignature(authObject, digest, signature);
2907            break;
2908    #endif //TPM_ALG_RSA
2909
2910    #ifdef TPM_ALG_ECC
2911        case TPM_ALG_ECC:
2912            result = CryptECCVerifySignature(authObject, digest, signature);
2913            break;
2914
2915    #endif // TPM_ALG_ECC
2916
2917        case TPM_ALG_KEYEDHASH:
2918            if(authObject->attributes.publicOnly)
2919                result = TPM_RCS_HANDLE;
2920            else
2921                result = CryptHMACVerifySignature(authObject, digest, signature);
2922            break;
2923
2924        default:
2925            break;
2926        }
2927        return result;
2928
2929    }
```

10.2.10 Math functions

10.2.10.1 CryptDivide()

This function interfaces to the math library for large number divide.

Error Returns	Meaning
TPM_RC_SIZE	*quotient* or *remainder* is too small to receive the result

```
2930    TPM_RC
2931    CryptDivide(
2932        TPM2B           *numerator,     // IN: numerator
2933        TPM2B           *denominator,   // IN: denominator
2934        TPM2B           *quotient,      // OUT: quotient = numerator / denominator.
2935        TPM2B           *remainder      // OUT: numerator mod denominator.
2936        )
2937    {
2938        pAssert(   numerator != NULL && denominator!= NULL
2939                && (quotient != NULL || remainder != NULL)
2940               );
2941        // assume denominator is not 0
2942        pAssert(denominator->size != 0);
2943
2944        return TranslateCryptErrors(_math__Div(numerator,
2945                                               denominator,
```

```
2946                                            quotient,
2947                                            remainder)
2948                                );
2949    }
```

10.2.10.2 CryptCompare()

This function interfaces to the math library for large number, unsigned compare.

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
2950    LIB_EXPORT int
2951    CryptCompare(
2952        const UINT32     aSize,         // IN: size of a
2953        const BYTE      *a,             // IN: a buffer
2954        const UINT32     bSize,         // IN: size of b
2955        const BYTE      *b              // IN: b buffer
2956        )
2957    {
2958        return _math__uComp(aSize, a, bSize, b);
2959    }
```

10.2.10.3 CryptCompareSigned()

This function interfaces to the math library for large number, signed compare.

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
2960    int
2961    CryptCompareSigned(
2962        UINT32           aSize,         // IN: size of a
2963        BYTE            *a,             // IN: a buffer
2964        UINT32           bSize,         // IN: size of b
2965        BYTE            *b              // IN: b buffer
2966        )
2967    {
2968        return _math__Comp(aSize, a, bSize, b);
2969    }
```

10.2.10.4 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().

```
2970    TPM_RC
2971    CryptGetTestResult(
2972        TPM2B_MAX_BUFFER    *outData        // OUT: test result data
```

```
2973        )
2974    {
2975        outData->t.size = 0;
2976        return TPM_RC_SUCCESS;
2977    }
```

10.2.11 Capability Support

10.2.11.1 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

Return Value	Meaning
YES	if no more ECC curve is available
NO	if there are more ECC curves not reported

```
2978    #ifdef TPM_ALG_ECC //% 5
2979    TPMI_YES_NO
2980    CryptCapGetECCCurve(
2981        TPM_ECC_CURVE    curveID,       // IN: the starting ECC curve
2982        UINT32           maxCount,      // IN: count of returned curve
2983        TPML_ECC_CURVE  *curveList      // OUT: ECC curve list
2984        )
2985    {
2986        TPMI_YES_NO       more = NO;
2987        UINT16            i;
2988        UINT32            count = _cpri__EccGetCurveCount();
2989        TPM_ECC_CURVE     curve;
2990
2991        // Initialize output property list
2992        curveList->count = 0;
2993
2994        // The maximum count of curves we may return is MAX_ECC_CURVES
2995        if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
2996
2997        // Scan the eccCurveValues array
2998        for(i = 0; i < count; i++)
2999        {
3000            curve = _cpri__GetCurveIdByIndex(i);
3001            // If curveID is less than the starting curveID, skip it
3002            if(curve < curveID)
3003                continue;
3004
3005            if(curveList->count < maxCount)
3006            {
3007                // If we have not filled up the return list, add more curves to
3008                // it
3009                curveList->eccCurves[curveList->count] = curve;
3010                curveList->count++;
3011            }
3012            else
3013            {
3014                // If the return list is full but we still have curves
3015                // available, report this and stop iterating
3016                more = YES;
3017                break;
3018            }
3019
3020        }
3021
3022        return more;
3023
```

```
3024    }
```

10.2.11.2 CryptCapGetEccCurveNumber()

This function returns the number of ECC curves supported by the TPM.

```
3025    UINT32
3026    CryptCapGetEccCurveNumber(
3027        void
3028        )
3029    {
3030        // There is an array that holds the curve data. Its size divided by the
3031        // size of an entry is the number of values in the table.
3032        return _cpri__EccGetCurveCount();
3033    }
3034    #endif //TPM_ALG_ECC //% 5
```

10.2.11.3 CryptAreKeySizesConsistent()

This function validates that the public key size values are consistent for an asymmetric key.

NOTE: This is not a comprehensive test of the public key.

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```
3035    BOOL
3036    CryptAreKeySizesConsistent(
3037        TPMT_PUBLIC     *publicArea     // IN: the public area to check
3038        )
3039    {
3040        BOOL            consistent = FALSE;
3041
3042        switch (publicArea->type)
3043        {
3044    #ifdef TPM_ALG_RSA
3045            case TPM_ALG_RSA:
3046                // The key size in bits is filtered by the unmarshaling
3047                consistent =  (    ((publicArea->parameters.rsaDetail.keyBits+7)/8)
3048                                == publicArea->unique.rsa.t.size);
3049                break;
3050    #endif //TPM_ALG_RSA
3051
3052    #ifdef TPM_ALG_ECC
3053            case TPM_ALG_ECC:
3054                {
3055                    UINT16            keySizeInBytes;
3056                    TPM_ECC_CURVE     curveId = publicArea->parameters.eccDetail.curveID;
3057
3058                    keySizeInBytes = CryptEccGetKeySizeInBytes(curveId);
3059
3060                    consistent =   keySizeInBytes > 0
3061                                && publicArea->unique.ecc.x.t.size <= keySizeInBytes
3062                                && publicArea->unique.ecc.y.t.size <= keySizeInBytes;
3063                }
3064                break;
3065    #endif //TPM_ALG_ECC
3066            default:
3067                break;
```

```
3068        }
3069
3070        return consistent;
3071    }
```

10.2.11.4 CryptAlgSetImplemented()

This function initializes the bit vector with one bit for each implemented algorithm. This function is called from _TPM_Init(). The vector of implemented algorithms should be generated by the part 2 parser so that the *g_implementedAlgorithms* vector can be a const. That's not how it is now

```
3072    void
3073    CryptAlgsSetImplemented(
3074        void
3075        )
3076    {
3077        AlgorithmGetImplementedVector(&g_implementedAlgorithms);
3078    }
```

10.3 Ticket.c

10.3.1 Introduction

This clause contains the functions used for ticket computations.

10.3.2 Includes

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

10.3.3 Functions

10.3.3.1 TicketIsSafe()

This function indicates if producing a ticket is safe. It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE or its substring of canonical form. If so, it is not safe to produce ticket for an input buffer claiming to be TPM generated buffer

Return Value	Meaning
TRUE	It is safe to produce ticket
FALSE	It is not safe to produce ticket

```
2   BOOL
3   TicketIsSafe(
4       TPM2B           *buffer
5       )
6   {
7       TPM_GENERATED   valueToCompare = TPM_GENERATED_VALUE;
8       BYTE            bufferToCompare[sizeof(valueToCompare)];
9       BYTE            *marshalBuffer;
10
11      // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
12      // it is not safe to generate a ticket
13      if(buffer->size < sizeof(valueToCompare))
14          return FALSE;
15
16      marshalBuffer = bufferToCompare;
```

```
17      TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);
18      if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
19          return FALSE;
20      else
21          return TRUE;
22  }
```

10.3.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```
23  void
24  TicketComputeVerified(
25      TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
26      TPM2B_DIGEST        *digest,         // IN: digest
27      TPM2B_NAME          *keyName,        // IN: name of key that signed the value
28      TPMT_TK_VERIFIED    *ticket          // OUT: verified ticket
29      )
30  {
31      TPM2B_AUTH          *proof;
32      HMAC_STATE           hmacState;
33
34      // Fill in ticket fields
35      ticket->tag = TPM_ST_VERIFIED;
36      ticket->hierarchy = hierarchy;
37
38      // Use the proof value of the hierarchy
39      proof = HierarchyGetProof(hierarchy);
40
41      // Start HMAC
42      ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
43                                               &proof->b, &hmacState);
44
45      // add TPM_ST_VERIFIED
46      CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
47
48      // add digest
49      CryptUpdateDigest2B(&hmacState, &digest->b);
50
51      // add key name
52      CryptUpdateDigest2B(&hmacState, &keyName->b);
53
54      // complete HMAC
55      CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
56
57      return;
58  }
```

10.3.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```
59  void
60  TicketComputeAuth(
61      TPM_ST               type,           // IN: the type of ticket.
62      TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
63      UINT64               timeout,        // IN: timeout
64      TPM2B_DIGEST        *cpHashA,        // IN: input cpHashA
65      TPM2B_NONCE         *policyRef,      // IN: input policyRef
66      TPM2B_NAME          *entityName,     // IN: name of entity
67      TPMT_TK_AUTH        *ticket          // OUT: Created ticket
68      )
69  {
```

```
70      TPM2B_AUTH          *proof;
71      HMAC_STATE           hmacState;
72
73      // Get proper proof
74      proof = HierarchyGetProof(hierarchy);
75
76      // Fill in ticket fields
77      ticket->tag = type;
78      ticket->hierarchy = hierarchy;
79
80      // Start HMAC
81      ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
82                                               &proof->b, &hmacState);
83
84      // Adding TPM_ST_AUTH
85      CryptUpdateDigestInt(&hmacState, sizeof(UINT16), &ticket->tag);
86
87      // Adding timeout
88      CryptUpdateDigestInt(&hmacState, sizeof(UINT64), &timeout);
89
90      // Adding cpHash
91      CryptUpdateDigest2B(&hmacState, &cpHashA->b);
92
93      // Adding policyRef
94      CryptUpdateDigest2B(&hmacState, &policyRef->b);
95
96      // Adding keyName
97      CryptUpdateDigest2B(&hmacState, &entityName->b);
98
99      // Compute HMAC
100     CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
101
102     return;
103 }
```

10.3.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```
104 void
105 TicketComputeHashCheck(
106     TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
107     TPM_ALG_ID           hashAlg,        // IN: the hash algorithm used to create
108                                          //     'digest'
109     TPM2B_DIGEST        *digest,         // IN: input digest
110     TPMT_TK_HASHCHECK   *ticket          // OUT: Created ticket
111     )
112 {
113     TPM2B_AUTH          *proof;
114     HMAC_STATE           hmacState;
115
116     // Get proper proof
117     proof = HierarchyGetProof(hierarchy);
118
119     // Fill in ticket fields
120     ticket->tag = TPM_ST_HASHCHECK;
121     ticket->hierarchy = hierarchy;
122
123     ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
124                                              &proof->b, &hmacState);
125
126     // Add TPM_ST_HASHCHECK
127     CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
128
```

```
129         // Add hash algorithm
130         CryptUpdateDigestInt(&hmacState, sizeof(hashAlg), &hashAlg);
131
132         // Add digest
133         CryptUpdateDigest2B(&hmacState, &digest->b);
134
135         // Compute HMAC
136         CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
137
138         return;
139     }
```

10.3.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```
140     void
141     TicketComputeCreation(
142         TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy for ticket
143         TPM2B_NAME          *name,           // IN: object name
144         TPM2B_DIGEST        *creation,       // IN: creation hash
145         TPMT_TK_CREATION    *ticket          // OUT: created ticket
146         )
147     {
148         TPM2B_AUTH          *proof;
149         HMAC_STATE           hmacState;
150
151         // Get proper proof
152         proof = HierarchyGetProof(hierarchy);
153
154         // Fill in ticket fields
155         ticket->tag = TPM_ST_CREATION;
156         ticket->hierarchy = hierarchy;
157
158         ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
159                                                  &proof->b, &hmacState);
160
161         // Add TPM_ST_CREATION
162         CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
163
164         // Add name
165         CryptUpdateDigest2B(&hmacState, &name->b);
166
167         // Add creation hash
168         CryptUpdateDigest2B(&hmacState, &creation->b);
169
170         // Compute HMAC
171         CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
172
173         return;
174     }
```

10.4 CryptSelfTest.c

10.4.1 Introduction

The functions in this file are designed to support self-test of cryptographic functions in the TPM. The TPM allows the user to decide whether to run self-test on a demand basis or to run all the self-tests before proceeding.

The self-tests are controlled by a set of bit vectors. The *g_untestedDecryptionAlgorithms* vector has a bit for each decryption algorithm that needs to be tested and *g_untestedEncryptionAlgorithms* has a bit for

each encryption algorithm that needs to be tested. Before an algorithm is used, the appropriate vector is checked (indexed using the algorithm ID). If the bit is SET, then the test function should be called.

```
#include    "Global.h"
#include    "CryptoEngine.h"
#include    "InternalRoutines.h"
#include    "AlgorithmCap_fp.h"
```

10.4.2 Functions

10.4.2.1 RunSelfTest()

Local function to run self-test

```
static TPM_RC
CryptRunSelfTests(
    ALGORITHM_VECTOR    *toTest         // IN: the vector of the algorithms to test
    )
{
    TPM_ALG_ID           alg;

    // For each of the algorithms that are in the toTestVecor, need to run a
    // test
    for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
    {
        if(TEST_BIT(alg, *toTest))
        {
            TPM_RC          result = CryptTestAlgorithm(alg, toTest);
            if(result != TPM_RC_SUCCESS)
                return result;
        }
    }
    return TPM_RC_SUCCESS;
}
```

10.4.2.2 CryptSelfTest()

This function is called to start/complete a full self-test. If *fullTest* is NO, then only the untested algorithms will be run. If *fullTest* is YES, then *g_untestedDecryptionAlgorithms* is reinitialized and then all tests are run. This implementation of the reference design does not support processing outside the framework of a TPM command. As a consequence, this command does not complete until all tests are done. Since this can take a long time, the TPM will check after each test to see if the command is canceled. If so, then the TPM will returned TPM_RC_CANCELLED. To continue with the self-tests, call TPM2_SelfTest(*fullTest* == No) and the TPM will complete the testing.

Error Returns	Meaning
TPM_RC_CANCELED	if the command is canceled

```
LIB_EXPORT
TPM_RC
CryptSelfTest(
    TPMI_YES_NO      fullTest       // IN: if full test is required
    )
{
    if(g_forceFailureMode)
        FAIL(FATAL_ERROR_FORCED);

    // If the caller requested a full test, then reset the to test vector so that
    // all the tests will be run
```

```
36      if(fullTest == YES)
37      {
38          MemoryCopy(g_toTest,
39                     g_implementedAlgorithms,
40                     sizeof(g_toTest), sizeof(g_toTest));
41      }
42      return CryptRunSelfTests(&g_toTest);
43  }
```

10.4.2.3 CryptIncrementalSelfTest()

This function is used to perform an incremental self-test. This implementation will perform the *toTest* values before returning. That is, it assumes that the TPM cannot perform background tasks between commands.

This command may be canceled. If it is, then there is no return result. However, this command can be run again and the incremental progress will not be lost.

Error Returns	Meaning
TPM_RC_CANCELED	processing of this command was canceled
TPM_RC_TESTING	if *toTest* list is not empty
TPM_RC_VALUE	an algorithm in the *toTest* list is not implemented

```
44  TPM_RC
45  CryptIncrementalSelfTest(
46      TPML_ALG        *toTest,        // IN: list of algorithms to be tested
47      TPML_ALG        *toDoList       // OUT: list of algorithms needing test
48      )
49  {
50      ALGORITHM_VECTOR    toTestVector = {0};
51      TPM_ALG_ID          alg;
52      UINT32                 i;
53
54      pAssert(toTest != NULL && toDoList != NULL);
55      if(toTest->count > 0)
56      {
57          // Transcribe the toTest list into the toTestVector
58          for(i = 0; i < toTest->count; i++)
59          {
60              TPM_ALG_ID      alg = toTest->algorithms[i];
61
62              // make sure that the algorithm value is not out of range
63              if((alg > TPM_ALG_LAST) ||  !TEST_BIT(alg, g_implementedAlgorithms))
64                  return TPM_RC_VALUE;
65              SET_BIT(alg, toTestVector);
66          }
67          // Run the test
68          if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
69              return TPM_RC_CANCELED;
70      }
71      // Fill in the toDoList with the algorithms that are still untested
72      toDoList->count = 0;
73
74      for(alg = TPM_ALG_FIRST;
75          toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
76          alg++)
77      {
78          if(TEST_BIT(alg, g_toTest))
79              toDoList->algorithms[toDoList->count++] = alg;
80      }
81      return TPM_RC_SUCCESS;
```

```
82  }
```

10.4.2.4 CryptInitializeToTest()

This function will initialize the data structures for testing all the algorithms. This should not be called unless CryptAlgsSetImplemented() has been called

```
83  void
84  CryptInitializeToTest(
85      void
86      )
87  {
88      MemoryCopy(g_toTest,
89                 g_implementedAlgorithms,
90                 sizeof(g_toTest),
91                 sizeof(g_toTest));
92      // Setting the algorithm to null causes the test function to just clear
93      // out any algorithms for which there is no test.
94      CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);
95
96      return;
97  }
```

10.4.2.5 CryptTestAlgorithm()

Only point of contact with the actual self tests. If a self-test fails, there is no return and the TPM goes into failure mode. The call to TestAlgorithm() uses an algorithms selector and a bit vector. When the test is run, the corresponding bit in *toTest* and in *g_toTest* is CLEAR. If *toTest* is NULL, then only the bit in *g_toTest* is CLEAR. There is a special case for the call to TestAlgorithm(). When *alg* is TPM_ALG_ERROR, TestAlgorithm() will CLEAR any bit in *toTest* for which it has no test. This allows the knowledge about which algorithms have test to be accessed through the interface that provides the test.

Error Returns	Meaning
TPM_RC_SUCCESS	test complete
TPM_RC_CANCELED	test was canceled

```
98  LIB_EXPORT
99  TPM_RC
100 CryptTestAlgorithm(
101     TPM_ALG_ID           alg,
102     ALGORITHM_VECTOR    *toTest
103     )
104 {
105     TPM_RC                  result = TPM_RC_SUCCESS;
106 #ifdef SELF_TEST
107     // This is the function prototype for TestAlgorithms(). It is here and not
108     // in a _fp.h file to avoid a compiler error when SELF_TEST is not defined and
109     // AlgorithmTexts.c is not part of the build.
110     TPM_RC TestAlgorithm(TPM_ALG_ID alg, ALGORITHM_VECTOR *toTest);
111     result = TestAlgorithm(alg, toTest);
112 #else
113     // If this is an attempt to determine the algorithms for which there is a
114     // self test, pretend that all of them do. We do that by not clearing any
115     // of the algorithm bits. When/if this function is called to run tests, it
116     // will over report. This can be changed so that any call to check on which
117     // algorithms have tests, 'toTest' can be cleared.
118     if(alg != TPM_ALG_ERROR)
119     {
120         CLEAR_BIT(alg, g_toTest);
121         if(toTest != NULL)
```

```
122                 CLEAR_BIT(alg, *toTest);
123         }
124     #endif
125         return result;
126     }
```

Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are derived from the values in the annexes of TPM 2.0 Part 2. This file would change based on the implementation.

The values shown in this version of the file reflect the example settings in TPM 2.0 Part 2.

A.2 Implementation.h

```
#ifndef _IMPLEMENTATION_H_
#define _IMPLEMENTATION_H_
#include    "BaseTypes.h"
#include    "TPMB.h"
#undef TRUE
#undef FALSE
```

This table is built in to TpmStructures() Change these definitions to turn all algorithms or commands on or off

```
#define      ALG_YES      YES
#define      ALG_NO       NO
#define      CC_YES       YES
#define      CC_NO        NO
```

From TPM 2.0 Part 2: Table 4 - Defines for Logic Values

```
#define  TRUE     1
#define  FALSE    0
#define  YES      1
#define  NO       0
#define  SET      1
#define  CLEAR    0
```

From Vendor-Specific: Table 1 - Defines for Processor Values

```
#define  BIG_ENDIAN_TPM       NO
#define  LITTLE_ENDIAN_TPM    YES
#define  NO_AUTO_ALIGN        NO
```

From Vendor-Specific: Table 2 - Defines for Implemented Algorithms

```
#define  ALG_RSA              ALG_YES
#define  ALG_SHA1             ALG_YES
#define  ALG_HMAC             ALG_YES
#define  ALG_AES              ALG_YES
#define  ALG_MGF1             ALG_YES
#define  ALG_XOR              ALG_YES
#define  ALG_KEYEDHASH        ALG_YES
#define  ALG_SHA256           ALG_YES
#define  ALG_SHA384           ALG_YES
#define  ALG_SHA512           ALG_NO
#define  ALG_SM3_256          ALG_NO
#define  ALG_SM4              ALG_NO
#define  ALG_RSASSA           (ALG_YES*ALG_RSA)
#define  ALG_RSAES            (ALG_YES*ALG_RSA)
#define  ALG_RSAPSS           (ALG_YES*ALG_RSA)
```



```
35  #define  ALG_OAEP              (ALG_YES*ALG_RSA)
36  #define  ALG_ECC               ALG_YES
37  #define  ALG_ECDH              (ALG_YES*ALG_ECC)
38  #define  ALG_ECDSA             (ALG_YES*ALG_ECC)
39  #define  ALG_ECDAA             (ALG_YES*ALG_ECC)
40  #define  ALG_SM2               (ALG_YES*ALG_ECC)
41  #define  ALG_ECSCHNORR         (ALG_YES*ALG_ECC)
42  #define  ALG_ECMQV             (ALG_NO*ALG_ECC)
43  #define  ALG_SYMCIPHER         ALG_YES
44  #define  ALG_KDF1_SP800_56A    (ALG_YES*ALG_ECC)
45  #define  ALG_KDF2              ALG_NO
46  #define  ALG_KDF1_SP800_108    ALG_YES
47  #define  ALG_CTR               ALG_YES
48  #define  ALG_OFB               ALG_YES
49  #define  ALG_CBC               ALG_YES
50  #define  ALG_CFB               ALG_YES
51  #define  ALG_ECB               ALG_YES
```

From Vendor-Specific: Table 4 - Defines for Key Size Constants

```
52  #define  RSA_KEY_SIZES_BITS         {1024,2048}
53  #define  RSA_KEY_SIZE_BITS_1024     RSA_ALLOWED_KEY_SIZE_1024
54  #define  RSA_KEY_SIZE_BITS_2048     RSA_ALLOWED_KEY_SIZE_2048
55  #define  MAX_RSA_KEY_BITS           2048
56  #define  MAX_RSA_KEY_BYTES          256
57  #define  AES_KEY_SIZES_BITS         {128,256}
58  #define  AES_KEY_SIZE_BITS_128      AES_ALLOWED_KEY_SIZE_128
59  #define  AES_KEY_SIZE_BITS_256      AES_ALLOWED_KEY_SIZE_256
60  #define  MAX_AES_KEY_BITS           256
61  #define  MAX_AES_KEY_BYTES          32
62  #define MAX_AES_BLOCK_SIZE_BYTES                              \
63              MAX(AES_128_BLOCK_SIZE_BYTES,                     \
64              MAX(AES_256_BLOCK_SIZE_BYTES, 0))
65  #define  SM4_KEY_SIZES_BITS         {128}
66  #define  SM4_KEY_SIZE_BITS_128      SM4_ALLOWED_KEY_SIZE_128
67  #define  MAX_SM4_KEY_BITS           128
68  #define  MAX_SM4_KEY_BYTES          16
69  #define MAX_SM4_BLOCK_SIZE_BYTES                              \
70              MAX(SM4_128_BLOCK_SIZE_BYTES, 0)
71  #define  CAMELLIA_KEY_SIZES_BITS    {128}
72  #define  CAMELLIA_KEY_SIZE_BITS_128    CAMELLIA_ALLOWED_KEY_SIZE_128
73  #define  MAX_CAMELLIA_KEY_BITS      128
74  #define  MAX_CAMELLIA_KEY_BYTES     16
75  #define MAX_CAMELLIA_BLOCK_SIZE_BYTES                         \
76              MAX(CAMELLIA_128_BLOCK_SIZE_BYTES, 0)
```

From Vendor-Specific: Table 5 - Defines for Implemented Curves

```
77  #define  ECC_NIST_P256         YES
78  #define  ECC_NIST_P384         YES
79  #define  ECC_BN_P256           YES
80  #define  ECC_CURVES            {\
81      TPM_ECC_BN_P256, TPM_ECC_NIST_P256, TPM_ECC_NIST_P384}
82  #define  ECC_KEY_SIZES_BITS    {256, 384}
83  #define  ECC_KEY_SIZE_BITS_256
84  #define  ECC_KEY_SIZE_BITS_384
85  #define  MAX_ECC_KEY_BITS      384
86  #define  MAX_ECC_KEY_BYTES     48
```

From Vendor-Specific: Table 6 - Defines for Implemented Commands

```
87  #define  CC_ActivateCredential           CC_YES
88  #define  CC_Certify                      CC_YES
89  #define  CC_CertifyCreation              CC_YES
```

```
#define  CC_ChangeEPS                    CC_YES
#define  CC_ChangePPS                    CC_YES
#define  CC_Clear                        CC_YES
#define  CC_ClearControl                 CC_YES
#define  CC_ClockRateAdjust              CC_YES
#define  CC_ClockSet                     CC_YES
#define  CC_Commit                       (CC_YES*ALG_ECC)
#define  CC_ContextLoad                  CC_YES
#define  CC_ContextSave                  CC_YES
#define  CC_Create                       CC_YES
#define  CC_CreatePrimary                CC_YES
#define  CC_DictionaryAttackLockReset    CC_YES
#define  CC_DictionaryAttackParameters   CC_YES
#define  CC_Duplicate                    CC_YES
#define  CC_ECC_Parameters               (CC_YES*ALG_ECC)
#define  CC_ECDH_KeyGen                  (CC_YES*ALG_ECC)
#define  CC_ECDH_ZGen                    (CC_YES*ALG_ECC)
#define  CC_EncryptDecrypt               CC_YES
#define  CC_EventSequenceComplete        CC_YES
#define  CC_EvictControl                 CC_YES
#define  CC_FieldUpgradeData             CC_NO
#define  CC_FieldUpgradeStart            CC_NO
#define  CC_FirmwareRead                 CC_NO
#define  CC_FlushContext                 CC_YES
#define  CC_GetCapability                CC_YES
#define  CC_GetCommandAuditDigest        CC_YES
#define  CC_GetRandom                    CC_YES
#define  CC_GetSessionAuditDigest        CC_YES
#define  CC_GetTestResult                CC_YES
#define  CC_GetTime                      CC_YES
#define  CC_Hash                         CC_YES
#define  CC_HashSequenceStart            CC_YES
#define  CC_HierarchyChangeAuth          CC_YES
#define  CC_HierarchyControl             CC_YES
#define  CC_HMAC                         CC_YES
#define  CC_HMAC_Start                   CC_YES
#define  CC_Import                       CC_YES
#define  CC_IncrementalSelfTest          CC_YES
#define  CC_Load                         CC_YES
#define  CC_LoadExternal                 CC_YES
#define  CC_MakeCredential               CC_YES
#define  CC_NV_Certify                   CC_YES
#define  CC_NV_ChangeAuth                CC_YES
#define  CC_NV_DefineSpace               CC_YES
#define  CC_NV_Extend                    CC_YES
#define  CC_NV_GlobalWriteLock           CC_YES
#define  CC_NV_Increment                 CC_YES
#define  CC_NV_Read                      CC_YES
#define  CC_NV_ReadLock                  CC_YES
#define  CC_NV_ReadPublic                CC_YES
#define  CC_NV_SetBits                   CC_YES
#define  CC_NV_UndefineSpace             CC_YES
#define  CC_NV_UndefineSpaceSpecial      CC_YES
#define  CC_NV_Write                     CC_YES
#define  CC_NV_WriteLock                 CC_YES
#define  CC_ObjectChangeAuth             CC_YES
#define  CC_PCR_Allocate                 CC_YES
#define  CC_PCR_Event                    CC_YES
#define  CC_PCR_Extend                   CC_YES
#define  CC_PCR_Read                     CC_YES
#define  CC_PCR_Reset                    CC_YES
#define  CC_PCR_SetAuthPolicy            CC_YES
#define  CC_PCR_SetAuthValue             CC_YES
#define  CC_PolicyAuthorize              CC_YES
#define  CC_PolicyAuthValue              CC_YES
#define  CC_PolicyCommandCode            CC_YES
```

```
156   #define  CC_PolicyCounterTimer            CC_YES
157   #define  CC_PolicyCpHash                  CC_YES
158   #define  CC_PolicyDuplicationSelect       CC_YES
159   #define  CC_PolicyGetDigest               CC_YES
160   #define  CC_PolicyLocality                CC_YES
161   #define  CC_PolicyNameHash                CC_YES
162   #define  CC_PolicyNV                      CC_YES
163   #define  CC_PolicyOR                      CC_YES
164   #define  CC_PolicyPassword                CC_YES
165   #define  CC_PolicyPCR                     CC_YES
166   #define  CC_PolicyPhysicalPresence        CC_YES
167   #define  CC_PolicyRestart                 CC_YES
168   #define  CC_PolicySecret                  CC_YES
169   #define  CC_PolicySigned                  CC_YES
170   #define  CC_PolicyTicket                  CC_YES
171   #define  CC_PP_Commands                   CC_YES
172   #define  CC_Quote                         CC_YES
173   #define  CC_ReadClock                     CC_YES
174   #define  CC_ReadPublic                    CC_YES
175   #define  CC_Rewrap                        CC_YES
176   #define  CC_RSA_Decrypt                   (CC_YES*ALG_RSA)
177   #define  CC_RSA_Encrypt                   (CC_YES*ALG_RSA)
178   #define  CC_SelfTest                      CC_YES
179   #define  CC_SequenceComplete              CC_YES
180   #define  CC_SequenceUpdate                CC_YES
181   #define  CC_SetAlgorithmSet               CC_YES
182   #define  CC_SetCommandCodeAuditStatus     CC_YES
183   #define  CC_SetPrimaryPolicy              CC_YES
184   #define  CC_Shutdown                      CC_YES
185   #define  CC_Sign                          CC_YES
186   #define  CC_StartAuthSession              CC_YES
187   #define  CC_Startup                       CC_YES
188   #define  CC_StirRandom                    CC_YES
189   #define  CC_TestParms                     CC_YES
190   #define  CC_Unseal                        CC_YES
191   #define  CC_VerifySignature               CC_YES
192   #define  CC_ZGen_2Phase                   (CC_YES*ALG_ECC)
193   #define  CC_EC_Ephemeral                  (CC_YES*ALG_ECC)
194   #define  CC_PolicyNvWritten               CC_YES
```

From Vendor-Specific: Table 7 - Defines for Implementation Values

```
195   #define  FIELD_UPGRADE_IMPLEMENTED        NO
196   #define  BSIZE                            UINT16
197   #define  BUFFER_ALIGNMENT                 4
198   #define  IMPLEMENTATION_PCR               24
199   #define  PLATFORM_PCR                     24
200   #define  DRTM_PCR                         17
201   #define  HCRTM_PCR                        0
202   #define  NUM_LOCALITIES                   5
203   #define  MAX_HANDLE_NUM                   3
204   #define  MAX_ACTIVE_SESSIONS              64
205   #define  CONTEXT_SLOT                     UINT16
206   #define  CONTEXT_COUNTER                  UINT64
207   #define  MAX_LOADED_SESSIONS              3
208   #define  MAX_SESSION_NUM                  3
209   #define  MAX_LOADED_OBJECTS               3
210   #define  MIN_EVICT_OBJECTS                2
211   #define  PCR_SELECT_MIN                   ((PLATFORM_PCR+7)/8)
212   #define  PCR_SELECT_MAX                   ((IMPLEMENTATION_PCR+7)/8)
213   #define  NUM_POLICY_PCR_GROUP             1
214   #define  NUM_AUTHVALUE_PCR_GROUP          1
215   #define  MAX_CONTEXT_SIZE                 2048
216   #define  MAX_DIGEST_BUFFER                1024
217   #define  MAX_NV_INDEX_SIZE                2048
```

```
218  #define  MAX_NV_BUFFER_SIZE              1024
219  #define  MAX_CAP_BUFFER                  1024
220  #define  NV_MEMORY_SIZE                  16384
221  #define  NUM_STATIC_PCR                  16
222  #define  MAX_ALG_LIST_SIZE               64
223  #define  TIMER_PRESCALE                  100000
224  #define  PRIMARY_SEED_SIZE               32
225  #define  CONTEXT_ENCRYPT_ALG             TPM_ALG_AES
226  #define  CONTEXT_ENCRYPT_KEY_BITS        MAX_SYM_KEY_BITS
227  #define  CONTEXT_ENCRYPT_KEY_BYTES       ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
228  #define  CONTEXT_INTEGRITY_HASH_ALG      TPM_ALG_SHA256
229  #define  CONTEXT_INTEGRITY_HASH_SIZE     SHA256_DIGEST_SIZE
230  #define  PROOF_SIZE                      CONTEXT_INTEGRITY_HASH_SIZE
231  #define  NV_CLOCK_UPDATE_INTERVAL        12
232  #define  NUM_POLICY_PCR                  1
233  #define  MAX_COMMAND_SIZE                4096
234  #define  MAX_RESPONSE_SIZE               4096
235  #define  ORDERLY_BITS                    8
236  #define  MAX_ORDERLY_COUNT               ((1<<ORDERLY_BITS)-1)
237  #define  ALG_ID_FIRST                    TPM_ALG_FIRST
238  #define  ALG_ID_LAST                     TPM_ALG_LAST
239  #define  MAX_SYM_DATA                    128
240  #define  MAX_RNG_ENTROPY_SIZE            64
241  #define  RAM_INDEX_SPACE                 512
242  #define  RSA_DEFAULT_PUBLIC_EXPONENT     0x00010001
243  #define  ENABLE_PCR_NO_INCREMENT         YES
244  #define  CRT_FORMAT_RSA                  YES
245  #define  PRIVATE_VENDOR_SPECIFIC_BYTES   \
246      ((MAX_RSA_KEY_BYTES/2)*(3+CRT_FORMAT_RSA*2))
```

From TCG Algorithm Registry: Table 2 - Definition of TPM_ALG_ID Constants

```
247  typedef  UINT16             TPM_ALG_ID;
248  #define  TPM_ALG_ERROR               (TPM_ALG_ID)(0x0000)
249  #define  ALG_ERROR_VALUE             0x0000
250  #if defined ALG_RSA && ALG_RSA == YES
251  #define  TPM_ALG_RSA                 (TPM_ALG_ID)(0x0001)
252  #endif
253  #define  ALG_RSA_VALUE               0x0001
254  #if defined ALG_SHA && ALG_SHA == YES
255  #define  TPM_ALG_SHA                 (TPM_ALG_ID)(0x0004)
256  #endif
257  #define  ALG_SHA_VALUE               0x0004
258  #if defined ALG_SHA1 && ALG_SHA1 == YES
259  #define  TPM_ALG_SHA1                (TPM_ALG_ID)(0x0004)
260  #endif
261  #define  ALG_SHA1_VALUE              0x0004
262  #if defined ALG_HMAC && ALG_HMAC == YES
263  #define  TPM_ALG_HMAC                (TPM_ALG_ID)(0x0005)
264  #endif
265  #define  ALG_HMAC_VALUE              0x0005
266  #if defined ALG_AES && ALG_AES == YES
267  #define  TPM_ALG_AES                 (TPM_ALG_ID)(0x0006)
268  #endif
269  #define  ALG_AES_VALUE               0x0006
270  #if defined ALG_MGF1 && ALG_MGF1 == YES
271  #define  TPM_ALG_MGF1                (TPM_ALG_ID)(0x0007)
272  #endif
273  #define  ALG_MGF1_VALUE              0x0007
274  #if defined ALG_KEYEDHASH && ALG_KEYEDHASH == YES
275  #define  TPM_ALG_KEYEDHASH           (TPM_ALG_ID)(0x0008)
276  #endif
277  #define  ALG_KEYEDHASH_VALUE         0x0008
278  #if defined ALG_XOR && ALG_XOR == YES
279  #define  TPM_ALG_XOR                 (TPM_ALG_ID)(0x000A)
```



```
280   #endif
281   #define  ALG_XOR_VALUE                0x000A
282   #if defined ALG_SHA256 && ALG_SHA256 == YES
283   #define  TPM_ALG_SHA256               (TPM_ALG_ID)(0x000B)
284   #endif
285   #define  ALG_SHA256_VALUE             0x000B
286   #if defined ALG_SHA384 && ALG_SHA384 == YES
287   #define  TPM_ALG_SHA384               (TPM_ALG_ID)(0x000C)
288   #endif
289   #define  ALG_SHA384_VALUE             0x000C
290   #if defined ALG_SHA512 && ALG_SHA512 == YES
291   #define  TPM_ALG_SHA512               (TPM_ALG_ID)(0x000D)
292   #endif
293   #define  ALG_SHA512_VALUE             0x000D
294   #define  TPM_ALG_NULL                 (TPM_ALG_ID)(0x0010)
295   #define  ALG_NULL_VALUE               0x0010
296   #if defined ALG_SM3_256 && ALG_SM3_256 == YES
297   #define  TPM_ALG_SM3_256              (TPM_ALG_ID)(0x0012)
298   #endif
299   #define  ALG_SM3_256_VALUE            0x0012
300   #if defined ALG_SM4 && ALG_SM4 == YES
301   #define  TPM_ALG_SM4                  (TPM_ALG_ID)(0x0013)
302   #endif
303   #define  ALG_SM4_VALUE                0x0013
304   #if defined ALG_RSASSA && ALG_RSASSA == YES
305   #define  TPM_ALG_RSASSA               (TPM_ALG_ID)(0x0014)
306   #endif
307   #define  ALG_RSASSA_VALUE             0x0014
308   #if defined ALG_RSAES && ALG_RSAES == YES
309   #define  TPM_ALG_RSAES                (TPM_ALG_ID)(0x0015)
310   #endif
311   #define  ALG_RSAES_VALUE              0x0015
312   #if defined ALG_RSAPSS && ALG_RSAPSS == YES
313   #define  TPM_ALG_RSAPSS               (TPM_ALG_ID)(0x0016)
314   #endif
315   #define  ALG_RSAPSS_VALUE             0x0016
316   #if defined ALG_OAEP && ALG_OAEP == YES
317   #define  TPM_ALG_OAEP                 (TPM_ALG_ID)(0x0017)
318   #endif
319   #define  ALG_OAEP_VALUE               0x0017
320   #if defined ALG_ECDSA && ALG_ECDSA == YES
321   #define  TPM_ALG_ECDSA                (TPM_ALG_ID)(0x0018)
322   #endif
323   #define  ALG_ECDSA_VALUE              0x0018
324   #if defined ALG_ECDH && ALG_ECDH == YES
325   #define  TPM_ALG_ECDH                 (TPM_ALG_ID)(0x0019)
326   #endif
327   #define  ALG_ECDH_VALUE               0x0019
328   #if defined ALG_ECDAA && ALG_ECDAA == YES
329   #define  TPM_ALG_ECDAA                (TPM_ALG_ID)(0x001A)
330   #endif
331   #define  ALG_ECDAA_VALUE              0x001A
332   #if defined ALG_SM2 && ALG_SM2 == YES
333   #define  TPM_ALG_SM2                  (TPM_ALG_ID)(0x001B)
334   #endif
335   #define  ALG_SM2_VALUE                0x001B
336   #if defined ALG_ECSCHNORR && ALG_ECSCHNORR == YES
337   #define  TPM_ALG_ECSCHNORR            (TPM_ALG_ID)(0x001C)
338   #endif
339   #define  ALG_ECSCHNORR_VALUE          0x001C
340   #if defined ALG_ECMQV && ALG_ECMQV == YES
341   #define  TPM_ALG_ECMQV                (TPM_ALG_ID)(0x001D)
342   #endif
343   #define  ALG_ECMQV_VALUE              0x001D
344   #if defined ALG_KDF1_SP800_56A && ALG_KDF1_SP800_56A == YES
345   #define  TPM_ALG_KDF1_SP800_56A       (TPM_ALG_ID)(0x0020)
```

```
346   #endif
347   #define  ALG_KDF1_SP800_56A_VALUE    0x0020
348   #if defined ALG_KDF2 && ALG_KDF2 == YES
349   #define  TPM_ALG_KDF2                (TPM_ALG_ID)(0x0021)
350   #endif
351   #define  ALG_KDF2_VALUE              0x0021
352   #if defined ALG_KDF1_SP800_108 && ALG_KDF1_SP800_108 == YES
353   #define  TPM_ALG_KDF1_SP800_108      (TPM_ALG_ID)(0x0022)
354   #endif
355   #define  ALG_KDF1_SP800_108_VALUE    0x0022
356   #if defined ALG_ECC && ALG_ECC == YES
357   #define  TPM_ALG_ECC                 (TPM_ALG_ID)(0x0023)
358   #endif
359   #define  ALG_ECC_VALUE               0x0023
360   #if defined ALG_SYMCIPHER && ALG_SYMCIPHER == YES
361   #define  TPM_ALG_SYMCIPHER           (TPM_ALG_ID)(0x0025)
362   #endif
363   #define  ALG_SYMCIPHER_VALUE         0x0025
364   #if defined ALG_CAMELLIA && ALG_CAMELLIA == YES
365   #define  TPM_ALG_CAMELLIA            (TPM_ALG_ID)(0x0026)
366   #endif
367   #define  ALG_CAMELLIA_VALUE          0x0026
368   #if defined ALG_CTR && ALG_CTR == YES
369   #define  TPM_ALG_CTR                 (TPM_ALG_ID)(0x0040)
370   #endif
371   #define  ALG_CTR_VALUE               0x0040
372   #if defined ALG_OFB && ALG_OFB == YES
373   #define  TPM_ALG_OFB                 (TPM_ALG_ID)(0x0041)
374   #endif
375   #define  ALG_OFB_VALUE               0x0041
376   #if defined ALG_CBC && ALG_CBC == YES
377   #define  TPM_ALG_CBC                 (TPM_ALG_ID)(0x0042)
378   #endif
379   #define  ALG_CBC_VALUE               0x0042
380   #if defined ALG_CFB && ALG_CFB == YES
381   #define  TPM_ALG_CFB                 (TPM_ALG_ID)(0x0043)
382   #endif
383   #define  ALG_CFB_VALUE               0x0043
384   #if defined ALG_ECB && ALG_ECB == YES
385   #define  TPM_ALG_ECB                 (TPM_ALG_ID)(0x0044)
386   #endif
387   #define  ALG_ECB_VALUE               0x0044
388   #define  TPM_ALG_FIRST               (TPM_ALG_ID)(0x0001)
389   #define  ALG_FIRST_VALUE             0x0001
390   #define  TPM_ALG_LAST                (TPM_ALG_ID)(0x0044)
391   #define  ALG_LAST_VALUE              0x0044
```

From TCG Algorithm Registry: Table 3 - Definition of TPM_ECC_CURVE Constants

```
392   typedef  UINT16             TPM_ECC_CURVE;
393   #define  TPM_ECC_NONE          (TPM_ECC_CURVE)(0x0000)
394   #define  TPM_ECC_NIST_P192     (TPM_ECC_CURVE)(0x0001)
395   #define  TPM_ECC_NIST_P224     (TPM_ECC_CURVE)(0x0002)
396   #define  TPM_ECC_NIST_P256     (TPM_ECC_CURVE)(0x0003)
397   #define  TPM_ECC_NIST_P384     (TPM_ECC_CURVE)(0x0004)
398   #define  TPM_ECC_NIST_P521     (TPM_ECC_CURVE)(0x0005)
399   #define  TPM_ECC_BN_P256       (TPM_ECC_CURVE)(0x0010)
400   #define  TPM_ECC_BN_P638       (TPM_ECC_CURVE)(0x0011)
401   #define  TPM_ECC_SM2_P256      (TPM_ECC_CURVE)(0x0020)
```

From TCG Algorithm Registry: Table 4 - Defines for NIST_P192 ECC Values Data in CrpiEccData.c From TCG Algorithm Registry: Table 5 - Defines for NIST_P224 ECC Values Data in CrpiEccData.c From TCG Algorithm Registry: Table 6 - Defines for NIST_P256 ECC Values Data in CrpiEccData.c From TCG Algorithm Registry: Table 7 - Defines for NIST_P384 ECC Values Data in CrpiEccData.c From TCG

Algorithm Registry: Table 8 - Defines for NIST_P521 ECC Values Data in CrpiEccData.c From TCG Algorithm Registry: Table 9 - Defines for BN_P256 ECC Values Data in CrpiEccData.c From TCG Algorithm Registry: Table 10 - Defines for BN_P638 ECC Values Data in CrpiEccData.c From TCG Algorithm Registry: Table 11 - Defines for SM2_P256 ECC Values Data in CrpiEccData.c From TCG Algorithm Registry: Table 12 - Defines for SHA1 Hash Values

```
402  #define  SHA1_DIGEST_SIZE    20
403  #define  SHA1_BLOCK_SIZE     64
404  #define  SHA1_DER_SIZE       15
405  #define  SHA1_DER            \
406      0x30,0x21,0x30,0x09,0x06,0x05,0x2B,0x0E,0x03,0x02,0x1A,0x05,0x00,0x04,0x14
```

From TCG Algorithm Registry: Table 13 - Defines for SHA256 Hash Values

```
407  #define  SHA256_DIGEST_SIZE    32
408  #define  SHA256_BLOCK_SIZE     64
409  #define  SHA256_DER_SIZE       19
410  #define  SHA256_DER            \
411
     0x30,0x31,0x30,0x0D,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x01,0x05,0x00,0
     x04,0x20
```

From TCG Algorithm Registry: Table 14 - Defines for SHA384 Hash Values

```
412  #define  SHA384_DIGEST_SIZE    48
413  #define  SHA384_BLOCK_SIZE     128
414  #define  SHA384_DER_SIZE       19
415  #define  SHA384_DER            \
416
     0x30,0x41,0x30,0x0D,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x02,0x05,0x00,0
     x04,0x30
```

From TCG Algorithm Registry: Table 15 - Defines for SHA512 Hash Values

```
417  #define  SHA512_DIGEST_SIZE    64
418  #define  SHA512_BLOCK_SIZE     128
419  #define  SHA512_DER_SIZE       19
420  #define  SHA512_DER            \
421
     0x30,0x51,0x30,0x0D,0x06,0x09,0x60,0x86,0x48,0x01,0x65,0x03,0x04,0x02,0x03,0x05,0x00,0
     x04,0x40
```

From TCG Algorithm Registry: Table 16 - Defines for SM3_256 Hash Values

```
422  #define  SM3_256_DIGEST_SIZE    32
423  #define  SM3_256_BLOCK_SIZE     64
424  #define  SM3_256_DER_SIZE       18
425  #define  SM3_256_DER            \
426
     0x30,0x30,0x30,0x0C,0x06,0x08,0x2A,0x81,0x1C,0x81,0x45,0x01,0x83,0x11,0x05,0x00,0x04,0
     x20
```

From TCG Algorithm Registry: Table 17 - Defines for AES Symmetric Cipher Algorithm Constants

```
427  #define  AES_ALLOWED_KEY_SIZE_128    YES
428  #define  AES_ALLOWED_KEY_SIZE_192    YES
429  #define  AES_ALLOWED_KEY_SIZE_256    YES
430  #define  AES_128_BLOCK_SIZE_BYTES    16
431  #define  AES_192_BLOCK_SIZE_BYTES    16
432  #define  AES_256_BLOCK_SIZE_BYTES    16
```

From TCG Algorithm Registry: Table 18 - Defines for SM4 Symmetric Cipher Algorithm Constants

```
433   #define  SM4_ALLOWED_KEY_SIZE_128    YES
434   #define  SM4_128_BLOCK_SIZE_BYTES    16
```

From TCG Algorithm Registry: Table 19 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```
435   #define  CAMELLIA_ALLOWED_KEY_SIZE_128    YES
436   #define  CAMELLIA_ALLOWED_KEY_SIZE_192    YES
437   #define  CAMELLIA_ALLOWED_KEY_SIZE_256    YES
438   #define  CAMELLIA_128_BLOCK_SIZE_BYTES    16
439   #define  CAMELLIA_192_BLOCK_SIZE_BYTES    16
440   #define  CAMELLIA_256_BLOCK_SIZE_BYTES    16
```

From TPM 2.0 Part 2: Table 13 - Definition of TPM_CC Constants

```
441   typedef  UINT32             TPM_CC;
442   #define  TPM_CC_FIRST                          (TPM_CC)(0x0000011F)
443   #define  TPM_CC_PP_FIRST                       (TPM_CC)(0x0000011F)
444   #if defined CC_NV_UndefineSpaceSpecial && CC_NV_UndefineSpaceSpecial == YES
445   #define  TPM_CC_NV_UndefineSpaceSpecial        (TPM_CC)(0x0000011F)
446   #endif
447   #if defined CC_EvictControl && CC_EvictControl == YES
448   #define  TPM_CC_EvictControl                   (TPM_CC)(0x00000120)
449   #endif
450   #if defined CC_HierarchyControl && CC_HierarchyControl == YES
451   #define  TPM_CC_HierarchyControl               (TPM_CC)(0x00000121)
452   #endif
453   #if defined CC_NV_UndefineSpace && CC_NV_UndefineSpace == YES
454   #define  TPM_CC_NV_UndefineSpace               (TPM_CC)(0x00000122)
455   #endif
456   #if defined CC_ChangeEPS && CC_ChangeEPS == YES
457   #define  TPM_CC_ChangeEPS                      (TPM_CC)(0x00000124)
458   #endif
459   #if defined CC_ChangePPS && CC_ChangePPS == YES
460   #define  TPM_CC_ChangePPS                      (TPM_CC)(0x00000125)
461   #endif
462   #if defined CC_Clear && CC_Clear == YES
463   #define  TPM_CC_Clear                          (TPM_CC)(0x00000126)
464   #endif
465   #if defined CC_ClearControl && CC_ClearControl == YES
466   #define  TPM_CC_ClearControl                   (TPM_CC)(0x00000127)
467   #endif
468   #if defined CC_ClockSet && CC_ClockSet == YES
469   #define  TPM_CC_ClockSet                       (TPM_CC)(0x00000128)
470   #endif
471   #if defined CC_HierarchyChangeAuth && CC_HierarchyChangeAuth == YES
472   #define  TPM_CC_HierarchyChangeAuth            (TPM_CC)(0x00000129)
473   #endif
474   #if defined CC_NV_DefineSpace && CC_NV_DefineSpace == YES
475   #define  TPM_CC_NV_DefineSpace                 (TPM_CC)(0x0000012A)
476   #endif
477   #if defined CC_PCR_Allocate && CC_PCR_Allocate == YES
478   #define  TPM_CC_PCR_Allocate                   (TPM_CC)(0x0000012B)
479   #endif
480   #if defined CC_PCR_SetAuthPolicy && CC_PCR_SetAuthPolicy == YES
481   #define  TPM_CC_PCR_SetAuthPolicy              (TPM_CC)(0x0000012C)
482   #endif
483   #if defined CC_PP_Commands && CC_PP_Commands == YES
484   #define  TPM_CC_PP_Commands                    (TPM_CC)(0x0000012D)
485   #endif
486   #if defined CC_SetPrimaryPolicy && CC_SetPrimaryPolicy == YES
487   #define  TPM_CC_SetPrimaryPolicy               (TPM_CC)(0x0000012E)
488   #endif
489   #if defined CC_FieldUpgradeStart && CC_FieldUpgradeStart == YES
490   #define  TPM_CC_FieldUpgradeStart              (TPM_CC)(0x0000012F)
491   #endif
```



```
#if defined CC_ClockRateAdjust && CC_ClockRateAdjust == YES
#define  TPM_CC_ClockRateAdjust                (TPM_CC)(0x00000130)
#endif
#if defined CC_CreatePrimary && CC_CreatePrimary == YES
#define  TPM_CC_CreatePrimary                  (TPM_CC)(0x00000131)
#endif
#if defined CC_NV_GlobalWriteLock && CC_NV_GlobalWriteLock == YES
#define  TPM_CC_NV_GlobalWriteLock             (TPM_CC)(0x00000132)
#endif
#define  TPM_CC_PP_LAST                        (TPM_CC)(0x00000132)
#if defined CC_GetCommandAuditDigest && CC_GetCommandAuditDigest == YES
#define  TPM_CC_GetCommandAuditDigest          (TPM_CC)(0x00000133)
#endif
#if defined CC_NV_Increment && CC_NV_Increment == YES
#define  TPM_CC_NV_Increment                   (TPM_CC)(0x00000134)
#endif
#if defined CC_NV_SetBits && CC_NV_SetBits == YES
#define  TPM_CC_NV_SetBits                     (TPM_CC)(0x00000135)
#endif
#if defined CC_NV_Extend && CC_NV_Extend == YES
#define  TPM_CC_NV_Extend                      (TPM_CC)(0x00000136)
#endif
#if defined CC_NV_Write && CC_NV_Write == YES
#define  TPM_CC_NV_Write                       (TPM_CC)(0x00000137)
#endif
#if defined CC_NV_WriteLock && CC_NV_WriteLock == YES
#define  TPM_CC_NV_WriteLock                   (TPM_CC)(0x00000138)
#endif
#if defined CC_DictionaryAttackLockReset && CC_DictionaryAttackLockReset == YES
#define  TPM_CC_DictionaryAttackLockReset      (TPM_CC)(0x00000139)
#endif
#if defined CC_DictionaryAttackParameters && CC_DictionaryAttackParameters == YES
#define  TPM_CC_DictionaryAttackParameters     (TPM_CC)(0x0000013A)
#endif
#if defined CC_NV_ChangeAuth && CC_NV_ChangeAuth == YES
#define  TPM_CC_NV_ChangeAuth                  (TPM_CC)(0x0000013B)
#endif
#if defined CC_PCR_Event && CC_PCR_Event == YES
#define  TPM_CC_PCR_Event                      (TPM_CC)(0x0000013C)
#endif
#if defined CC_PCR_Reset && CC_PCR_Reset == YES
#define  TPM_CC_PCR_Reset                      (TPM_CC)(0x0000013D)
#endif
#if defined CC_SequenceComplete && CC_SequenceComplete == YES
#define  TPM_CC_SequenceComplete               (TPM_CC)(0x0000013E)
#endif
#if defined CC_SetAlgorithmSet && CC_SetAlgorithmSet == YES
#define  TPM_CC_SetAlgorithmSet                (TPM_CC)(0x0000013F)
#endif
#if defined CC_SetCommandCodeAuditStatus && CC_SetCommandCodeAuditStatus == YES
#define  TPM_CC_SetCommandCodeAuditStatus      (TPM_CC)(0x00000140)
#endif
#if defined CC_FieldUpgradeData && CC_FieldUpgradeData == YES
#define  TPM_CC_FieldUpgradeData               (TPM_CC)(0x00000141)
#endif
#if defined CC_IncrementalSelfTest && CC_IncrementalSelfTest == YES
#define  TPM_CC_IncrementalSelfTest            (TPM_CC)(0x00000142)
#endif
#if defined CC_SelfTest && CC_SelfTest == YES
#define  TPM_CC_SelfTest                       (TPM_CC)(0x00000143)
#endif
#if defined CC_Startup && CC_Startup == YES
#define  TPM_CC_Startup                        (TPM_CC)(0x00000144)
#endif
#if defined CC_Shutdown && CC_Shutdown == YES
#define  TPM_CC_Shutdown                       (TPM_CC)(0x00000145)
```

```
558   #endif
559   #if defined CC_StirRandom && CC_StirRandom == YES
560   #define  TPM_CC_StirRandom                    (TPM_CC)(0x00000146)
561   #endif
562   #if defined CC_ActivateCredential && CC_ActivateCredential == YES
563   #define  TPM_CC_ActivateCredential            (TPM_CC)(0x00000147)
564   #endif
565   #if defined CC_Certify && CC_Certify == YES
566   #define  TPM_CC_Certify                       (TPM_CC)(0x00000148)
567   #endif
568   #if defined CC_PolicyNV && CC_PolicyNV == YES
569   #define  TPM_CC_PolicyNV                      (TPM_CC)(0x00000149)
570   #endif
571   #if defined CC_CertifyCreation && CC_CertifyCreation == YES
572   #define  TPM_CC_CertifyCreation               (TPM_CC)(0x0000014A)
573   #endif
574   #if defined CC_Duplicate && CC_Duplicate == YES
575   #define  TPM_CC_Duplicate                     (TPM_CC)(0x0000014B)
576   #endif
577   #if defined CC_GetTime && CC_GetTime == YES
578   #define  TPM_CC_GetTime                       (TPM_CC)(0x0000014C)
579   #endif
580   #if defined CC_GetSessionAuditDigest && CC_GetSessionAuditDigest == YES
581   #define  TPM_CC_GetSessionAuditDigest         (TPM_CC)(0x0000014D)
582   #endif
583   #if defined CC_NV_Read && CC_NV_Read == YES
584   #define  TPM_CC_NV_Read                       (TPM_CC)(0x0000014E)
585   #endif
586   #if defined CC_NV_ReadLock && CC_NV_ReadLock == YES
587   #define  TPM_CC_NV_ReadLock                   (TPM_CC)(0x0000014F)
588   #endif
589   #if defined CC_ObjectChangeAuth && CC_ObjectChangeAuth == YES
590   #define  TPM_CC_ObjectChangeAuth              (TPM_CC)(0x00000150)
591   #endif
592   #if defined CC_PolicySecret && CC_PolicySecret == YES
593   #define  TPM_CC_PolicySecret                  (TPM_CC)(0x00000151)
594   #endif
595   #if defined CC_Rewrap && CC_Rewrap == YES
596   #define  TPM_CC_Rewrap                        (TPM_CC)(0x00000152)
597   #endif
598   #if defined CC_Create && CC_Create == YES
599   #define  TPM_CC_Create                        (TPM_CC)(0x00000153)
600   #endif
601   #if defined CC_ECDH_ZGen && CC_ECDH_ZGen == YES
602   #define  TPM_CC_ECDH_ZGen                     (TPM_CC)(0x00000154)
603   #endif
604   #if defined CC_HMAC && CC_HMAC == YES
605   #define  TPM_CC_HMAC                          (TPM_CC)(0x00000155)
606   #endif
607   #if defined CC_Import && CC_Import == YES
608   #define  TPM_CC_Import                        (TPM_CC)(0x00000156)
609   #endif
610   #if defined CC_Load && CC_Load == YES
611   #define  TPM_CC_Load                          (TPM_CC)(0x00000157)
612   #endif
613   #if defined CC_Quote && CC_Quote == YES
614   #define  TPM_CC_Quote                         (TPM_CC)(0x00000158)
615   #endif
616   #if defined CC_RSA_Decrypt && CC_RSA_Decrypt == YES
617   #define  TPM_CC_RSA_Decrypt                   (TPM_CC)(0x00000159)
618   #endif
619   #if defined CC_HMAC_Start && CC_HMAC_Start == YES
620   #define  TPM_CC_HMAC_Start                    (TPM_CC)(0x0000015B)
621   #endif
622   #if defined CC_SequenceUpdate && CC_SequenceUpdate == YES
623   #define  TPM_CC_SequenceUpdate                (TPM_CC)(0x0000015C)
```

```
624   #endif
625   #if defined CC_Sign && CC_Sign == YES
626   #define  TPM_CC_Sign                          (TPM_CC)(0x0000015D)
627   #endif
628   #if defined CC_Unseal && CC_Unseal == YES
629   #define  TPM_CC_Unseal                        (TPM_CC)(0x0000015E)
630   #endif
631   #if defined CC_PolicySigned && CC_PolicySigned == YES
632   #define  TPM_CC_PolicySigned                  (TPM_CC)(0x00000160)
633   #endif
634   #if defined CC_ContextLoad && CC_ContextLoad == YES
635   #define  TPM_CC_ContextLoad                   (TPM_CC)(0x00000161)
636   #endif
637   #if defined CC_ContextSave && CC_ContextSave == YES
638   #define  TPM_CC_ContextSave                   (TPM_CC)(0x00000162)
639   #endif
640   #if defined CC_ECDH_KeyGen && CC_ECDH_KeyGen == YES
641   #define  TPM_CC_ECDH_KeyGen                   (TPM_CC)(0x00000163)
642   #endif
643   #if defined CC_EncryptDecrypt && CC_EncryptDecrypt == YES
644   #define  TPM_CC_EncryptDecrypt                (TPM_CC)(0x00000164)
645   #endif
646   #if defined CC_FlushContext && CC_FlushContext == YES
647   #define  TPM_CC_FlushContext                  (TPM_CC)(0x00000165)
648   #endif
649   #if defined CC_LoadExternal && CC_LoadExternal == YES
650   #define  TPM_CC_LoadExternal                  (TPM_CC)(0x00000167)
651   #endif
652   #if defined CC_MakeCredential && CC_MakeCredential == YES
653   #define  TPM_CC_MakeCredential                (TPM_CC)(0x00000168)
654   #endif
655   #if defined CC_NV_ReadPublic && CC_NV_ReadPublic == YES
656   #define  TPM_CC_NV_ReadPublic                 (TPM_CC)(0x00000169)
657   #endif
658   #if defined CC_PolicyAuthorize && CC_PolicyAuthorize == YES
659   #define  TPM_CC_PolicyAuthorize               (TPM_CC)(0x0000016A)
660   #endif
661   #if defined CC_PolicyAuthValue && CC_PolicyAuthValue == YES
662   #define  TPM_CC_PolicyAuthValue               (TPM_CC)(0x0000016B)
663   #endif
664   #if defined CC_PolicyCommandCode && CC_PolicyCommandCode == YES
665   #define  TPM_CC_PolicyCommandCode             (TPM_CC)(0x0000016C)
666   #endif
667   #if defined CC_PolicyCounterTimer && CC_PolicyCounterTimer == YES
668   #define  TPM_CC_PolicyCounterTimer            (TPM_CC)(0x0000016D)
669   #endif
670   #if defined CC_PolicyCpHash && CC_PolicyCpHash == YES
671   #define  TPM_CC_PolicyCpHash                  (TPM_CC)(0x0000016E)
672   #endif
673   #if defined CC_PolicyLocality && CC_PolicyLocality == YES
674   #define  TPM_CC_PolicyLocality                (TPM_CC)(0x0000016F)
675   #endif
676   #if defined CC_PolicyNameHash && CC_PolicyNameHash == YES
677   #define  TPM_CC_PolicyNameHash                (TPM_CC)(0x00000170)
678   #endif
679   #if defined CC_PolicyOR && CC_PolicyOR == YES
680   #define  TPM_CC_PolicyOR                      (TPM_CC)(0x00000171)
681   #endif
682   #if defined CC_PolicyTicket && CC_PolicyTicket == YES
683   #define  TPM_CC_PolicyTicket                  (TPM_CC)(0x00000172)
684   #endif
685   #if defined CC_ReadPublic && CC_ReadPublic == YES
686   #define  TPM_CC_ReadPublic                    (TPM_CC)(0x00000173)
687   #endif
688   #if defined CC_RSA_Encrypt && CC_RSA_Encrypt == YES
689   #define  TPM_CC_RSA_Encrypt                   (TPM_CC)(0x00000174)
```

```
690   #endif
691   #if defined CC_StartAuthSession && CC_StartAuthSession == YES
692   #define  TPM_CC_StartAuthSession              (TPM_CC)(0x00000176)
693   #endif
694   #if defined CC_VerifySignature && CC_VerifySignature == YES
695   #define  TPM_CC_VerifySignature               (TPM_CC)(0x00000177)
696   #endif
697   #if defined CC_ECC_Parameters && CC_ECC_Parameters == YES
698   #define  TPM_CC_ECC_Parameters                (TPM_CC)(0x00000178)
699   #endif
700   #if defined CC_FirmwareRead && CC_FirmwareRead == YES
701   #define  TPM_CC_FirmwareRead                  (TPM_CC)(0x00000179)
702   #endif
703   #if defined CC_GetCapability && CC_GetCapability == YES
704   #define  TPM_CC_GetCapability                 (TPM_CC)(0x0000017A)
705   #endif
706   #if defined CC_GetRandom && CC_GetRandom == YES
707   #define  TPM_CC_GetRandom                     (TPM_CC)(0x0000017B)
708   #endif
709   #if defined CC_GetTestResult && CC_GetTestResult == YES
710   #define  TPM_CC_GetTestResult                 (TPM_CC)(0x0000017C)
711   #endif
712   #if defined CC_Hash && CC_Hash == YES
713   #define  TPM_CC_Hash                          (TPM_CC)(0x0000017D)
714   #endif
715   #if defined CC_PCR_Read && CC_PCR_Read == YES
716   #define  TPM_CC_PCR_Read                      (TPM_CC)(0x0000017E)
717   #endif
718   #if defined CC_PolicyPCR && CC_PolicyPCR == YES
719   #define  TPM_CC_PolicyPCR                     (TPM_CC)(0x0000017F)
720   #endif
721   #if defined CC_PolicyRestart && CC_PolicyRestart == YES
722   #define  TPM_CC_PolicyRestart                 (TPM_CC)(0x00000180)
723   #endif
724   #if defined CC_ReadClock && CC_ReadClock == YES
725   #define  TPM_CC_ReadClock                     (TPM_CC)(0x00000181)
726   #endif
727   #if defined CC_PCR_Extend && CC_PCR_Extend == YES
728   #define  TPM_CC_PCR_Extend                    (TPM_CC)(0x00000182)
729   #endif
730   #if defined CC_PCR_SetAuthValue && CC_PCR_SetAuthValue == YES
731   #define  TPM_CC_PCR_SetAuthValue              (TPM_CC)(0x00000183)
732   #endif
733   #if defined CC_NV_Certify && CC_NV_Certify == YES
734   #define  TPM_CC_NV_Certify                    (TPM_CC)(0x00000184)
735   #endif
736   #if defined CC_EventSequenceComplete && CC_EventSequenceComplete == YES
737   #define  TPM_CC_EventSequenceComplete         (TPM_CC)(0x00000185)
738   #endif
739   #if defined CC_HashSequenceStart && CC_HashSequenceStart == YES
740   #define  TPM_CC_HashSequenceStart             (TPM_CC)(0x00000186)
741   #endif
742   #if defined CC_PolicyPhysicalPresence && CC_PolicyPhysicalPresence == YES
743   #define  TPM_CC_PolicyPhysicalPresence        (TPM_CC)(0x00000187)
744   #endif
745   #if defined CC_PolicyDuplicationSelect && CC_PolicyDuplicationSelect == YES
746   #define  TPM_CC_PolicyDuplicationSelect       (TPM_CC)(0x00000188)
747   #endif
748   #if defined CC_PolicyGetDigest && CC_PolicyGetDigest == YES
749   #define  TPM_CC_PolicyGetDigest               (TPM_CC)(0x00000189)
750   #endif
751   #if defined CC_TestParms && CC_TestParms == YES
752   #define  TPM_CC_TestParms                     (TPM_CC)(0x0000018A)
753   #endif
754   #if defined CC_Commit && CC_Commit == YES
755   #define  TPM_CC_Commit                        (TPM_CC)(0x0000018B)
```

```
#endif
#if defined CC_PolicyPassword && CC_PolicyPassword == YES
#define  TPM_CC_PolicyPassword                (TPM_CC)(0x0000018C)
#endif
#if defined CC_ZGen_2Phase && CC_ZGen_2Phase == YES
#define  TPM_CC_ZGen_2Phase                   (TPM_CC)(0x0000018D)
#endif
#if defined CC_EC_Ephemeral && CC_EC_Ephemeral == YES
#define  TPM_CC_EC_Ephemeral                  (TPM_CC)(0x0000018E)
#endif
#if defined CC_PolicyNvWritten && CC_PolicyNvWritten == YES
#define  TPM_CC_PolicyNvWritten               (TPM_CC)(0x0000018F)
#endif
#define  TPM_CC_LAST                          (TPM_CC)(0x0000018F)
#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif
#define MAX_HASH_BLOCK_SIZE  (                  \
    MAX(ALG_SHA1 * SHA1_BLOCK_SIZE,             \
    MAX(ALG_SHA256 * SHA256_BLOCK_SIZE,         \
    MAX(ALG_SHA384 * SHA384_BLOCK_SIZE,         \
    MAX(ALG_SM3_256 * SM3_256_BLOCK_SIZE,       \
    MAX(ALG_SHA512 * SHA512_BLOCK_SIZE,         \
    0 ))))))
#define MAX_DIGEST_SIZE      (                  \
    MAX(ALG_SHA1 * SHA1_DIGEST_SIZE,            \
    MAX(ALG_SHA256 * SHA256_DIGEST_SIZE,        \
    MAX(ALG_SHA384 * SHA384_DIGEST_SIZE,        \
    MAX(ALG_SM3_256 * SM3_256_DIGEST_SIZE,      \
    MAX(ALG_SHA512 * SHA512_DIGEST_SIZE,        \
    0 ))))))
#if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
#error "Hash data not valid"
#endif
#define HASH_COUNT (ALG_SHA1+ALG_SHA256+ALG_SHA384+ALG_SM3_256+ALG_SHA512)
```

Define the 2B structure that would hold any hash block

```
TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
```

Folloing typedef is for some old code

```
typedef TPM2B_MAX_HASH_BLOCK    TPM2B_HASH_BLOCK;
#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif
#ifndef ALG_CAMELLIA
#   define ALG_CAMELLIA         NO
#endif
#ifndef MAX_CAMELLIA_KEY_BITS
#   define      MAX_CAMELLIA_KEY_BITS  0
#   define      MAX_CAMELLIA_BLOCK_SIZE_BYTES 0
#endif
#ifndef ALG_SM4
#   define ALG_SM4          NO
#endif
#ifndef MAX_SM4_KEY_BITS
#   define      MAX_SM4_KEY_BITS  0
#   define      MAX_SM4_BLOCK_SIZE_BYTES 0
#endif
#ifndef ALG_AES
#   define ALG_AES          NO
#endif
#ifndef MAX_AES_KEY_BITS
#   define      MAX_AES_KEY_BITS  0
```

```
#   define      MAX_AES_BLOCK_SIZE_BYTES 0
#endif
#define MAX_SYM_KEY_BITS (                                  \
            MAX(MAX_CAMELLIA_KEY_BITS * ALG_CAMELLIA,           \
            MAX(MAX_SM4_KEY_BITS * ALG_SM4,           \
            MAX(MAX_AES_KEY_BITS * ALG_AES,           \
            0))))
#define  MAX_SYM_KEY_BYTES ((MAX_SYM_KEY_BITS + 7) / 8)
#define MAX_SYM_BLOCK_SIZE  (                               \
            MAX(MAX_CAMELLIA_BLOCK_SIZE_BYTES * ALG_CAMELLIA,   \
            MAX(MAX_SM4_BLOCK_SIZE_BYTES * ALG_SM4,   \
            MAX(MAX_AES_BLOCK_SIZE_BYTES * ALG_AES,   \
            0))))
#if MAX_SYM_KEY_BITS == 0 || MAX_SYM_BLOCK_SIZE == 0
#   error Bad size for MAX_SYM_KEY_BITS or MAX_SYM_BLOCK_SIZE
#endif
```

Define the 2B structure for a seed

```
TPM2B_TYPE(SEED, PRIMARY_SEED_SIZE);
#endif  // _IMPLEMENTATION_H_
```

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 Integer Format

The big integers passed to/from the function interfaces in the crypto engine are in BYTE buffers that have the same format used in the TPM 2.0 specification that states:

"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.3 CryptoEngine.h

B.3.1. Introduction

This file contains constant definition shared by CryptUtil() and the parts of the Crypto Engine.

```
1   #ifndef _CRYPT_PRI_H
2   #define _CRYPT_PRI_H
3   #include    <stddef.h>
4   #include    "TpmBuildSwitches.h"
5   #include    "BaseTypes.h"
6   #include    "TpmError.h"
7   #include    "swap.h"
8   #include    "Implementation.h"
9   #include    "TPM_types.h"
10  //#include    "TPMB.h"
11  #include    "bool.h"
12  #include    "Platform.h"
13  #ifndef NULL
14  #define NULL    0
15  #endif
16  typedef UINT16  NUMBYTES;       // When a size is a number of bytes
17  typedef UINT32  NUMDIGITS;      // When a size is a number of "digits"
```

B.3.2. General Purpose Macros

```
#ifndef MAX
#   define MAX(a, b) ((a) > (b) ? (a) : b)
#endif
```

This is the definition of a bit array with one bit per algorithm

```
typedef BYTE    ALGORITHM_VECTOR[(ALG_LAST_VALUE + 7) / 8];
```

B.3.3. Self-test

This structure is used to contain self-test tracking information for the crypto engine. Each of the major modules is given a 32-bit value in which it may maintain its own self test information. The convention for this state is that when all of the bits in this structure are 0, all functions need to be tested.

```
typedef struct {
    UINT32      rng;
    UINT32      hash;
    UINT32      sym;
#ifdef TPM_ALG_RSA
    UINT32      rsa;
#endif
#ifdef  TPM_ALG_ECC
    UINT32      ecc;
#endif
} CRYPTO_SELF_TEST_STATE;
```

B.3.4. Hash-related Structures

```
typedef struct {
    const TPM_ALG_ID      alg;
    const NUMBYTES        digestSize;
    const NUMBYTES        blockSize;
    const NUMBYTES        derSize;
    const BYTE            der[20];
} HASH_INFO;
```

This value will change with each implementation. The value of 16 is used to account for any slop in the context values. The overall size needs to be as large as any of the hash contexts. The structure needs to start on an alignment boundary and be an even multiple of the alignment

```
#define ALIGNED_SIZE(x, b) ((((x) + (b) - 1) / (b)) * (b))
#define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
#define MAX_HASH_STATE_SIZE_ALIGNED                                          \
                    ALIGNED_SIZE(MAX_HASH_STATE_SIZE, CRYPTO_ALIGNMENT)
```

This is an byte array that will hold any of the hash contexts.

```
typedef CRYPTO_ALIGNED BYTE ALIGNED_HASH_STATE[MAX_HASH_STATE_SIZE_ALIGNED];
```

Macro to align an address to the next higher size

```
#define AlignPointer(address, align)                                         \
   ((((intptr_t)&(address)) + (align - 1)) & ~(align - 1))
```

Macro to test alignment

```
#define IsAddressAligned(address, align)                                     \
                    (((intptr_t)(address) & (align - 1)) == 0)
```

This is the structure that is used for passing a context into the hashing functions. It should be the same size as the function context used within the hashing functions. This is checked when the hash function is initialized. This version uses a new layout for the contexts and a different definition. The state buffer is an array of HASH_UNIT values so that a decent compiler will put the structure on a HASH_UNIT boundary. If the structure is not properly aligned, the code that manipulates the structure will copy to a properly aligned structure before it is used and copy the result back. This just makes things slower.

```
49  typedef struct _HASH_STATE
50  {
51      ALIGNED_HASH_STATE       state;
52      TPM_ALG_ID               hashAlg;
53  } CPRI_HASH_STATE, *PCPRI_HASH_STATE;
54  extern const HASH_INFO   g_hashData[HASH_COUNT + 1];
```

This is for the external hash state. This implementation assumes that the size of the exported hash state is no larger than the internal hash state. There is a compile-time check to make sure that this is true.

```
55  typedef struct {
56      ALIGNED_HASH_STATE      buffer;
57      TPM_ALG_ID              hashAlg;
58  } EXPORT_HASH_STATE;
59  typedef enum {
60      IMPORT_STATE,       // Converts externally formatted state to internal
61      EXPORT_STATE        // Converts internal formatted state to external
62  } IMPORT_EXPORT;
```

Values and structures for the random number generator. These values are defined in this header file so that the size of the RNG state can be known to TPM.lib. This allows the allocation of some space in NV memory for the state to be stored on an orderly shutdown. The GET_PUT enum is used by _cpri__DrbgGetPutState() to indicate the direction of data flow.

```
63  typedef enum {
64      GET_STATE,      // Get the state to save to NV
65      PUT_STATE       // Restore the state from NV
66  } GET_PUT;
```

The DRBG based on a symmetric block cipher is defined by three values,

a) the key size

b) the block size (the IV size)

c) the symmetric algorithm

```
67  #define DRBG_KEY_SIZE_BITS      MAX_AES_KEY_BITS
68  #define DRBG_IV_SIZE_BITS       (MAX_AES_BLOCK_SIZE_BYTES * 8)
69  #define DRBG_ALGORITHM          TPM_ALG_AES
70  #if ((DRBG_KEY_SIZE_BITS % 8) != 0) || ((DRBG_IV_SIZE_BITS % 8) != 0)
71  #error "Key size and IV for DRBG must be even multiples of 8"
72  #endif
73  #if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
74  #error "Key size for DRBG must be even multiple of the cypher block size"
75  #endif
76  typedef UINT32    DRBG_SEED[(DRBG_KEY_SIZE_BITS + DRBG_IV_SIZE_BITS) / 32];
77  typedef struct {
78      UINT64      reseedCounter;
79      UINT32      magic;
80      DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
81      UINT32      lastValue[4];   // used when the TPM does continuous self-test
82                                  // for FIPS compliance of DRBG
83  } DRBG_STATE, *pDRBG_STATE;
```

B.3.5. Asymmetric Structures and Values

```
#ifdef TPM_ALG_ECC
```

B.3.5.1. ECC-related Structures

This structure replicates the structure definition in TPM_Types.h. It is duplicated to avoid inclusion of all of TPM_Types.h This structure is similar to the RSA_KEY structure below. The purpose of these structures is to reduce the overhead of a function call and to make the code less dependent on key types as much as possible.

```
typedef struct {
    UINT32                  curveID;       // The curve identifier
    TPMS_ECC_POINT         *publicPoint;   // Pointer to the public point
    TPM2B_ECC_PARAMETER    *privateKey;    // Pointer to the private key
} ECC_KEY;
#endif // TPM_ALG_ECC
#ifdef TPM_ALG_RSA
```

B.3.5.2. RSA-related Structures

This structure is a succinct representation of the cryptographic components of an RSA key.

```
typedef struct {
    UINT32         exponent;      // The public exponent pointer
    TPM2B         *publicKey;     // Pointer to the public modulus
    TPM2B         *privateKey;    // The private exponent (not a prime)
} RSA_KEY;
#endif // TPM_ALG_RSA
```

B.3.6. Miscelaneous

```
#ifdef TPM_ALG_RSA
#   ifdef TPM_ALG_ECC
#       if   MAX_RSA_KEY_BYTES > MAX_ECC_KEY_BYTES
#           define  MAX_NUMBER_SIZE         MAX_RSA_KEY_BYTES
#       else
#           define  MAX_NUMBER_SIZE         MAX_ECC_KEY_BYTES
#       endif
#   else // RSA but no ECC
#       define MAX_NUMBER_SIZE              MAX_RSA_KEY_BYTES
#   endif
#elif defined TPM_ALG_ECC
#   define MAX_NUMBER_SIZE                 MAX_ECC_KEY_BYTES
#else
#   error No assymmetric algorithm implemented.
#endif
typedef INT16     CRYPT_RESULT;
#define CRYPT_RESULT_MIN    INT16_MIN
#define CRYPT_RESULT_MAX    INT16_MAX
```

< 0	recoverable error
0	success
> 0	command specific return value (generally a digest size)

```
#define CRYPT_FAIL          ((CRYPT_RESULT)  1)
#define CRYPT_SUCCESS       ((CRYPT_RESULT)  0)
#define CRYPT_NO_RESULT     ((CRYPT_RESULT) -1)
```



```
119   #define CRYPT_SCHEME        ((CRYPT_RESULT) -2)
120   #define CRYPT_PARAMETER     ((CRYPT_RESULT) -3)
121   #define CRYPT_UNDERFLOW     ((CRYPT_RESULT) -4)
122   #define CRYPT_POINT         ((CRYPT_RESULT) -5)
123   #define CRYPT_CANCEL        ((CRYPT_RESULT) -6)
124   typedef UINT64              HASH_CONTEXT[MAX_HASH_STATE_SIZE/sizeof(UINT64)];
125   #include    "CpriCryptPri_fp.h"
126   #ifdef  TPM_ALG_ECC
127   #   include "CpriDataEcc.h"
128   #   include "CpriECC_fp.h"
129   #endif
130   #include    "MathFunctions_fp.h"
131   #include    "CpriRNG_fp.h"
132   #include    "CpriHash_fp.h"
133   #include    "CpriSym_fp.h"
134   #ifdef  TPM_ALG_RSA
135   #   include    "CpriRSA_fp.h"
136   #endif
137   #endif // !_CRYPT_PRI_H
```


B.4 OsslCryptoEngine.h

B.4.1. Introduction

This is the header file used by the components of the CryptoEngine(). This file should not be included in any file other than the files in the crypto engine.

Vendors may replace the implementation in this file by a local crypto engine. The implementation in this file is based on OpenSSL() library. Integer format: the big integers passed in/out the function interfaces in this library by a byte buffer (BYTE *) adopt the same format used in 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.4.2. Defines

```
#ifndef _OSSL_CRYPTO_ENGINE_H
#define _OSSL_CRYPTO_ENGINE_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     CRYPTO_ENGINE
#include "CryptoEngine.h"
#include "CpriMisc_fp.h"
#define MAX_ECC_PARAMETER_BYTES 32
#define MAX_2B_BYTES MAX((MAX_RSA_KEY_BYTES * ALG_RSA),              \
                         MAX((MAX_ECC_PARAMETER_BYTES * ALG_ECC),  \
                             MAX_DIGEST_SIZE))
#define assert2Bsize(a) pAssert((a).size <= sizeof((a).buffer))
#ifdef TPM_ALG_RSA
#   ifdef   RSA_KEY_SIEVE
#       include     "RsaKeySieve.h"
#       include     "RsaKeySieve_fp.h"
#   endif
#   include    "CpriRSA_fp.h"
#endif
```

This is a structure to hold the parameters for the version of KDFa() used by the CryptoEngine(). This structure allows the state to be passed between multiple functions that use the same pseudo-random sequence.

```
typedef struct {
    CPRI_HASH_STATE          iPadCtx;
    CPRI_HASH_STATE          oPadCtx;
    TPM2B                   *extra;
    UINT32                  *outer;
    TPM_ALG_ID               hashAlg;
    UINT16                   keySizeInBits;
} KDFa_CONTEXT;
#endif // _OSSL_CRYPTO_ENGINE_H
```

B.5 MathFunctions.c

B.5.1. Introduction

This file contains implementation of some of the big number primitives. This is used in order to reduce the overhead in dealing with data conversions to standard big number format.

The simulator code uses the canonical form whenever possible in order to make the code in Part 3 more accessible. The canonical data formats are simple and not well suited for complex big number computations. This library provides functions that are found in typical big number libraries but they are written to handle the canonical data format of the reference TPM.

In some cases, data is converted to a big number format used by a standard library, such as OpenSSL(). This is done when the computations are complex enough warrant conversion. Vendors may replace the implementation in this file with a library that provides equivalent functions. A vendor may also rewrite the TPM code so that it uses a standard big number format instead of the canonical form and use the standard libraries instead of the code in this file.

The implementation in this file makes use of the OpenSSL() library.

Integer format: integers passed through the function interfaces in this library adopt the same format used in TPM 2.0 specification. It defines an integer as "an array of one or more octets with the most significant octet at the lowest index of the array." An additional value is needed to indicate the number of significant bytes.

```
1   #include "OsslCryptoEngine.h"
```

B.5.2. Externally Accessible Functions

B.5.2.1. _math__Normalize2B()

This function will normalize the value in a TPM2B. If there are **leading** bytes of zero, the first non-zero byte is shifted up.

Return Value	Meaning
0	no significant bytes, value is zero
>0	number of significant bytes

```
2   LIB_EXPORT UINT16
3   _math__Normalize2B(
4       TPM2B           *b              // IN/OUT: number to normalize
5       )
6   {
7       UINT16      from;
8       UINT16      to;
9       UINT16      size = b->size;
10
11      for(from = 0; b->buffer[from] == 0 && from < size; from++);
12      b->size -= from;
13      for(to = 0; from < size; to++, from++ )
14          b->buffer[to] = b->buffer[from];
15      return b->size;
16  }
```

B.5.2.2. _math__Denormalize2B()

This function is used to adjust a TPM2B so that the number has the desired number of bytes. This is accomplished by adding bytes of zero at the start of the number.

Return Value	Meaning
TRUE	number de-normalized
FALSE	number already larger than the desired size

```
17  LIB_EXPORT BOOL
18  _math__Denormalize2B(
19      TPM2B           *in,            // IN:OUT TPM2B number to de-normalize
20      UINT32           size           // IN: the desired size
21      )
22  {
23      UINT32       to;
24      UINT32       from;
25      // If the current size is greater than the requested size, see if this can be
26      // normalized to a value smaller than the requested size and then de-normalize
27      if(in->size > size)
28      {
29          _math__Normalize2B(in);
30          if(in->size > size)
31              return FALSE;
32      }
33      // If the size is already what is requested, leave
34      if(in->size == size)
35          return TRUE;
36
37      // move the bytes to the 'right'
38      for(from = in->size, to = size; from > 0;)
39          in->buffer[--to] = in->buffer[--from];
40
41      // 'to' will always be greater than 0 because we checked for equal above.
42      for(; to > 0;)
43          in->buffer[--to] = 0;
44
45      in->size = (UINT16)size;
46      return TRUE;
47  }
```

B.5.2.3. _math__sub()

This function to subtract one unsigned value from another *c* = *a* - *b*. *c* may be the same as *a* or *b*.

Return Value	Meaning
1	if (a > b) so no borrow
0	if (a = b) so no borrow and b == a
-1	if (a < b) so there was a borrow

```
48  LIB_EXPORT int
49  _math__sub(
50      const UINT32     aSize,         // IN: size of a
51      const BYTE      *a,             // IN: a
52      const UINT32     bSize,         // IN: size of b
53      const BYTE      *b,             // IN: b
54      UINT16          *cSize,         // OUT: set to MAX(aSize, bSize)
55      BYTE            *c              // OUT: the difference
56      )
```

```
{
    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.5.2.4. _math__Inc()

This function increments a large, big-endian number value by one.

Return Value	Meaning
0	result is zero
!0	result is not zero

```
LIB_EXPORT int
_math__Inc(
    UINT32           aSize,         // IN: size of a
    BYTE            *a              // IN: a
    )
{
```

```
110
111         for(a = &a[aSize-1];aSize > 0; aSize--)
112         {
113             if((*a-- += 1) != 0)
114                 return 1;
115         }
116         return 0;
117     }
```

B.5.2.5. _math__Dec()

This function decrements a large, ENDIAN value by one.

```
118     LIB_EXPORT void
119     _math__Dec(
120         UINT32           aSize,         // IN: size of a
121         BYTE            *a              // IN: a
122         )
123     {
124         for(a = &a[aSize-1]; aSize > 0; aSize--)
125         {
126             if((*a-- -= 1) != 0xff)
127                 return;
128         }
129         return;
130     }
```

B.5.2.6. _math__Mul()

This function is used to multiply two large integers: *p* = *a* b*. If the size of *p* is not specified *(pSize* == NULL), the size of the results *p* is assumed to be *aSize* + *bSize* and the results are de-normalized so that the resulting size is exactly *aSize* + *bSize*. If *pSize* is provided, then the actual size of the result is returned. The initial value for *pSize* must be at least *aSize* + *pSize*.

Return Value	Meaning
< 0	indicates an error
>= 0	the size of the product

```
131     LIB_EXPORT int
132     _math__Mul(
133         const UINT32     aSize,         // IN: size of a
134         const BYTE      *a,             // IN: a
135         const UINT32     bSize,         // IN: size of b
136         const BYTE      *b,             // IN: b
137         UINT32          *pSize,         // IN/OUT: size of the product
138         BYTE            *p              // OUT: product. length of product = aSize +
139                                         //      bSize
140         )
141     {
142         BIGNUM          *bnA;
143         BIGNUM          *bnB;
144         BIGNUM          *bnP;
145         BN_CTX          *context;
146         int              retVal = 0;
147
148         // First check that pSize is large enough if present
149         if((pSize != NULL) && (*pSize < (aSize + bSize)))
150             return CRYPT_PARAMETER;
151         pAssert(pSize == NULL || *pSize <= MAX_2B_BYTES);
152         //
```

```
153         // Allocate space for BIGNUM context
154         //
155         context = BN_CTX_new();
156         if(context == NULL)
157             FAIL(FATAL_ERROR_ALLOCATION);
158         bnA = BN_CTX_get(context);
159         bnB = BN_CTX_get(context);
160         bnP = BN_CTX_get(context);
161         if (bnP == NULL)
162             FAIL(FATAL_ERROR_ALLOCATION);
163
164         // Convert the inputs to BIGNUMs
165         //
166         if (BN_bin2bn(a, aSize, bnA) == NULL || BN_bin2bn(b, bSize, bnB) == NULL)
167             FAIL(FATAL_ERROR_INTERNAL);
168
169         // Perform the multiplication
170         //
171         if (BN_mul(bnP, bnA, bnB, context) != 1)
172             FAIL(FATAL_ERROR_INTERNAL);
173
174         // If the size of the results is allowed to float, then set the return
175         // size. Otherwise, it might be necessary to de-normalize the results
176         retVal = BN_num_bytes(bnP);
177         if(pSize == NULL)
178         {
179             BN_bn2bin(bnP, &p[aSize + bSize - retVal]);
180             memset(p, 0, aSize + bSize - retVal);
181             retVal = aSize + bSize;
182         }
183         else
184         {
185             BN_bn2bin(bnP, p);
186             *pSize = retVal;
187         }
188
189         BN_CTX_end(context);
190         BN_CTX_free(context);
191         return retVal;
192     }
```

B.5.2.7. _math__Div()

Divide an integer *(n)* by an integer *(d)* producing a quotient *(q)* and a remainder *(r)*. If *q* or *r* is not needed, then the pointer to them may be set to NULL.

Return Value	Meaning
CRYPT_SUCCESS	operation complete
CRYPT_UNDERFLOW	*q* or *r* is too small to receive the result

```
193     LIB_EXPORT CRYPT_RESULT
194     _math__Div(
195         const TPM2B     *n,             // IN: numerator
196         const TPM2B     *d,             // IN: denominator
197         TPM2B           *q,             // OUT: quotient
198         TPM2B           *r              // OUT: remainder
199         )
200     {
201         BIGNUM          *bnN;
202         BIGNUM          *bnD;
203         BIGNUM          *bnQ;
204         BIGNUM          *bnR;
```

```
205         BN_CTX          *context;
206         CRYPT_RESULT     retVal = CRYPT_SUCCESS;
207
208         // Get structures for the big number representations
209         context = BN_CTX_new();
210         if(context == NULL)
211             FAIL(FATAL_ERROR_ALLOCATION);
212         BN_CTX_start(context);
213         bnN = BN_CTX_get(context);
214         bnD = BN_CTX_get(context);
215         bnQ = BN_CTX_get(context);
216         bnR = BN_CTX_get(context);
217
218         // Errors in BN_CTX_get() are sticky so only need to check the last allocation
219         if (   bnR == NULL
220             || BN_bin2bn(n->buffer, n->size, bnN) == NULL
221             || BN_bin2bn(d->buffer, d->size, bnD) == NULL)
222                 FAIL(FATAL_ERROR_INTERNAL);
223
224         // Check for divide by zero.
225         if(BN_num_bits(bnD) == 0)
226             FAIL(FATAL_ERROR_DIVIDE_ZERO);
227
228         // Perform the division
229         if (BN_div(bnQ, bnR, bnN, bnD, context) != 1)
230             FAIL(FATAL_ERROR_INTERNAL);
231
232         // Convert the BIGNUM result back to our format
233         if(q != NULL)   // If the quotient is being returned
234         {
235             if(!BnTo2B(q, bnQ, q->size))
236             {
237                 retVal = CRYPT_UNDERFLOW;
238                 goto Done;
239             }
240          }
241         if(r != NULL)   // If the remainder is being returned
242         {
243             if(!BnTo2B(r, bnR, r->size))
244                 retVal = CRYPT_UNDERFLOW;
245         }
246
247     Done:
248         BN_CTX_end(context);
249         BN_CTX_free(context);
250
251         return retVal;
252     }
```

B.5.2.8. _math__uComp()

This function compare two unsigned values.

Return Value	Meaning
1	if (a > b)
0	if (a = b)
-1	if (a < b)

```
253     LIB_EXPORT int
254     _math__uComp(
255         const UINT32     aSize,          // IN: size of a
256         const BYTE      *a,              // IN: a
```

```
257         const UINT32     bSize,          // IN: size of b
258         const BYTE      *b               // IN: b
259         )
260     {
261         int             borrow = 0;
262         int             notZero = 0;
263         int             i;
264         // If a has more digits than b, then a is greater than b if
265         // any of the more significant bytes is non zero
266         if((i = (int)aSize - (int)bSize) > 0)
267             for(; i > 0; i--)
268                 if(*a++) // means a > b
269                     return 1;
270         // If b has more digits than a, then b is greater if any of the
271         // more significant bytes is non zero
272         if(i < 0)  // Means that b is longer than a
273             for(; i < 0; i++)
274                 if(*b++) // means that b > a
275                     return -1;
276         // Either the vales are the same size or the upper bytes of a or b are
277         // all zero, so compare the rest
278         i = (aSize > bSize) ? bSize : aSize;
279         a = &a[i-1];
280         b = &b[i-1];
281         for(; i > 0; i--)
282         {
283             borrow = *a-- - *b-- + borrow;
284             notZero = notZero || borrow;
285             borrow >>= 8;
286         }
287         // if there is a borrow, then b > a
288         if(borrow)
289             return -1;
290         // either a > b or they are the same
291         return notZero;
292     }
```

B.5.2.9. _math__Comp()

Compare two signed integers:

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
293     LIB_EXPORT int
294     _math__Comp(
295         const UINT32     aSize,          // IN: size of a
296         const BYTE      *a,              // IN: a buffer
297         const UINT32     bSize,          // IN: size of b
298         const BYTE      *b               // IN: b buffer
299         )
300     {
301         int      signA, signB;       // sign of a and b
302
303         // For positive or 0, sign_a is 1
304         // for negative, sign_a is 0
305         signA = ((a[0] & 0x80) == 0) ? 1 : 0;
306
307         // For positive or 0, sign_b is 1
308         // for negative, sign_b is 0
```

```
309         signB = ((b[0] & 0x80) == 0) ? 1 : 0;
310
311         if(signA != signB)
312         {
313             return signA - signB;
314         }
315
316         if(signA == 1)
317             // do unsigned compare function
318             return _math__uComp(aSize, a, bSize, b);
319         else
320             // do unsigned compare the other way
321             return 0 - _math__uComp(aSize, a, bSize, b);
322     }
```

B.5.2.10. _math__ModExp

This function is used to do modular exponentiation in support of RSA. The most typical uses are: *c* = *m*^*e* mod *n* (RSA encrypt) and *m* = *c*^*d* mod *n* (RSA decrypt). When doing decryption, the *e* parameter of the function will contain the private exponent *d* instead of the public exponent *e*.

If the results will not fit in the provided buffer, an error is returned (CRYPT_ERROR_UNDERFLOW). If the results is smaller than the buffer, the results is de-normalized.

This version is intended for use with RSA and requires that *m* be less than *n*.

Return Value	Meaning
CRYPT_SUCCESS	exponentiation succeeded
CRYPT_PARAMETER	number to exponentiate is larger than the modulus
CRYPT_UNDERFLOW	result will not fit into the provided buffer

```
323     LIB_EXPORT CRYPT_RESULT
324     _math__ModExp(
325         UINT32           cSize,         // IN: size of the result
326         BYTE            *c,             // OUT: results buffer
327         const UINT32     mSize,         // IN: size of number to be exponentiated
328         const BYTE      *m,             // IN: number to be exponentiated
329         const UINT32     eSize,         // IN: size of power
330         const BYTE      *e,             // IN: power
331         const UINT32     nSize,         // IN: modulus size
332         const BYTE      *n              // IN: modulu
333         )
334     {
335         CRYPT_RESULT     retVal = CRYPT_SUCCESS;
336         BN_CTX          *context;
337         BIGNUM          *bnC;
338         BIGNUM          *bnM;
339         BIGNUM          *bnE;
340         BIGNUM          *bnN;
341         INT32            i;
342
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);
351
352         // Errors for BN_CTX_get are sticky so only need to check last allocation
353         if(bnN == NULL)
```

```
354             FAIL(FATAL_ERROR_ALLOCATION);
355
356         //convert arguments
357         if (   BN_bin2bn(m, mSize, bnM) == NULL
358             || BN_bin2bn(e, eSize, bnE) == NULL
359             || BN_bin2bn(n, nSize, bnN) == NULL)
360                 FAIL(FATAL_ERROR_INTERNAL);
361
362         // Don't do exponentiation if the number being exponentiated is
363         // larger than the modulus.
364         if(BN_ucmp(bnM, bnN) >= 0)
365         {
366             retVal = CRYPT_PARAMETER;
367             goto Cleanup;
368         }
369         // Perform the exponentiation
370         if(!(BN_mod_exp(bnC, bnM, bnE, bnN, context)))
371             FAIL(FATAL_ERROR_INTERNAL);
372
373         // Convert the results
374         // Make sure that the results will fit in the provided buffer.
375         if((unsigned)BN_num_bytes(bnC) > cSize)
376         {
377             retVal = CRYPT_UNDERFLOW;
378             goto Cleanup;
379         }
380         i = cSize - BN_num_bytes(bnC);
381         BN_bn2bin(bnC, &c[i]);
382         memset(c, 0, i);
383
384     Cleanup:
385         // Free up allocated BN values
386         BN_CTX_end(context);
387         BN_CTX_free(context);
388         return retVal;
389     }
```

B.5.2.11. _math__IsPrime()

Check if an 32-bit integer is a prime.

Return Value	Meaning
TRUE	if the integer is probably a prime
FALSE	if the integer is definitely not a prime

```
390     LIB_EXPORT BOOL
391     _math__IsPrime(
392         const UINT32     prime
393         )
394     {
395         int     isPrime;
396         BIGNUM  *p;
397
398         // Assume the size variables are not overflow, which should not happen in
399         // the contexts that this function will be called.
400         if((p = BN_new()) == NULL)
401             FAIL(FATAL_ERROR_ALLOCATION);
402         if(!BN_set_word(p, prime))
403             FAIL(FATAL_ERROR_INTERNAL);
404
405         //
406         // BN_is_prime returning -1 means that it ran into an error.
```

```
407         // It should only return 0 or 1
408         //
409         if((isPrime = BN_is_prime_ex(p, BN_prime_checks, NULL, NULL)) < 0)
410             FAIL(FATAL_ERROR_INTERNAL);
411 
412         if(p != NULL)
413             BN_clear_free(p);
414         return (isPrime == 1);
415     }
```

B.6 CpriCryptPri.c

B.6.1. Introduction

This file contains the interface to the initialization, startup and shutdown functions of the crypto library.

B.6.2. Includes and Locals

```
 #include "OsslCryptoEngine.h"
static void Trap(const char *function, int line, int code);
FAIL_FUNCTION       TpmFailFunction = (FAIL_FUNCTION)&Trap;
```

B.6.3. Functions

B.6.3.1. TpmFail()

This is a shim function that is called when a failure occurs. It simply relays the call to the callback pointed to by TpmFailFunction(). It is only defined for the sake of NO_RETURN specifier that cannot be added to a function pointer with some compilers.

```
void
TpmFail(
    const char          *function,
    int                  line,
    int                  code)
{
    TpmFailFunction(function, line, code);
}
```

B.6.3.2. FAILURE_TRAP()

This function is called if the caller to _cpri__InitCryptoUnits() doesn't provide a call back address.

```
static void
Trap(
    const char      *function,
    int              line,
    int              code
    )
{
    UNREFERENCED(function);
    UNREFERENCED(line);
    UNREFERENCED(code);
    abort();
}
```

B.6.3.3. _cpri__InitCryptoUnits()

This function calls the initialization functions of the other crypto modules that are part of the crypto engine for this implementation. This function should be called as a result of _TPM_Init(). The parameter to this function is a call back function it TPM.lib that is called when the crypto engine has a failure.

```
LIB_EXPORT CRYPT_RESULT
_cpri__InitCryptoUnits(
    FAIL_FUNCTION    failFunction
    )
{
```

```
29      TpmFailFunction = failFunction;
30
31      _cpri__RngStartup();
32      _cpri__HashStartup();
33      _cpri__SymStartup();
34
35  #ifdef TPM_ALG_RSA
36      _cpri__RsaStartup();
37  #endif
38
39  #ifdef TPM_ALG_ECC
40      _cpri__EccStartup();
41  #endif
42
43      return CRYPT_SUCCESS;
44  }
```

B.6.3.4. _cpri__StopCryptoUnits()

This function calls the shutdown functions of the other crypto modules that are part of the crypto engine for this implementation.

```
45  LIB_EXPORT void
46  _cpri__StopCryptoUnits(
47      void
48      )
49  {
50      return;
51  }
```

B.6.3.5. _cpri__Startup()

This function calls the startup functions of the other crypto modules that are part of the crypto engine for this implementation. This function should be called during processing of TPM2_Startup().

```
52  LIB_EXPORT BOOL
53  _cpri__Startup(
54      void
55      )
56  {
57
58      return(   _cpri__HashStartup()
59             && _cpri__RngStartup()
60  #ifdef TPM_ALG_RSA
61             && _cpri__RsaStartup()
62  #endif // TPM_ALG_RSA
63  #ifdef TPM_ALG_ECC
64             && _cpri__EccStartup()
65  #endif // TPM_ALG_ECC
66             && _cpri__SymStartup());
67  }
```

B.7 CpriRNG.c

```
1   //#define __TPM_RNG_FOR_DEBUG__
```

B.7.1. Introduction

This file contains the interface to the OpenSSL() random number functions.

B.7.2. Includes

```
2   #include "OsslCryptoEngine.h"
3   int         s_entropyFailure;
```

B.7.3. Functions

B.7.3.1. _cpri__RngStartup()

This function is called to initialize the random number generator. It collects entropy from the platform to seed the OpenSSL() random number generator.

```
4   LIB_EXPORT BOOL
5   _cpri__RngStartup(void)
6   {
7       UINT32       entropySize;
8       BYTE         entropy[MAX_RNG_ENTROPY_SIZE];
9       INT32        returnedSize = 0;
10
11      // Initialize the entropy source
12      s_entropyFailure = FALSE;
13      _plat__GetEntropy(NULL, 0);
14
15      // Collect entropy until we have enough
16      for(entropySize  = 0;
17          entropySize < MAX_RNG_ENTROPY_SIZE && returnedSize >= 0;
18          entropySize += returnedSize)
19      {
20          returnedSize = _plat__GetEntropy(&entropy[entropySize],
21                                           MAX_RNG_ENTROPY_SIZE - entropySize);
22      }
23      // Got some entropy on the last call and did not get an error
24      if(returnedSize > 0)
25      {
26          // Seed OpenSSL with entropy
27          RAND_seed(entropy, entropySize);
28      }
29      else
30      {
31          s_entropyFailure = TRUE;
32      }
33      return s_entropyFailure == FALSE;
34  }
```

B.7.3.2. _cpri__DrbgGetPutState()

This function is used to set the state of the RNG *(direction* == PUT_STATE) or to recover the state of the RNG *(direction* == GET_STATE).

NOTE: This not currently supported on OpenSSL() version.

```
35  LIB_EXPORT CRYPT_RESULT
36  _cpri__DrbgGetPutState(
37      GET_PUT           direction,
38      int               bufferSize,
39      BYTE             *buffer
40      )
41  {
42      UNREFERENCED_PARAMETER(direction);
43      UNREFERENCED_PARAMETER(bufferSize);
44      UNREFERENCED_PARAMETER(buffer);
45
46      return CRYPT_SUCCESS;       // Function is not implemented
47  }
```

B.7.3.3. _cpri__StirRandom()

This function is called to add external entropy to the OpenSSL() random number generator.

```
48  LIB_EXPORT CRYPT_RESULT
49  _cpri__StirRandom(
50      INT32             entropySize,
51      BYTE             *entropy
52      )
53  {
54      if (entropySize >= 0)
55      {
56          RAND_add((const void *)entropy, (int) entropySize, 0.0);
57
58      }
59      return CRYPT_SUCCESS;
60  }
```

B.7.3.4. _cpri__GenerateRandom()

This function is called to get a string of random bytes from the OpenSSL() random number generator. The return value is the number of bytes placed in the buffer. If the number of bytes returned is not equal to the number of bytes requested *(randomSize*) it is indicative of a failure of the OpenSSL() random number generator and is probably fatal.

```
61  LIB_EXPORT UINT16
62  _cpri__GenerateRandom(
63      INT32             randomSize,
64      BYTE             *buffer
65      )
66  {
67      //
68      // We don't do negative sizes or ones that are too large
69      if (randomSize < 0 || randomSize > UINT16_MAX)
70          return 0;
71      // RAND_bytes uses 1 for success and we use 0
72      if(RAND_bytes(buffer, randomSize) == 1)
73          return (UINT16)randomSize;
74      else
75          return 0;
76  }
```

B.7.3.4.1. _cpri__GenerateSeededRandom()

This funciton is used to generate a pseudo-random number from some seed values This funciton returns the same result each time it is called with the same parameters

```
77  LIB_EXPORT UINT16
78  _cpri__GenerateSeededRandom(
79      INT32            randomSize,     // IN: the size of the request
80      BYTE            *random,         // OUT: receives the data
81      TPM_ALG_ID       hashAlg,        // IN: used by KDF version but not here
82      TPM2B           *seed,           // IN: the seed value
83      const char      *label,          // IN: a label string (optional)
84      TPM2B           *partyU,         // IN: other data (oprtional)
85      TPM2B           *partyV          // IN: still more (optional)
86      )
87  {
88
89      return (_cpri__KDFa(hashAlg, seed, label, partyU, partyV,
90                          randomSize * 8, random, NULL, FALSE));
91  }
92  #endif  //%
```

B.8 CpriHash.c

B.8.1. Description

This file contains implementation of cryptographic functions for hashing.

B.8.2. Includes, Defines, and Types

```
#include    "OsslCryptoEngine.h"
#include    "CpriHashData.c"
#define OSSL_HASH_STATE_DATA_SIZE    (MAX_HASH_STATE_SIZE - 8)
typedef struct {
    union   {
        EVP_MD_CTX  context;
        BYTE        data[OSSL_HASH_STATE_DATA_SIZE];
    } u;
    INT16           copySize;
} OSSL_HASH_STATE;
```

Temporary aliasing of SM3 to SHA256 until SM3 is available

```
#define EVP_sm3_256 EVP_sha256
```

B.8.3. Static Functions

B.8.3.1. GetHashServer()

This function returns the address of the hash server function

```
static EVP_MD *
GetHashServer(
    TPM_ALG_ID   hashAlg
)
{
    switch (hashAlg)
    {
#ifdef TPM_ALG_SHA1
    case TPM_ALG_SHA1:
        return (EVP_MD *)EVP_sha1();
        break;
#endif
#ifdef TPM_ALG_SHA256
    case TPM_ALG_SHA256:
        return (EVP_MD *)EVP_sha256();
        break;
#endif
#ifdef TPM_ALG_SHA384
    case TPM_ALG_SHA384:
        return (EVP_MD *)EVP_sha384();
        break;
#endif
#ifdef TPM_ALG_SHA512
    case TPM_ALG_SHA512:
        return (EVP_MD *)EVP_sha512();
        break;
#endif
#ifdef TPM_ALG_SM3_256
    case TPM_ALG_SM3_256:
        return (EVP_MD *)EVP_sm3_256();
        break;
```

```
43  #endif
44      case TPM_ALG_NULL:
45          return NULL;
46      default:
47          FAIL(FATAL_ERROR_INTERNAL);
48      }
49  }
```

B.8.3.2. MarshalHashState()

This function copies an OpenSSL() hash context into a caller provided buffer.

Return Value	Meaning
> 0	the number of bytes of buf used.

```
50  static UINT16
51  MarshalHashState(
52      EVP_MD_CTX      *ctxt,          // IN: Context to marshal
53      BYTE            *buf            // OUT: The buffer that will receive the
54                                      //      context. This buffer is at least
55                                      //      MAX_HASH_STATE_SIZE byte
56      )
57  {
58      // make sure everything will fit
59      pAssert(ctxt->digest->ctx_size <= OSSL_HASH_STATE_DATA_SIZE);
60
61      // Copy the context data
62      memcpy(buf, (void*) ctxt->md_data, ctxt->digest->ctx_size);
63
64      return (UINT16)ctxt->digest->ctx_size;
65  }
```

B.8.3.3. GetHashState()

This function will unmarshal a caller provided buffer into an OpenSSL() hash context. The function returns the number of bytes copied (which may be zero).

```
66  static UINT16
67  GetHashState(
68      EVP_MD_CTX      *ctxt,          // OUT: The context structure to receive the
69                                      //      result of unmarshaling.
70      TPM_ALG_ID       algType,       // IN: The hash algorithm selector
71      BYTE            *buf            // IN: Buffer containing marshaled hash data
72      )
73  {
74      EVP_MD          *evpmdAlgorithm = NULL;
75
76      pAssert(ctxt != NULL);
77
78      EVP_MD_CTX_init(ctxt);
79
80      evpmdAlgorithm = GetHashServer(algType);
81      if(evpmdAlgorithm == NULL)
82          return 0;
83
84      // This also allocates the ctxt->md_data
85      if((EVP_DigestInit_ex(ctxt, evpmdAlgorithm, NULL)) != 1)
86          FAIL(FATAL_ERROR_INTERNAL);
87
88      pAssert(ctxt->digest->ctx_size < sizeof(ALIGNED_HASH_STATE));
89      memcpy(ctxt->md_data, buf, ctxt->digest->ctx_size);
```

```
90      return (UINT16)ctxt->digest->ctx_size;
91  }
```

B.8.3.4. GetHashInfoPointer()

This function returns a pointer to the hash info for the algorithm. If the algorithm is not supported, function returns a pointer to the data block associated with TPM_ALG_NULL.

```
92  static const HASH_INFO *
93  GetHashInfoPointer(
94      TPM_ALG_ID       hashAlg
95      )
96  {
97      UINT32 i, tableSize;
98
99      // Get the table size of g_hashData
100     tableSize = sizeof(g_hashData) / sizeof(g_hashData[0]);
101
102     for(i = 0; i < tableSize - 1; i++)
103     {
104         if(g_hashData[i].alg == hashAlg)
105             return &g_hashData[i];
106     }
107     return &g_hashData[tableSize-1];
108 }
```

B.8.4. Hash Functions

B.8.4.1. _cpri__HashStartup()

Function that is called to initialize the hash service. In this implementation, this function does nothing but it is called by the CryptUtilStartup() function and must be present.

```
109 LIB_EXPORT BOOL
110 _cpri__HashStartup(
111     void
112     )
113 {
114     // On startup, make sure that the structure sizes are compatible. It would
115     // be nice if this could be done at compile time but I couldn't figure it out.
116     CPRI_HASH_STATE *cpriState = NULL;
117 //    NUMBYTES         evpCtxSize = sizeof(EVP_MD_CTX);
118     NUMBYTES        cpriStateSize = sizeof(cpriState->state);
119 //    OSSL_HASH_STATE *osslState;
120     NUMBYTES        osslStateSize = sizeof(OSSL_HASH_STATE);
121 //    int              dataSize = sizeof(osslState->u.data);
122     pAssert(cpriStateSize >= osslStateSize);
123
124     return TRUE;
125 }
```

B.8.4.2. _cpri__GetHashAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and and *index* of 2 will return the last. All other index values will return TPM_ALG_NULL.

Return Value	Meaning
TPM_ALG_xxx()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
126 LIB_EXPORT TPM_ALG_ID
127 _cpri__GetHashAlgByIndex(
128     UINT32          index           // IN: the index
129     )
130 {
131     if(index >= HASH_COUNT)
132         return TPM_ALG_NULL;
133     return g_hashData[index].alg;
134 }
```

B.8.4.3. _cpri__GetHashBlockSize()

Returns the size of the block used for the hash

Return Value	Meaning
< 0	the algorithm is not a supported hash
>=	the digest size (0 for TPM_ALG_NULL)

```
135 LIB_EXPORT UINT16
136 _cpri__GetHashBlockSize(
137     TPM_ALG_ID      hashAlg         // IN: hash algorithm to look up
138     )
139 {
140     return GetHashInfoPointer(hashAlg)->blockSize;
141 }
```

B.8.4.4. _cpri__GetHashDER

This function returns a pointer to the DER string for the algorithm and indicates its size.

```
142 LIB_EXPORT UINT16
143 _cpri__GetHashDER(
144     TPM_ALG_ID      hashAlg,        // IN: the algorithm to look up
145     const BYTE      **p
146     )
147 {
148     const HASH_INFO       *q;
149     q = GetHashInfoPointer(hashAlg);
150     *p = &q->der[0];
151     return q->derSize;
152 }
```

B.8.4.5. _cpri__GetDigestSize()

Gets the digest size of the algorithm. The algorithm is required to be supported.

Return Value	Meaning
=0	the digest size for TPM_ALG_NULL
>0	the digest size of a hash algorithm

```
153 LIB_EXPORT UINT16
```

```
154 _cpri__GetDigestSize(
155     TPM_ALG_ID       hashAlg        // IN: hash algorithm to look up
156     )
157 {
158     return GetHashInfoPointer(hashAlg)->digestSize;
159 }
```

B.8.4.6. _cpri__GetContextAlg()

This function returns the algorithm associated with a hash context

```
160 LIB_EXPORT TPM_ALG_ID
161 _cpri__GetContextAlg(
162     CPRI_HASH_STATE     *hashState      // IN: the hash context
163     )
164 {
165     return hashState->hashAlg;
166 }
```

B.8.4.7. _cpri__CopyHashState

This function is used to **clone** a CPRI_HASH_STATE. The return value is the size of the state.

```
167 LIB_EXPORT UINT16
168 _cpri__CopyHashState (
169     CPRI_HASH_STATE     *out,           // OUT: destination of the state
170     CPRI_HASH_STATE     *in             // IN: source of the state
171     )
172 {
173     OSSL_HASH_STATE    *i = (OSSL_HASH_STATE *)&in->state;
174     OSSL_HASH_STATE    *o = (OSSL_HASH_STATE *)&out->state;
175     pAssert(sizeof(i) <= sizeof(in->state));
176
177     EVP_MD_CTX_init(&o->u.context);
178     EVP_MD_CTX_copy_ex(&o->u.context, &i->u.context);
179     o->copySize = i->copySize;
180     out->hashAlg = in->hashAlg;
181     return sizeof(CPRI_HASH_STATE);
182 }
```

B.8.4.8. _cpri__StartHash()

Functions starts a hash stack Start a hash stack and returns the digest size. As a side effect, the value of *stateSize* in *hashState* is updated to indicate the number of bytes of state that were saved. This function calls GetHashServer() and that function will put the TPM into failure mode if the hash algorithm is not supported.

Return Value	Meaning
0	hash is TPM_ALG_NULL
>0	digest size

```
183 LIB_EXPORT UINT16
184 _cpri__StartHash(
185     TPM_ALG_ID           hashAlg,       // IN: hash algorithm
186     BOOL                 sequence,      // IN: TRUE if the state should be saved
187     CPRI_HASH_STATE     *hashState      // OUT: the state of hash stack.
188     )
189 {
190     EVP_MD_CTX       localState;
```

```
191         OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
192         BYTE            *stateData = state->u.data;
193         EVP_MD_CTX      *context;
194         EVP_MD          *evpmdAlgorithm = NULL;
195         UINT16           retVal = 0;
196 
197         if(sequence)
198             context = &localState;
199         else
200             context = &state->u.context;
201 
202         hashState->hashAlg = hashAlg;
203 
204         EVP_MD_CTX_init(context);
205         evpmdAlgorithm = GetHashServer(hashAlg);
206         if(evpmdAlgorithm == NULL)
207             goto Cleanup;
208 
209         if(EVP_DigestInit_ex(context, evpmdAlgorithm, NULL) != 1)
210             FAIL(FATAL_ERROR_INTERNAL);
211         retVal = (CRYPT_RESULT)EVP_MD_CTX_size(context);
212 
213     Cleanup:
214         if(retVal > 0)
215         {
216             if (sequence)
217             {
218                 if((state->copySize = MarshalHashState(context, stateData)) == 0)
219                 {
220                     // If MarshalHashState returns a negative number, it is an error
221                     // code and not a hash size so copy the error code to be the return
222                     // from this function and set the actual stateSize to zero.
223                     retVal = state->copySize;
224                     state->copySize = 0;
225                 }
226                 // Do the cleanup
227                 EVP_MD_CTX_cleanup(context);
228             }
229             else
230                 state->copySize = -1;
231         }
232         else
233             state->copySize = 0;
234         return retVal;
235     }
```

B.8.4.9. _cpri__UpdateHash()

Add data to a hash or HMAC stack.

```
236     LIB_EXPORT void
237     _cpri__UpdateHash(
238         CPRI_HASH_STATE     *hashState,     // IN: the hash context information
239         UINT32               dataSize,      // IN: the size of data to be added to the
240                                             //     digest
241         BYTE                *data           // IN: data to be hashed
242         )
243     {
244         EVP_MD_CTX       localContext;
245         OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
246         BYTE            *stateData = state->u.data;
247         EVP_MD_CTX      *context;
248         CRYPT_RESULT     retVal = CRYPT_SUCCESS;
249 
```

```
250         // If there is no context, return
251         if(state->copySize == 0)
252             return;
253         if(state->copySize > 0)
254         {
255             context = &localContext;
256             if((retVal = GetHashState(context, hashState->hashAlg, stateData)) <= 0)
257                 return;
258         }
259         else
260             context = &state->u.context;
261
262         if(EVP_DigestUpdate(context, data, dataSize) != 1)
263             FAIL(FATAL_ERROR_INTERNAL);
264         else if(   state->copySize > 0
265                    && (retVal= MarshalHashState(context, stateData)) >= 0)
266         {
267             // retVal is the size of the marshaled data. Make sure that it is consistent
268             // by ensuring that we didn't get more than allowed
269             if(retVal < state->copySize)
270                 FAIL(FATAL_ERROR_INTERNAL);
271             else
272                 EVP_MD_CTX_cleanup(context);
273         }
274         return;
275     }
```

B.8.4.10. _cpri__CompleteHash()

Complete a hash or HMAC computation. This function will place the smaller of *digestSize* or the size of the digest in *dOut*. The number of bytes in the placed in the buffer is returned. If there is a failure, the returned value is <= 0.

Return Value	Meaning
0	no data returned
> 0	the number of bytes in the digest

```
276     LIB_EXPORT UINT16
277     _cpri__CompleteHash(
278         CPRI_HASH_STATE     *hashState,     // IN: the state of hash stack
279         UINT32               dOutSize,      // IN: size of digest buffer
280         BYTE                *dOut           // OUT: hash digest
281         )
282     {
283         EVP_MD_CTX       localState;
284         OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
285         BYTE            *stateData = state->u.data;
286         EVP_MD_CTX      *context;
287         UINT16           retVal;
288         int              hLen;
289         BYTE             temp[MAX_DIGEST_SIZE];
290         BYTE            *rBuffer = dOut;
291
292         if(state->copySize == 0)
293             return 0;
294         if(state->copySize > 0)
295         {
296             context = &localState;
297             if((retVal = GetHashState(context, hashState->hashAlg, stateData)) <= 0)
298                 goto Cleanup;
299         }
300         else
```

```
301             context = &state->u.context;
302 
303         hLen = EVP_MD_CTX_size(context);
304         if((unsigned)hLen > dOutSize)
305             rBuffer = temp;
306         if(EVP_DigestFinal_ex(context, rBuffer, NULL) == 1)
307         {
308             if(rBuffer != dOut)
309             {
310                 if(dOut != NULL)
311                 {
312                     memcpy(dOut, temp, dOutSize);
313                 }
314                 retVal = (UINT16)dOutSize;
315             }
316             else
317             {
318                 retVal = (UINT16)hLen;
319             }
320             state->copySize = 0;
321         }
322         else
323         {
324             retVal = 0; // Indicate that no data is returned
325         }
326     Cleanup:
327         EVP_MD_CTX_cleanup(context);
328         return retVal;
329     }
```

B.8.4.11. _cpri__ImportExportHashState()

This function is used to import or export the hash state. This function would be called to export state when a sequence object was being prepared for export

```
330     LIB_EXPORT void
331     _cpri__ImportExportHashState(
332         CPRI_HASH_STATE     *osslFmt,       // IN/OUT: the hash state formated for use
333                                             //     by openSSL
334         EXPORT_HASH_STATE   *externalFmt,   // IN/OUT: the exported hash state
335         IMPORT_EXPORT        direction      //
336         )
337     {
338         UNREFERENCED_PARAMETER(direction);
339         UNREFERENCED_PARAMETER(externalFmt);
340         UNREFERENCED_PARAMETER(osslFmt);
341         return;
342 
343     #if 0
344         if(direction == IMPORT_STATE)
345         {
346             // don't have the import export functions yet so just copy
347             _cpri__CopyHashState(osslFmt, (CPRI_HASH_STATE *)externalFmt);
348         }
349         else
350         {
351             _cpri__CopyHashState((CPRI_HASH_STATE *)externalFmt, osslFmt);
352         }
353     #endif
354     }
```

B.8.4.12. _cpri__HashBlock()

Start a hash, hash a single block, update *digest* and return the size of the results.

The **digestSize** parameter can be smaller than the digest. If so, only the more significant bytes are returned.

Return Value	Meaning
>= 0	number of bytes in *digest* (may be zero)

```
355  LIB_EXPORT UINT16
356  _cpri__HashBlock(
357      TPM_ALG_ID       hashAlg,       // IN: The hash algorithm
358      UINT32           dataSize,      // IN: size of buffer to hash
359      BYTE            *data,          // IN: the buffer to hash
360      UINT32           digestSize,    // IN: size of the digest buffer
361      BYTE            *digest         // OUT: hash digest
362      )
363  {
364      EVP_MD_CTX       hashContext;
365      EVP_MD          *hashServer = NULL;
366      UINT16           retVal = 0;
367      BYTE             b[MAX_DIGEST_SIZE]; // temp buffer in case digestSize not
368      // a full digest
369      unsigned int     dSize = _cpri__GetDigestSize(hashAlg);
370
371      // If there is no digest to compute return
372      if(dSize == 0)
373          return 0;
374
375      // After the call to EVP_MD_CTX_init(), will need to call EVP_MD_CTX_cleanup()
376      EVP_MD_CTX_init(&hashContext);     // Initialize the local hash context
377      hashServer = GetHashServer(hashAlg); // Find the hash server
378
379      // It is an error if the digest size is non-zero but there is no server
380      if(   (hashServer == NULL)
381         || (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
382         || (EVP_DigestUpdate(&hashContext, data, dataSize) != 1))
383          FAIL(FATAL_ERROR_INTERNAL);
384      else
385      {
386          // If the size of the digest produced (dSize) is larger than the available
387          // buffer (digestSize), then put the digest in a temp buffer and only copy
388          // the most significant part into the available buffer.
389          if(dSize > digestSize)
390          {
391              if(EVP_DigestFinal_ex(&hashContext, b, &dSize) != 1)
392                  FAIL(FATAL_ERROR_INTERNAL);
393              memcpy(digest, b, digestSize);
394              retVal = (UINT16)digestSize;
395          }
396          else
397          {
398              if((EVP_DigestFinal_ex(&hashContext, digest, &dSize)) != 1)
399                  FAIL(FATAL_ERROR_INTERNAL);
400              retVal = (UINT16) dSize;
401          }
402      }
403      EVP_MD_CTX_cleanup(&hashContext);
404      return retVal;
405  }
```


B.8.5. HMAC Functions

B.8.5.1. _cpri__StartHMAC

This function is used to start an HMAC using a temp hash context. The function does the initialization of the hash with the HMAC key XOR *iPad* and updates the HMAC key XOR *oPad*.

The function returns the number of bytes in a digest produced by *hashAlg*.

Return Value	Meaning
>= 0	number of bytes in digest produced by *hashAlg* (may be zero)

```
406  LIB_EXPORT UINT16
407  _cpri__StartHMAC(
408      TPM_ALG_ID            hashAlg,        // IN: the algorithm to use
409      BOOL                  sequence,       // IN: indicates if the state should be
410                                            //     saved
411      CPRI_HASH_STATE      *state,          // IN/OUT: the state buffer
412      UINT16                keySize,        // IN: the size of the HMAC key
413      BYTE                 *key,            // IN: the HMAC key
414      TPM2B                *oPadKey         // OUT: the key prepared for the oPad round
415      )
416  {
417      CPRI_HASH_STATE  localState;
418      UINT16           blockSize = _cpri__GetHashBlockSize(hashAlg);
419      UINT16           digestSize;
420      BYTE            *pb;        // temp pointer
421      UINT32           i;
422
423      // If the key size is larger than the block size, then the hash of the key
424      // is used as the key
425      if(keySize > blockSize)
426      {
427          // large key so digest
428          if((digestSize = _cpri__StartHash(hashAlg, FALSE, &localState)) == 0)
429              return 0;
430          _cpri__UpdateHash(&localState, keySize, key);
431          _cpri__CompleteHash(&localState, digestSize, oPadKey->buffer);
432          oPadKey->size = digestSize;
433      }
434      else
435      {
436          // key size is ok
437          memcpy(oPadKey->buffer, key, keySize);
438          oPadKey->size = keySize;
439      }
440      // XOR the key with iPad (0x36)
441      pb = oPadKey->buffer;
442      for(i = oPadKey->size; i > 0; i--)
443          *pb++ ^= 0x36;
444
445      // if the keySize is smaller than a block, fill the rest with 0x36
446      for(i = blockSize - oPadKey->size; i >  0; i--)
447          *pb++ = 0x36;
448
449      // Increase the oPadSize to a full block
450      oPadKey->size = blockSize;
451
452      // Start a new hash with the HMAC key
453      // This will go in the caller's state structure and may be a sequence or not
454
455      if((digestSize = _cpri__StartHash(hashAlg, sequence, state)) > 0)
456      {
```

```
457
458         _cpri__UpdateHash(state, oPadKey->size, oPadKey->buffer);
459
460         // XOR the key block with 0x5c ^ 0x36
461         for(pb = oPadKey->buffer, i = blockSize; i > 0; i--)
462             *pb++ ^= (0x5c ^ 0x36);
463     }
464
465     return digestSize;
466 }
```

B.8.5.2. _cpri_CompleteHMAC()

This function is called to complete an HMAC. It will finish the current digest, and start a new digest. It will then add the *oPadKey* and the completed digest and return the results in *dOut*. It will not return more than *dOutSize* bytes.

Return Value	Meaning
>= 0	number of bytes in *dOut* (may be zero)

```
467 LIB_EXPORT UINT16
468 _cpri__CompleteHMAC(
469     CPRI_HASH_STATE     *hashState,     // IN: the state of hash stack
470     TPM2B               *oPadKey,       // IN: the HMAC key in oPad format
471     UINT32               dOutSize,      // IN: size of digest buffer
472     BYTE                *dOut           // OUT: hash digest
473     )
474 {
475     BYTE             digest[MAX_DIGEST_SIZE];
476     CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
477     CPRI_HASH_STATE  localState;
478     UINT16           digestSize = _cpri__GetDigestSize(state->hashAlg);
479
480     _cpri__CompleteHash(hashState, digestSize, digest);
481
482     // Using the local hash state, do a hash with the oPad
483     if(_cpri__StartHash(state->hashAlg, FALSE, &localState) != digestSize)
484         return 0;
485
486     _cpri__UpdateHash(&localState, oPadKey->size, oPadKey->buffer);
487     _cpri__UpdateHash(&localState, digestSize, digest);
488     return _cpri__CompleteHash(&localState, dOutSize, dOut);
489 }
```

B.8.6. Mask and Key Generation Functions

B.8.6.1. _crypi_MGF1()

This function performs MGF1 using the selected hash. MGF1 is T(n) = T(n-1) || H(seed || counter). This function returns the length of the mask produced which could be zero if the digest algorithm is not supported

Return Value	Meaning
0	hash algorithm not supported
> 0	should be the same as *mSize*

```
490 LIB_EXPORT CRYPT_RESULT
491 _cpri__MGF1(
```

```
492         UINT32           mSize,         // IN: length of the mask to be produced
493         BYTE            *mask,          // OUT: buffer to receive the mask
494         TPM_ALG_ID       hashAlg,       // IN: hash to use
495         UINT32           sSize,         // IN: size of the seed
496         BYTE            *seed           // IN: seed size
497         )
498     {
499         EVP_MD_CTX           hashContext;
500         EVP_MD              *hashServer = NULL;
501         CRYPT_RESULT         retVal = 0;
502         BYTE                 b[MAX_DIGEST_SIZE]; // temp buffer in case mask is not an
503         // even multiple of a full digest
504         CRYPT_RESULT         dSize = _cpri__GetDigestSize(hashAlg);
505         unsigned int         digestSize = (UINT32)dSize;
506         UINT32               remaining;
507         UINT32               counter;
508         BYTE                 swappedCounter[4];
509
510         // Parameter check
511         if(mSize > (1024*16)) // Semi-arbitrary maximum
512             FAIL(FATAL_ERROR_INTERNAL);
513
514         // If there is no digest to compute return
515         if(dSize <= 0)
516             return 0;
517
518         EVP_MD_CTX_init(&hashContext);     // Initialize the local hash context
519         hashServer = GetHashServer(hashAlg); // Find the hash server
520         if(hashServer == NULL)
521             // If there is no server, then there is no digest
522             return 0;
523
524         for(counter = 0, remaining = mSize; remaining > 0; counter++)
525         {
526             // Because the system may be either Endian...
527             UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
528
529             // Start the hash and include the seed and counter
530             if(    (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
531                 || (EVP_DigestUpdate(&hashContext, seed, sSize) != 1)
532                 || (EVP_DigestUpdate(&hashContext, swappedCounter, 4) != 1)
533               )
534                 FAIL(FATAL_ERROR_INTERNAL);
535
536             // Handling the completion depends on how much space remains in the mask
537             // buffer. If it can hold the entire digest, put it there. If not
538             // put the digest in a temp buffer and only copy the amount that
539             // will fit into the mask buffer.
540             if(remaining < (unsigned)dSize)
541             {
542                 if(EVP_DigestFinal_ex(&hashContext, b, &digestSize) != 1)
543                     FAIL(FATAL_ERROR_INTERNAL);
544                 memcpy(mask, b, remaining);
545                 break;
546             }
547             else
548             {
549                 if(EVP_DigestFinal_ex(&hashContext, mask, &digestSize) != 1)
550                     FAIL(FATAL_ERROR_INTERNAL);
551                 remaining -= dSize;
552                 mask = &mask[dSize];
553             }
554             retVal = (CRYPT_RESULT)mSize;
555         }
556
557         EVP_MD_CTX_cleanup(&hashContext);
```

```
558         return retVal;
559     }
```

B.8.6.2. _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.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
560     LIB_EXPORT UINT16
561     _cpri__KDFa(
562         TPM_ALG_ID       hashAlg,        // IN: hash algorithm used in HMAC
563         TPM2B           *key,            // IN: HMAC key
564         const char      *label,          // IN: a 0-byte terminated label used in KDF
565         TPM2B           *contextU,       // IN: context U
566         TPM2B           *contextV,       // IN: context V
567         UINT32           sizeInBits,     // IN: size of generated key in bit
568         BYTE            *keyStream,      // OUT: key buffer
569         UINT32          *counterInOut,   // IN/OUT: caller may provide the iteration
570                                          //     counter for incremental operations to
571                                          //     avoid large intermediate buffers.
572         BOOL             once            // IN: TRUE if only one iteration is performed
573                                          //     FALSE if iteration count determined by
574                                          //     "sizeInBits"
575         )
576     {
577         UINT32                   counter = 0;     // counter value
578         INT32                    lLen = 0;        // length of the label
579         INT16                    hLen;            // length of the hash
580         INT16                    bytes;           // number of bytes to produce
581         BYTE                    *stream = keyStream;
582         BYTE                     marshaledUint32[4];
583         CPRI_HASH_STATE          hashState;
584         TPM2B_MAX_HASH_BLOCK     hmacKey;
585
586         pAssert(key != NULL && keyStream != NULL);
587         pAssert(once == FALSE || (sizeInBits & 7) == 0);
588
589         if(counterInOut != NULL)
590             counter = *counterInOut;
591
592         // Prepare label buffer.  Calculate its size and keep the last 0 byte
593         if(label != NULL)
594             for(lLen = 0; label[lLen++] != 0; );
595
596         // Get the hash size.  If it is less than or 0, either the
597         // algorithm is not supported or the hash is TPM_ALG_NULL
```

```
598         // In either case the digest size is zero.  This is the only return
599         // other than the one at the end. All other exits from this function
600         // are fatal errors. After we check that the algorithm is supported
601         // anything else that goes wrong is an implementation flaw.
602         if((hLen = (INT16) _cpri__GetDigestSize(hashAlg)) == 0)
603             return 0;
604 
605         // If the size of the request is larger than the numbers will handle,
606         // it is a fatal error.
607         pAssert(((sizeInBits + 7)/ 8) <= INT16_MAX);
608 
609         bytes = once ? hLen : (INT16)((sizeInBits + 7) / 8);
610 
611         // Generate required bytes
612         for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
613         {
614             if(bytes < hLen)
615                 hLen = bytes;
616 
617             counter++;
618             // Start HMAC
619             if(_cpri__StartHMAC(hashAlg,
620                                 FALSE,
621                                 &hashState,
622                                 key->size,
623                                 &key->buffer[0],
624                                 &hmacKey.b)         <= 0)
625                 FAIL(FATAL_ERROR_INTERNAL);
626 
627             // Adding counter
628             UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
629             _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
630 
631             // Adding label
632             if(label != NULL)
633                 _cpri__UpdateHash(&hashState,  lLen, (BYTE *)label);
634 
635             // Adding contextU
636             if(contextU != NULL)
637                 _cpri__UpdateHash(&hashState, contextU->size, contextU->buffer);
638 
639             // Adding contextV
640             if(contextV != NULL)
641                 _cpri__UpdateHash(&hashState, contextV->size, contextV->buffer);
642 
643             // Adding size in bits
644             UINT32_TO_BYTE_ARRAY(sizeInBits, marshaledUint32);
645             _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
646 
647             // Compute HMAC. At the start of each iteration, hLen is set
648             // to the smaller of hLen and bytes. This causes bytes to decrement
649             // exactly to zero to complete the loop
650             _cpri__CompleteHMAC(&hashState, &hmacKey.b, hLen, stream);
651         }
652 
653         // Mask off bits if the required bits is not a multiple of byte size
654         if((sizeInBits % 8) != 0)
655             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
656         if(counterInOut != NULL)
657             *counterInOut = counter;
658         return (CRYPT_RESULT)((sizeInBits + 7)/8);
659     }
```

B.8.6.3. _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.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
660 LIB_EXPORT UINT16
661 _cpri__KDFe(
662     TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
663     TPM2B           *Z,             // IN: Z
664     const char      *label,         // IN: a 0 terminated label using in KDF
665     TPM2B           *partyUInfo,    // IN: PartyUInfo
666     TPM2B           *partyVInfo,    // IN: PartyVInfo
667     UINT32           sizeInBits,    // IN: size of generated key in bit
668     BYTE            *keyStream      // OUT: key buffer
669     )
670 {
671     UINT32       counter = 0;       // counter value
672     UINT32       lSize = 0;
673     BYTE        *stream = keyStream;
674     CPRI_HASH_STATE         hashState;
675     INT16        hLen = (INT16) _cpri__GetDigestSize(hashAlg);
676     INT16        bytes;             // number of bytes to generate
677     BYTE         marshaledUint32[4];
678
679     pAssert(   keyStream != NULL
680                && Z != NULL
681                && ((sizeInBits + 7) / 8) < INT16_MAX);
682
683     if(hLen == 0)
684         return 0;
685
686     bytes = (INT16)((sizeInBits + 7) / 8);
687
688     // Prepare label buffer.  Calculate its size and keep the last 0 byte
689     if(label != NULL)
690         for(lSize = 0; label[lSize++] != 0;);
691
692     // Generate required bytes
693     //The inner loop of that KDF uses:
694     //  Hashi := H(counter | Z | OtherInfo) (5)
695     // Where:
696     //  Hashi   the hash generated on the i-th iteration of the loop.
697     //  H()     an approved hash function
698     //  counter a 32-bit counter that is initialized to 1 and incremented
699     //          on each iteration
700     //  Z       the X coordinate of the product of a public ECC key and a
701     //          different private ECC key.
702     //  OtherInfo   a collection of qualifying data for the KDF defined below.
703     //  In this specification, OtherInfo will be constructed by:
704     //      OtherInfo := Use | PartyUInfo  | PartyVInfo
705     for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
706     {
707         if(bytes < hLen)
708             hLen = bytes;
```

```
709
710             counter++;
711             // Start hash
712             if(_cpri__StartHash(hashAlg, FALSE,  &hashState) == 0)
713                 return 0;
714
715             // Add counter
716             UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
717             _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
718
719             // Add Z
720             if(Z != NULL)
721                 _cpri__UpdateHash(&hashState, Z->size, Z->buffer);
722
723             // Add label
724             if(label != NULL)
725                 _cpri__UpdateHash(&hashState, lSize, (BYTE *)label);
726             else
727
728                 // The SP800-108 specification requires a zero between the label
729                 // and the context.
730                 _cpri__UpdateHash(&hashState, 1, (BYTE *)"");
731
732             // Add PartyUInfo
733             if(partyUInfo != NULL)
734                 _cpri__UpdateHash(&hashState, partyUInfo->size, partyUInfo->buffer);
735
736             // Add PartyVInfo
737             if(partyVInfo != NULL)
738                 _cpri__UpdateHash(&hashState, partyVInfo->size, partyVInfo->buffer);
739
740             // Compute Hash. hLen was changed to be the smaller of bytes or hLen
741             // at the start of each iteration.
742             _cpri__CompleteHash(&hashState, hLen, stream);
743         }
744
745         // Mask off bits if the required bits is not a multiple of byte size
746         if((sizeInBits % 8) != 0)
747             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
748
749         return (CRYPT_RESULT)((sizeInBits + 7) / 8);
750
751     }
```

B.9 CpriHashData.c

This file should be included by the library hash module.

```
1       const HASH_INFO   g_hashData[HASH_COUNT + 1] = {
2   #ifdef TPM_ALG_SHA1
3       {TPM_ALG_SHA1,    SHA1_DIGEST_SIZE,   SHA1_BLOCK_SIZE,
4       SHA1_DER_SIZE,   SHA1_DER},
5   #endif
6   #ifdef TPM_ALG_SHA256
7       {TPM_ALG_SHA256,    SHA256_DIGEST_SIZE,   SHA256_BLOCK_SIZE,
8       SHA256_DER_SIZE,   SHA256_DER},
9   #endif
10  #ifdef TPM_ALG_SHA384
11      {TPM_ALG_SHA384,    SHA384_DIGEST_SIZE,   SHA384_BLOCK_SIZE,
12      SHA384_DER_SIZE,   SHA384_DER},
13  #endif
14  #ifdef TPM_ALG_SM3_256
15      {TPM_ALG_SM3_256,    SM3_256_DIGEST_SIZE,   SM3_256_BLOCK_SIZE,
16      SM3_256_DER_SIZE,   SM3_256_DER},
17  #endif
18  #ifdef TPM_ALG_SHA512
19      {TPM_ALG_SHA512,    SHA512_DIGEST_SIZE,   SHA512_BLOCK_SIZE,
20      SHA512_DER_SIZE,   SHA512_DER},
21  #endif
22          {TPM_ALG_NULL,0,0,0,{0}}
23      };
```


B.10 CpriMisc.c

B.10.1. Includes

```
#include "OsslCryptoEngine.h"
```

B.10.2. Functions

B.10.2.1. BnTo2B()

This function is used to convert a BigNum() to a byte array of the specified size. If the number is too large to fit, then 0 is returned. Otherwise, the number is converted into the low-order bytes of the provided array and the upper bytes are set to zero.

Return Value	Meaning
0	failure (probably fatal)
1	conversion successful

```
BOOL
BnTo2B(
    TPM2B           *outVal,        // OUT: place for the result
    BIGNUM          *inVal,         // IN: number to convert
    UINT16           size           // IN: size of the output.
    )
{
    BYTE    *pb = outVal->buffer;

    outVal->size = size;

    size = size - (((UINT16) BN_num_bits(inVal) + 7) / 8);
    if(size < 0)
        return FALSE;
    for(;size > 0; size--)
        *pb++ = 0;
    BN_bn2bin(inVal, pb);
    return TRUE;
}
```

B.10.2.2. Copy2B()

This function copies a TPM2B structure. The compiler can't generate a copy of a TPM2B generic structure because the actual size is not known. This function performs the copy on any TPM2B pair. The size of the destination should have been checked before this call to make sure that it will hold the TPM2B being copied.

This replicates the functionality in the MemoryLib.c.

```
void
Copy2B(
    TPM2B           *out,           // OUT: The TPM2B to receive the copy
    TPM2B           *in             // IN: the TPM2B to copy
    )
{
    BYTE        *pIn = in->buffer;
    BYTE        *pOut = out->buffer;
    int          count;
    out->size = in->size;
    for(count = in->size; count > 0; count--)
```

```
32          *pOut++ = *pIn++;
33      return;
34  }
```

B.10.2.3. BnFrom2B()

This function creates a BIGNUM from a TPM2B and fails if the conversion fails.

```
35  BIGNUM *
36  BnFrom2B(
37      BIGNUM          *out,           // OUT: The BIGNUM
38      const TPM2B     *in             // IN: the TPM2B to copy
39      )
40  {
41      if(BN_bin2bn(in->buffer, in->size, out) == NULL)
42          FAIL(FATAL_ERROR_INTERNAL);
43      return out;
44  }
```

B.11 CpriSym.c

B.11.1. Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These function only use the single block encryption and decryption functions of OpesnSSL().

Currently, this module only supports AES encryption. The SM4 code actually calls an AES routine

B.11.2. Includes, Defines, and Typedefs

```
1   #include     "OsslCryptoEngine.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.11.3. Utility Functions

B.11.3.1. _cpri_SymStartup()

```
7   LIB_EXPORT BOOL
8   _cpri__SymStartup(
9       void
10  )
11  {
12      return TRUE;
13  }
```

B.11.3.2. _cpri__GetSymmetricBlockSize()

This function returns the block size of the algorithm.

Return Value	Meaning
<= 0	cipher not supported
> 0	the cipher block size in bytes

```
14  LIB_EXPORT 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
```

```
            // keySizeInBits for the compiler
            return  16;
        else
            return 0;
        break;

    default:
        return 0;
    }
}
```

B.11.4. AES Encryption

B.11.4.1. _cpri__AESEncryptCBC()

This function performs AES encryption in CBC chain mode. The input *dIn* buffer is encrypted into *dOut*.

The input iv buffer is required to have a size equal to the block size (16 bytes). The *dInSize* is required to be a multiple of the block size.

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	*dInSize* is not a multiple of the block size

```
LIB_EXPORT CRYPT_RESULT
_cpri__AESEncryptCBC(
    BYTE            *dOut,          // OUT:
    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 (is required to be a multiple
                                    //     of 16 bytes)
    BYTE            *dIn            // IN: 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)
    {
```

```
77          pIv = iv;
78          for(i = 16; i > 0; i--)
79              *pIv++ ^= *dIn++;
80          AES_encrypt(iv, iv, &AesKey);
81          pIv = iv;
82          for(i = 16; i > 0; i--)
83              *dOut++ = *pIv++;
84      }
85      return CRYPT_SUCCESS;
86  }
```

B.11.4.2. _cpri__AESDecryptCBC()

This function performs AES decryption in CBC chain mode. The input *dIn* buffer is decrypted into *dOut*.

The input iv buffer is required to have a size equal to the block size (16 bytes). The *dInSize* is required to be a multiple of the block size.

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	*dInSize* is not a multiple of the block size

```
87  LIB_EXPORT CRYPT_RESULT
88  _cpri__AESDecryptCBC(
89      BYTE            *dOut,          // OUT: the decrypted data
90      UINT32           keySizeInBits, // IN: key size in bit
91      BYTE            *key,           // IN: key buffer. The size of this buffer in
92                                      //     bytes is (keySizeInBits + 7) / 8
93      BYTE            *iv,            // IN/OUT: IV for decryption. The size of this
94                                      //     buffer is 16 byte
95      UINT32           dInSize,       // IN: data size
96      BYTE            *dIn            // IN: data buffer
97      )
98  {
99      AES_KEY      AesKey;
100     BYTE        *pIv;
101     int          i;
102     BYTE         tmp[16];
103     BYTE        *pT = NULL;
104     INT32        dSize;
105
106     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
107
108     if(dInSize == 0)
109         return CRYPT_SUCCESS;
110
111     pAssert(dInSize <= INT32_MAX);
112     dSize = (INT32)dInSize;
113
114     // For CBC, the data size must be an even multiple of the
115     // cipher block size
116     if((dSize % 16) != 0)
117         return CRYPT_PARAMETER;
118
119     // Create AES key schedule
120     if (AES_set_decrypt_key(key, keySizeInBits, &AesKey) != 0)
121         FAIL(FATAL_ERROR_INTERNAL);
122
123     // Copy the input data to a temp buffer, decrypt the buffer into the output;
124     // XOR in the IV, and copy the temp buffer to the IV and repeat.
125     for(; dSize > 0; dSize -= 16)
126     {
```

```
127             pT = tmp;
128             for(i = 16; i> 0; i--)
129                 *pT++ = *dIn++;
130             AES_decrypt(tmp, dOut, &AesKey);
131             pIv = iv;
132             pT = tmp;
133             for(i = 16; i> 0; i--)
134             {
135                 *dOut++ ^= *pIv;
136                 *pIv++ = *pT++;
137             }
138         }
139         return CRYPT_SUCCESS;
140     }
```

B.11.4.3. _cpri__AESEncryptCFB()

This function performs AES encryption in CFB chain mode. The *dOut* buffer receives the values encrypted *dIn*. The input *iv* is assumed to be the size of an encryption block (16 bytes). The *iv* buffer will be modified to contain the last encrypted block.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
141     LIB_EXPORT CRYPT_RESULT
142     _cpri__AESEncryptCFB(
143         BYTE            *dOut,          // OUT: the encrypted
144         UINT32           keySizeInBits, // IN: key size in bit
145         BYTE            *key,           // IN: key buffer. The size of this buffer in
146                                         //     bytes is (keySizeInBits + 7) / 8
147         BYTE            *iv,            // IN/OUT: IV for decryption.
148         UINT32           dInSize,       // IN: data size
149         BYTE            *dIn            // IN: data buffer
150         )
151     {
152         BYTE        *pIv = NULL;
153         AES_KEY      AesKey;
154         INT32        dSize;         // Need a signed version of dInSize
155         int          i;
156
157         pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
158
159         if(dInSize == 0)
160             return CRYPT_SUCCESS;
161
162         pAssert(dInSize <= INT32_MAX);
163         dSize = (INT32)dInSize;
164
165         // Create AES encryption key schedule
166         if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
167             FAIL(FATAL_ERROR_INTERNAL);
168
169         // Encrypt the IV into the IV, XOR in the data, and copy to output
170         for(; dSize > 0; dSize -= 16)
171         {
172             // Encrypt the current value of the IV
173             AES_encrypt(iv, iv, &AesKey);
174             pIv = iv;
175             for(i = (int)(dSize < 16) ? dSize : 16; i > 0; i--)
176                 // XOR the data into the IV to create the cipher text
177                 // and put into the output
178                 *dOut++ = *pIv++ ^= *dIn++;
179         }
```

```
180         // If the inner loop (i loop) was smaller than 16, then dSize would have been
181         // smaller than 16 and it is now negative. If it is negative, then it indicates
182         // how many bytes are needed to pad out the IV for the next round.
183         for(; dSize < 0; dSize++)
184             *pIv++ = 0;
185         return CRYPT_SUCCESS;
186     }
```

B.11.4.4. _cpri__AESDecryptCFB()

This function performs AES decrypt in CFB chain mode. The *dOut* buffer receives the values decrypted from *dIn*.

The input *iv* is assumed to be the size of an encryption block (16 bytes). The *iv* buffer will be modified to contain the last decoded block, padded with zeros

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
187     LIB_EXPORT CRYPT_RESULT
188     _cpri__AESDecryptCFB(
189         BYTE            *dOut,          // OUT: the decrypted data
190         UINT32           keySizeInBits, // IN: key size in bit
191         BYTE            *key,           // IN: key buffer. The size of this buffer in
192                                         //     bytes is (keySizeInBits + 7) / 8
193         BYTE            *iv,            // IN/OUT: IV for decryption.
194         UINT32           dInSize,       // IN: data size
195         BYTE            *dIn            // IN: data buffer
196         )
197     {
198         BYTE        *pIv = NULL;
199         BYTE         tmp[16];
200         int          i;
201         BYTE        *pT;
202         AES_KEY      AesKey;
203         INT32        dSize;
204
205         pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
206
207         if(dInSize == 0)
208             return CRYPT_SUCCESS;
209
210         pAssert(dInSize <= INT32_MAX);
211         dSize = (INT32)dInSize;
212
213         // Create AES encryption key schedule
214         if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
215             FAIL(FATAL_ERROR_INTERNAL);
216
217         for(; dSize > 0; dSize -= 16)
218         {
219             // Encrypt the IV into the temp buffer
220             AES_encrypt(iv, tmp, &AesKey);
221             pT = tmp;
222             pIv = iv;
223             for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
224                 // Copy the current cipher text to IV, XOR
225                 // with the temp buffer and put into the output
226                 *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
227         }
228         // If the inner loop (i loop) was smaller than 16, then dSize
229         // would have been smaller than 16 and it is now negative
230         // If it is negative, then it indicates how may fill bytes
```

```
231         // are needed to pad out the IV for the next round.
232         for(; dSize < 0; dSize++)
233             *pIv++ = 0;
234 
235         return CRYPT_SUCCESS;
236     }
```

B.11.4.5. _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.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
237     LIB_EXPORT CRYPT_RESULT
238     _cpri__AESEncryptCTR(
239         BYTE            *dOut,          // OUT: the encrypted data
240         UINT32           keySizeInBits, // IN: key size in bit
241         BYTE            *key,           // IN: key buffer. The size of this buffer in
242                                         //     bytes is (keySizeInBits + 7) / 8
243         BYTE            *iv,            // IN/OUT: IV for decryption.
244         UINT32           dInSize,       // IN: data size
245         BYTE            *dIn            // IN: data buffer
246         )
247     {
248         BYTE         tmp[16];
249         BYTE        *pT;
250         AES_KEY      AesKey;
251         int          i;
252         INT32        dSize;
253 
254         pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
255 
256         if(dInSize == 0)
257             return CRYPT_SUCCESS;
258 
259         pAssert(dInSize <= INT32_MAX);
260         dSize = (INT32)dInSize;
261 
262         // Create AES encryption schedule
263         if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
264             FAIL(FATAL_ERROR_INTERNAL);
265 
266         for(; dSize > 0; dSize -= 16)
267         {
268             // Encrypt the current value of the IV(counter)
269             AES_encrypt(iv, (BYTE *)tmp, &AesKey);
270 
271             //increment the counter (counter is big-endian so start at end)
272             for(i = 15; i >= 0; i--)
273                 if((iv[i] += 1) != 0)
274                     break;
275 
276             // XOR the encrypted counter value with input and put into output
277             pT = tmp;
278             for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
279                 *dOut++ = *dIn++ ^ *pT++;
280         }
281         return CRYPT_SUCCESS;
282     }
```


B.11.4.6. _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)

```
283 //% #define _cpri__AESDecryptCTR(dOut, keySize, key, iv, dInSize, dIn) \
284 //%         _cpri__AESEncryptCTR(                                      \
285 //%                              ((BYTE *)dOut),                       \
286 //%                              ((UINT32)keySize),                    \
287 //%                              ((BYTE *)key),                        \
288 //%                              ((BYTE *)iv),                         \
289 //%                              ((UINT32)dInSize),                    \
290 //%                              ((BYTE *)dIn)                         \
291 //%                             )
292 //%
293 // The //% is used by the prototype extraction program to cause it to include the
294 // 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.11.4.7. _cpri__AESEncryptECB()

AES encryption in ECB mode. The *data* buffer is modified to contain the cipher text.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
295 LIB_EXPORT CRYPT_RESULT
296 _cpri__AESEncryptECB(
297     BYTE            *dOut,          // OUT: encrypted data
298     UINT32           keySizeInBits, // IN: key size in bit
299     BYTE            *key,           // IN: key buffer. The size of this buffer in
300                                     //     bytes is (keySizeInBits + 7) / 8
301     UINT32           dInSize,       // IN: data size
302     BYTE            *dIn            // IN: clear text buffer
303     )
304 {
305     AES_KEY      AesKey;
306     INT32        dSize;
307
308     pAssert(dOut != NULL && key != NULL && dIn != NULL);
309
310     if(dInSize == 0)
311         return CRYPT_SUCCESS;
312
313     pAssert(dInSize <= INT32_MAX);
314     dSize = (INT32)dInSize;
315
316     // For ECB, the data size must be an even multiple of the
317     // cipher block size
318     if((dSize % 16) != 0)
319         return CRYPT_PARAMETER;
320     // Create AES encrypting key schedule
321     if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
322         FAIL(FATAL_ERROR_INTERNAL);
323
324     for(; dSize > 0; dSize -= 16)
325     {
326         AES_encrypt(dIn, dOut, &AesKey);
327         dIn = &dIn[16];
328         dOut = &dOut[16];
329     }
```

```
330         return CRYPT_SUCCESS;
331     }
```

B.11.4.8. _cpri__AESDecryptECB()

This function performs AES decryption using ECB (not recommended). The cipher text *dIn* is decrypted into *dOut*.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
332     LIB_EXPORT CRYPT_RESULT
333     _cpri__AESDecryptECB(
334         BYTE            *dOut,          // OUT: the clear text data
335         UINT32           keySizeInBits, // IN: key size in bit
336         BYTE            *key,           // IN: key buffer. The size of this buffer in
337                                         //     bytes is (keySizeInBits + 7) / 8
338         UINT32           dInSize,       // IN: data size
339         BYTE            *dIn            // IN: cipher text buffer
340         )
341     {
342         AES_KEY      AesKey;
343         INT32        dSize;
344
345         pAssert(dOut != NULL && key != NULL && dIn != NULL);
346
347         if(dInSize == 0)
348             return CRYPT_SUCCESS;
349
350         pAssert(dInSize <= INT32_MAX);
351         dSize = (INT32)dInSize;
352
353         // For ECB, the data size must be an even multiple of the
354         // cipher block size
355         if((dSize % 16) != 0)
356             return CRYPT_PARAMETER;
357
358         // Create AES decryption key schedule
359         if (AES_set_decrypt_key(key, keySizeInBits, &AesKey) != 0)
360             FAIL(FATAL_ERROR_INTERNAL);
361
362         for(; dSize > 0; dSize -= 16)
363         {
364             AES_decrypt(dIn, dOut, &AesKey);
365             dIn = &dIn[16];
366             dOut = &dOut[16];
367         }
368         return CRYPT_SUCCESS;
369     }
```

B.11.4.9. _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.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
370 LIB_EXPORT CRYPT_RESULT
371 _cpri__AESEncryptOFB(
372     BYTE            *dOut,          // OUT: the encrypted/decrypted data
373     UINT32           keySizeInBits, // IN: key size in bit
374     BYTE            *key,           // IN: key buffer. The size of this buffer in
375                                     //     bytes is (keySizeInBits + 7) / 8
376     BYTE            *iv,            // IN/OUT: IV for decryption. The size of this
377                                     //     buffer is 16 byte
378     UINT32           dInSize,       // IN: data size
379     BYTE            *dIn            // IN: data buffer
380     )
381 {
382     BYTE        *pIv;
383     AES_KEY      AesKey;
384     INT32        dSize;
385     int          i;
386
387     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
388
389     if(dInSize == 0)
390         return CRYPT_SUCCESS;
391
392     pAssert(dInSize <= INT32_MAX);
393     dSize = (INT32)dInSize;
394
395     // Create AES key schedule
396     if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
397         FAIL(FATAL_ERROR_INTERNAL);
398
399     // This is written so that dIn and dOut may be the same
400
401     for(; dSize > 0; dSize -= 16)
402     {
403         // Encrypt the current value of the "IV"
404         AES_encrypt(iv, iv, &AesKey);
405
406         // XOR the encrypted IV into dIn to create the cipher text (dOut)
407         pIv = iv;
408         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
409             *dOut++ = (*pIv++ ^ *dIn++);
410     }
411     return CRYPT_SUCCESS;
412 }
```

B.11.4.10._cpri__AESDecryptOFB()

OFB encryption and decryption use the same algorithms for both. The _cpri__AESDecryptOFB() function is implemented as a macro call to _cpri__AESEncrytOFB(). (skip)

```
413 //%#define _cpri__AESDecryptOFB(dOut,keySizeInBits, key, iv, dInSize, dIn) \
414 //%        _cpri__AESEncryptOFB (                                  \
415 //%                              ((BYTE *)dOut),                   \
416 //%                              ((UINT32)keySizeInBits),          \
417 //%                              ((BYTE *)key),                    \
418 //%                              ((BYTE *)iv),                     \
419 //%                              ((UINT32)dInSize),                \
420 //%                              ((BYTE *)dIn)                     \
421 //%                             )
422 //%
```

```
#ifdef  TPM_ALG_SM4     //%
```

B.11.5. SM4 Encryption

B.11.5.1. _cpri__SM4EncryptCBC()

This function performs SM4 encryption in CBC chain mode. The input *dIn* buffer is encrypted into *dOut*.

The input iv buffer is required to have a size equal to the block size (16 bytes). The *dInSize* is required to be a multiple of the block size.

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	*dInSize* is not a multiple of the block size

```
LIB_EXPORT CRYPT_RESULT
_cpri__SM4EncryptCBC(
    BYTE            *dOut,          // OUT:
    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 (is required to be a multiple
                                    //     of 16 bytes)
    BYTE            *dIn            // IN: 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;
```

```
471 }
```

B.11.5.2. _cpri__SM4DecryptCBC()

This function performs SM4 decryption in CBC chain mode. The input *dIn* buffer is decrypted into *dOut*.

The input iv buffer is required to have a size equal to the block size (16 bytes). The *dInSize* is required to be a multiple of the block size.

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	*dInSize* is not a multiple of the block size

```
472 LIB_EXPORT CRYPT_RESULT
473 _cpri__SM4DecryptCBC(
474     BYTE            *dOut,          // OUT: the decrypted data
475     UINT32           keySizeInBits, // IN: key size in bit
476     BYTE            *key,           // IN: key buffer. The size of this buffer in
477                                     //     bytes is (keySizeInBits + 7) / 8
478     BYTE            *iv,            // IN/OUT: IV for decryption. The size of this
479                                     //     buffer is 16 byte
480     UINT32           dInSize,       // IN: data size
481     BYTE            *dIn            // IN: data buffer
482     )
483 {
484     SM4_KEY      Sm4Key;
485     BYTE        *pIv;
486     int          i;
487     BYTE         tmp[16];
488     BYTE        *pT = NULL;
489     INT32        dSize;
490
491     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
492
493     if(dInSize == 0)
494         return CRYPT_SUCCESS;
495
496     pAssert(dInSize <= INT32_MAX);
497     dSize = (INT32)dInSize;
498
499     // For CBC, the data size must be an even multiple of the
500     // cipher block size
501     if((dSize % 16) != 0)
502         return CRYPT_PARAMETER;
503
504     // Create SM4 key schedule
505     if (SM4_set_decrypt_key(key, keySizeInBits, &Sm4Key) != 0)
506         FAIL(FATAL_ERROR_INTERNAL);
507
508     // Copy the input data to a temp buffer, decrypt the buffer into the output;
509     // XOR in the IV, and copy the temp buffer to the IV and repeat.
510     for(; dSize > 0; dSize -= 16)
511     {
512         pT = tmp;
513         for(i = 16; i> 0; i--)
514             *pT++ = *dIn++;
515         SM4_decrypt(tmp, dOut, &Sm4Key);
516         pIv = iv;
517         pT = tmp;
518         for(i = 16; i> 0; i--)
519         {
520             *dOut++ ^= *pIv;
```

```
521                 *pIv++ = *pT++;
522             }
523         }
524         return CRYPT_SUCCESS;
525     }
```

B.11.5.3. _cpri__SM4EncryptCFB()

This function performs SM4 encryption in CFB chain mode. The *dOut* buffer receives the values encrypted *dIn*. The input *iv* is assumed to be the size of an encryption block (16 bytes). The *iv* buffer will be modified to contain the last encrypted block.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
526     LIB_EXPORT CRYPT_RESULT
527     _cpri__SM4EncryptCFB(
528         BYTE            *dOut,          // OUT: the encrypted
529         UINT32           keySizeInBits, // IN: key size in bit
530         BYTE            *key,           // IN: key buffer. The size of this buffer in
531                                         //     bytes is (keySizeInBits + 7) / 8
532         BYTE            *iv,            // IN/OUT: IV for decryption.
533         UINT32           dInSize,       // IN: data size
534         BYTE            *dIn            // IN: data buffer
535         )
536     {
537         BYTE        *pIv;
538         SM4_KEY      Sm4Key;
539         INT32        dSize;         // Need a signed version of dInSize
540         int          i;
541
542         pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
543
544         if(dInSize == 0)
545             return CRYPT_SUCCESS;
546
547         pAssert(dInSize <= INT32_MAX);
548         dSize = (INT32)dInSize;
549
550         // Create SM4 encryption key schedule
551         if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
552             FAIL(FATAL_ERROR_INTERNAL);
553
554         // Encrypt the IV into the IV, XOR in the data, and copy to output
555         for(; dSize > 0; dSize -= 16)
556         {
557             // Encrypt the current value of the IV
558             SM4_encrypt(iv, iv, &Sm4Key);
559             pIv = iv;
560             for(i = (int)(dSize < 16) ? dSize : 16; i > 0; i--)
561                 // XOR the data into the IV to create the cipher text
562                 // and put into the output
563                 *dOut++ = *pIv++ ^= *dIn++;
564         }
565         return CRYPT_SUCCESS;
566     }
```

B.11.5.4. _cpri__SM4DecryptCFB()

This function performs SM4 decrypt in CFB chain mode. The *dOut* buffer receives the values decrypted from *dIn*.

The input *iv* is assumed to be the size of an encryption block (16 bytes). The *iv* buffer will be modified to contain the last decoded block, padded with zeros

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
567  LIB_EXPORT CRYPT_RESULT
568  _cpri__SM4DecryptCFB(
569      BYTE            *dOut,          // OUT: the decrypted data
570      UINT32           keySizeInBits, // IN: key size in bit
571      BYTE            *key,           // IN: key buffer. The size of this buffer in
572                                      //     bytes is (keySizeInBits + 7) / 8
573      BYTE            *iv,            // IN/OUT: IV for decryption.
574      UINT32           dInSize,       // IN: data size
575      BYTE            *dIn            // IN: data buffer
576      )
577  {
578      BYTE        *pIv;
579      BYTE         tmp[16];
580      int          i;
581      BYTE        *pT;
582      SM4_KEY      Sm4Key;
583      INT32        dSize;
584
585      pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
586
587      if(dInSize == 0)
588          return CRYPT_SUCCESS;
589
590      pAssert(dInSize <= INT32_MAX);
591      dSize = (INT32)dInSize;
592
593      // Create SM4 encryption key schedule
594      if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
595          FAIL(FATAL_ERROR_INTERNAL);
596
597      for(; dSize > 0; dSize -= 16)
598      {
599          // Encrypt the IV into the temp buffer
600          SM4_encrypt(iv, tmp, &Sm4Key);
601          pT = tmp;
602          pIv = iv;
603          for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
604              // Copy the current cipher text to IV, XOR
605              // with the temp buffer and put into the output
606              *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
607      }
608      // If the inner loop (i loop) was smaller than 16, then dSize
609      // would have been smaller than 16 and it is now negative
610      // If it is negative, then it indicates how may fill bytes
611      // are needed to pad out the IV for the next round.
612      for(; dSize < 0; dSize++)
613          *iv++ = 0;
614
615      return CRYPT_SUCCESS;
616  }
```

B.11.5.5. _cpri__SM4EncryptCTR()

This function performs SM4 encryption/decryption in CTR chain mode. The *dIn* buffer is encrypted into *dOut*. The input iv buffer is assumed to have a size equal to the SM4 block size (16 bytes). The iv will be incremented by the number of blocks (full and partial) that were encrypted.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
617 LIB_EXPORT CRYPT_RESULT
618 _cpri__SM4EncryptCTR(
619     BYTE            *dOut,          // OUT: the encrypted data
620     UINT32           keySizeInBits, // IN: key size in bit
621     BYTE            *key,           // IN: key buffer. The size of this buffer in
622                                     //     bytes is (keySizeInBits + 7) / 8
623     BYTE            *iv,            // IN/OUT: IV for decryption.
624     UINT32           dInSize,       // IN: data size
625     BYTE            *dIn            // IN: data buffer
626     )
627 {
628     BYTE         tmp[16];
629     BYTE        *pT;
630     SM4_KEY      Sm4Key;
631     int          i;
632     INT32        dSize;
633
634     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
635
636     if(dInSize == 0)
637         return CRYPT_SUCCESS;
638
639     pAssert(dInSize <= INT32_MAX);
640     dSize = (INT32)dInSize;
641
642     // Create SM4 encryption schedule
643     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
644         FAIL(FATAL_ERROR_INTERNAL);
645
646     for(; dSize > 0; dSize--)
647     {
648         // Encrypt the current value of the IV(counter)
649         SM4_encrypt(iv, (BYTE *)tmp, &Sm4Key);
650
651         //increment the counter
652         for(i = 0; i < 16; i++)
653             if((iv[i] += 1) != 0)
654                 break;
655
656         // XOR the encrypted counter value with input and put into output
657         pT = tmp;
658         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
659             *dOut++ = *dIn++ ^ *pT++;
660     }
661     return CRYPT_SUCCESS;
662 }
```

B.11.5.6. _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)

```
663 //% #define _cpri__SM4DecryptCTR(dOut, keySize, key, iv, dInSize, dIn) \
664 //%         _cpri__SM4EncryptCTR(                                     \
665 //%                              ((BYTE *)dOut),                      \
666 //%                              ((UINT32)keySize),                   \
667 //%                              ((BYTE *)key),                       \
668 //%                              ((BYTE *)iv),                        \
669 //%                              ((UINT32)dInSize),                   \
```

```
670 //%                              ((BYTE *)dIn)              \
671 //%                            )
672 //%
673 // The //% is used by the prototype extraction program to cause it to include the
674 // 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.11.5.7. _cpri__SM4EncryptECB()

SM4 encryption in ECB mode. The *data* buffer is modified to contain the cipher text.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
675 LIB_EXPORT CRYPT_RESULT
676 _cpri__SM4EncryptECB(
677     BYTE            *dOut,          // OUT: encrypted data
678     UINT32           keySizeInBits, // IN: key size in bit
679     BYTE            *key,           // IN: key buffer. The size of this buffer in
680                                     //     bytes is (keySizeInBits + 7) / 8
681     UINT32           dInSize,       // IN: data size
682     BYTE            *dIn            // IN: clear text buffer
683     )
684 {
685     SM4_KEY      Sm4Key;
686     INT32        dSize;
687
688     pAssert(dOut != NULL && key != NULL && dIn != NULL);
689
690     if(dInSize == 0)
691         return CRYPT_SUCCESS;
692
693     pAssert(dInSize <= INT32_MAX);
694     dSize = (INT32)dInSize;
695
696     // For ECB, the data size must be an even multiple of the
697     // cipher block size
698     if((dSize % 16) != 0)
699         return CRYPT_PARAMETER;
700     // Create SM4 encrypting key schedule
701     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
702         FAIL(FATAL_ERROR_INTERNAL);
703
704     for(; dSize > 0; dSize -= 16)
705     {
706         SM4_encrypt(dIn, dOut, &Sm4Key);
707         dIn = &dIn[16];
708         dOut = &dOut[16];
709     }
710     return CRYPT_SUCCESS;
711 }
```

B.11.5.8. _cpri__SM4DecryptECB()

This function performs SM4 decryption using ECB (not recommended). The cipher text *dIn* is decrypted into *dOut*.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
712  LIB_EXPORT CRYPT_RESULT
713  _cpri__SM4DecryptECB(
714      BYTE            *dOut,          // OUT: the clear text data
715      UINT32           keySizeInBits, // IN: key size in bit
716      BYTE            *key,           // IN: key buffer. The size of this buffer in
717                                      //     bytes is (keySizeInBits + 7) / 8
718      UINT32           dInSize,       // IN: data size
719      BYTE            *dIn            // IN: cipher text buffer
720      )
721  {
722      SM4_KEY      Sm4Key;
723      INT32        dSize;
724
725      pAssert(dOut != NULL && key != NULL && dIn != NULL);
726
727      if(dInSize == 0)
728          return CRYPT_SUCCESS;
729
730      pAssert(dInSize <= INT32_MAX);
731      dSize = (INT32)dInSize;
732
733      // For ECB, the data size must be an even multiple of the
734      // cipher block size
735      if((dSize % 16) != 0)
736          return CRYPT_PARAMETER;
737
738      // Create SM4 decryption key schedule
739      if (SM4_set_decrypt_key(key, keySizeInBits, &Sm4Key) != 0)
740          FAIL(FATAL_ERROR_INTERNAL);
741
742      for(; dSize > 0; dSize -= 16)
743      {
744          SM4_decrypt(dIn, dOut, &Sm4Key);
745          dIn = &dIn[16];
746          dOut = &dOut[16];
747      }
748      return CRYPT_SUCCESS;
749  }
```

B.11.5.9. _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.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
750  LIB_EXPORT CRYPT_RESULT
751  _cpri__SM4EncryptOFB(
752      BYTE            *dOut,          // OUT: the encrypted/decrypted data
753      UINT32           keySizeInBits, // IN: key size in bit
754      BYTE            *key,           // IN: key buffer. The size of this buffer in
755                                      //     bytes is (keySizeInBits + 7) / 8
756      BYTE            *iv,            // IN/OUT: IV for decryption. The size of this
757                                      //     buffer is 16 byte
```

```
758         UINT32           dInSize,       // IN: data size
759         BYTE            *dIn            // IN: data buffer
760         )
761     {
762         BYTE        *pIv;
763         SM4_KEY      Sm4Key;
764         INT32        dSize;
765         int          i;
766
767         pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
768
769         if(dInSize == 0)
770             return CRYPT_SUCCESS;
771
772         pAssert(dInSize <= INT32_MAX);
773         dSize = (INT32)dInSize;
774
775         // Create SM4 key schedule
776         if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
777             FAIL(FATAL_ERROR_INTERNAL);
778
779         // This is written so that dIn and dOut may be the same
780
781         for(; dSize > 0; dSize -= 16)
782         {
783             // Encrypt the current value of the "IV"
784             SM4_encrypt(iv, iv, &Sm4Key);
785
786             // XOR the encrypted IV into dIn to create the cipher text (dOut)
787             pIv = iv;
788             for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
789                 *dOut++ = (*pIv++ ^ *dIn++);
790         }
791         return CRYPT_SUCCESS;
792     }
```

B.11.5.10._cpri__SM4DecryptOFB()

OFB encryption and decryption use the same algorithms for both. The _cpri__SM4DecryptOFB() function is implemented as a macro call to _cpri__SM4EncrytOFB(). (skip)

```
793     //%#define _cpri__SM4DecryptOFB(dOut,keySizeInBits, key, iv, dInSize, dIn) \
794     //%        _cpri__SM4EncryptOFB (                                          \
795     //%                              ((BYTE *)dOut),                           \
796     //%                              ((UINT32)keySizeInBits),                  \
797     //%                              ((BYTE *)key),                            \
798     //%                              ((BYTE *)iv),                             \
799     //%                              ((UINT32)dInSize),                        \
800     //%                              ((BYTE *)dIn)                             \
801     //%                             )
802     //%
803     #endif      //% TPM_ALG_SM4
```

B.12 RSA Files

B.12.1. CpriRSA.c

B.12.1.1. Introduction

This file contains implementation of crypto primitives for RSA. This is a simulator of a crypto engine. Vendors may replace the implementation in this file with their own library functions.

Integer format: the big integers passed in/out to the function interfaces in this library adopt the same format used in 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.12.1.2. Includes

```
#include "OsslCryptoEngine.h"
#ifdef  TPM_ALG_RSA
```

B.12.1.3. Local Functions

B.12.1.3.1. RsaPrivateExponent()

This function computes the private exponent *de* = 1 mod *(p*-1)*(q*-1) The inputs are the public modulus and one of the primes.

The results are returned in the key->private structure. The size of that structure is expanded to hold the private exponent. If the computed value is smaller than the public modulus, the private exponent is de-normalized.

Return Value	Meaning
CRYPT_SUCCESS	private exponent computed
CRYPT_PARAMETER	prime is not half the size of the modulus, or the modulus is not evenly divisible by the prime, or no private exponent could be computed from the input parameters

```
static CRYPT_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;

    CRYPT_RESULT     retVal = CRYPT_SUCCESS;     // Assume success

    pAssert(key != NULL && key->privateKey != NULL && key->publicKey != NULL);

    context = BN_CTX_new();
```

```
24          if(context == NULL)
25              FAIL(FATAL_ERROR_ALLOCATION);
26          BN_CTX_start(context);
27          bnE = BN_CTX_get(context);
28          bnD = BN_CTX_get(context);
29          bnN = BN_CTX_get(context);
30          bnP = BN_CTX_get(context);
31          bnPhi = BN_CTX_get(context);
32          bnQ = BN_CTX_get(context);
33          bnQr = BN_CTX_get(context);
34
35          if(bnQr == NULL)
36              FAIL(FATAL_ERROR_ALLOCATION);
37
38          // Assume the size of the public key value is within range
39          pAssert(key->publicKey->size <= MAX_RSA_KEY_BYTES);
40
41          if(   BN_bin2bn(key->publicKey->buffer, key->publicKey->size, bnN) == NULL
42             || BN_bin2bn(key->privateKey->buffer, key->privateKey->size, bnP) == NULL)
43
44              FAIL(FATAL_ERROR_INTERNAL);
45
46          // If P size is not 1/2 of n size, then this is not a valid value for this
47          // implementation. This will also catch the case were P is input as zero.
48          // This generates a return rather than an assert because the key being loaded
49          // might be SW generated and wrong.
50          if(BN_num_bits(bnP) < BN_num_bits(bnN)/2)
51          {
52              retVal = CRYPT_PARAMETER;
53              goto Cleanup;
54          }
55          // Get q = n/p;
56          if (BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
57              FAIL(FATAL_ERROR_INTERNAL);
58
59          // If there is a remainder, then this is not a valid n
60          if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
61          {
62              retVal = CRYPT_PARAMETER;       // problem may be recoverable
63              goto Cleanup;
64          }
65          // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
66          if(   BN_copy(bnPhi, bnN) == NULL
67             || !BN_sub(bnPhi, bnPhi, bnP)
68             || !BN_sub(bnPhi, bnPhi, bnQ)
69             || !BN_add_word(bnPhi, 1))
70              FAIL(FATAL_ERROR_INTERNAL);
71
72          // Compute the multiplicative inverse
73          BN_set_word(bnE, key->exponent);
74          if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
75          {
76              // Going to assume that the error is caused by a bad
77              // set of parameters. Specifically, an exponent that is
78              // not compatible with the primes. In an implementation that
79              // has better visibility to the error codes, this might be
80              // refined so that failures in the library would return
81              // a more informative value.  Should not assume here that
82              // the error codes will remain unchanged.
83
84              retVal = CRYPT_PARAMETER;
85              goto Cleanup;
86          }
87
88          fill = key->publicKey->size - BN_num_bytes(bnD);
89          BN_bn2bin(bnD, &key->privateKey->buffer[fill]);
```

```
90      memset(key->privateKey->buffer, 0, fill);
91
92      // Change the size of the private key so that it is known to contain
93      // a private exponent rather than a prime.
94      key->privateKey->size = key->publicKey->size;
95
96  Cleanup:
97      BN_CTX_end(context);
98      BN_CTX_free(context);
99      return retVal;
100 }
```

B.12.1.3.2. _cpri__TestKeyRSA()

This function computes the private exponent *de* = 1 mod *(p*-1)*(q*-1) The inputs are the public modulus and one of the primes or two primes.

If both primes are provided, the public modulus is computed. If only one prime is provided, the second prime is computed. In either case, a private exponent is produced and placed in *d*.

If no modular inverse exists, then CRYPT_PARAMETER is returned.

Return Value	Meaning
CRYPT_SUCCESS	private exponent (d) was generated
CRYPT_PARAMETER	one or more parameters are invalid

```
101 LIB_EXPORT CRYPT_RESULT
102 _cpri__TestKeyRSA(
103     TPM2B           *d,             // OUT: the address to receive the private
104                                     //      exponent
105     UINT32           exponent,      // IN: the public modulu
106     TPM2B           *publicKey,     // IN/OUT: an input if only one prime is
107                                     //      provided. an output if both primes are
108                                     //      provided
109     TPM2B           *prime1,        // IN: a first prime
110     TPM2B           *prime2         // IN: an optional second prime
111     )
112 {
113     BN_CTX          *context;
114     BIGNUM          *bnD;
115     BIGNUM          *bnN;
116     BIGNUM          *bnP;
117     BIGNUM          *bnE;
118     BIGNUM          *bnPhi;
119     BIGNUM          *bnQ;
120     BIGNUM          *bnQr;
121     UINT32          fill;
122
123     CRYPT_RESULT    retVal = CRYPT_SUCCESS;     // Assume success
124
125     pAssert(publicKey != NULL && prime1 != NULL);
126     // Make sure that the sizes are within range
127     pAssert(   prime1->size <= MAX_RSA_KEY_BYTES/2
128             && publicKey->size <= MAX_RSA_KEY_BYTES);
129     pAssert( prime2 == NULL || prime2->size < MAX_RSA_KEY_BYTES/2);
130
131     if(publicKey->size/2 != prime1->size)
132         return CRYPT_PARAMETER;
133
134     context = BN_CTX_new();
135     if(context == NULL)
136         FAIL(FATAL_ERROR_ALLOCATION);
137     BN_CTX_start(context);
```

```
138         bnE = BN_CTX_get(context);      // public exponent (e)
139         bnD = BN_CTX_get(context);      // private exponent (d)
140         bnN = BN_CTX_get(context);      // public modulus (n)
141         bnP = BN_CTX_get(context);      // prime1 (p)
142         bnPhi = BN_CTX_get(context);    // (p-1)(q-1)
143         bnQ = BN_CTX_get(context);      // prime2 (q)
144         bnQr = BN_CTX_get(context);     // n mod p
145
146         if(bnQr == NULL)
147             FAIL(FATAL_ERROR_ALLOCATION);
148
149         if(BN_bin2bn(prime1->buffer, prime1->size, bnP) == NULL)
150             FAIL(FATAL_ERROR_INTERNAL);
151
152         // If prime2 is provided, then compute n
153         if(prime2 != NULL)
154         {
155             // Two primes provided so use them to compute n
156             if(BN_bin2bn(prime2->buffer, prime2->size, bnQ) == NULL)
157                 FAIL(FATAL_ERROR_INTERNAL);
158
159             // Make sure that the sizes of the primes are compatible
160             if(BN_num_bits(bnQ) != BN_num_bits(bnP))
161             {
162                 retVal = CRYPT_PARAMETER;
163                 goto Cleanup;
164             }
165             // Multiply the primes to get the public modulus
166
167             if(BN_mul(bnN, bnP, bnQ, context) != 1)
168                 FAIL(FATAL_ERROR_INTERNAL);
169
170             // if the space provided for the public modulus is large enough,
171             // save the created value
172             if(BN_num_bits(bnN) != (publicKey->size * 8))
173             {
174                 retVal = CRYPT_PARAMETER;
175                 goto Cleanup;
176             }
177             BN_bn2bin(bnN, publicKey->buffer);
178         }
179         else
180         {
181             // One prime provided so find the second prime by division
182             BN_bin2bn(publicKey->buffer, publicKey->size, bnN);
183
184             // Get q = n/p;
185             if(BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
186                 FAIL(FATAL_ERROR_INTERNAL);
187
188             // If there is a remainder, then this is not a valid n
189             if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
190             {
191                 retVal = CRYPT_PARAMETER;       // problem may be recoverable
192                 goto Cleanup;
193             }
194         }
195         // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
196         BN_copy(bnPhi, bnN);
197         BN_sub(bnPhi, bnPhi, bnP);
198         BN_sub(bnPhi, bnPhi, bnQ);
199         BN_add_word(bnPhi, 1);
200         // Compute the multiplicative inverse
201         BN_set_word(bnE, exponent);
202         if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
203         {
```

```
204             // Going to assume that the error is caused by a bad set of parameters.
205             // Specifically, an exponent that is not compatible with the primes.
206             // In an implementation that has better visibility to the error codes,
207             // this might be refined so that failures in the library would return
208             // a more informative value.
209             // Do not assume that the error codes will remain unchanged.
210             retVal = CRYPT_PARAMETER;
211             goto Cleanup;
212         }
213         // Return the private exponent.
214         // Make sure it is normalized to have the correct size.
215         d->size = publicKey->size;
216         fill = d->size - BN_num_bytes(bnD);
217         BN_bn2bin(bnD, &d->buffer[fill]);
218         memset(d->buffer, 0, fill);
219     Cleanup:
220         BN_CTX_end(context);
221         BN_CTX_free(context);
222         return retVal;
223     }
```

B.12.1.3.3. RSAEP()

This function performs the RSAEP operation defined in PKCS#1v2.1. It is an exponentiation of a value *(m)* with the public exponent *(e)*, modulo the public *(n)*.

Return Value	Meaning
CRYPT_SUCCESS	encryption complete
CRYPT_PARAMETER	number to exponentiate is larger than the modulus

```
224     static CRYPT_RESULT
225     RSAEP (
226         UINT32           dInOutSize,    // OUT size of the encrypted block
227         BYTE            *dInOut,        // OUT: the encrypted data
228         RSA_KEY         *key            // IN: the key to use
229         )
230     {
231         UINT32       e;
232         BYTE         exponent[4];
233         CRYPT_RESULT retVal;
234
235         e = key->exponent;
236         if(e == 0)
237             e = RSA_DEFAULT_PUBLIC_EXPONENT;
238         UINT32_TO_BYTE_ARRAY(e, exponent);
239
240         //!!! Can put check for test of RSA here
241
242         retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut, 4, exponent,
243                                key->publicKey->size, key->publicKey->buffer);
244
245         // Exponentiation result is stored in-place, thus no space shortage is possible.
246         pAssert(retVal != CRYPT_UNDERFLOW);
247
248         return retVal;
249     }
```

B.12.1.3.4. RSADP()

This function performs the RSADP operation defined in PKCS#1v2.1. It is an exponentiation of a value *(c)* with the private exponent *(d)*, modulo the public modulus *(n)*. The decryption is in place.

This function also checks the size of the private key. If the size indicates that only a prime value is present, the key is converted to being a private exponent.

Return Value	Meaning
CRYPT_SUCCESS	decryption succeeded
CRYPT_PARAMETER	the value to decrypt is larger than the modulus

```
250  static CRYPT_RESULT
251  RSADP (
252      UINT32           dInOutSize,     // IN/OUT: size of decrypted data
253      BYTE            *dInOut,         // IN/OUT: the decrypted data
254      RSA_KEY         *key             // IN: the key
255      )
256  {
257      CRYPT_RESULT retVal;
258
259      //!!! Can put check for RSA tested here
260
261      // Make sure that the pointers are provided and that the private key is present
262      // If the private key is present it is assumed to have been created by
263      // so is presumed good _cpri__PrivateExponent
264      pAssert(key != NULL && dInOut != NULL &&
265              key->publicKey->size == key->publicKey->size);
266
267      // make sure that the value to be decrypted is smaller than the modulus
268      // note: this check is redundant as is also performed by _math__ModExp()
269      // which is optimized for use in RSA operations
270      if(_math__uComp(key->publicKey->size, key->publicKey->buffer,
271                      dInOutSize, dInOut) <= 0)
272          return CRYPT_PARAMETER;
273
274      // _math__ModExp can return CRYPT_PARAMTER or CRYPT_UNDERFLOW but actual
275      // underflow is not possible because everything is in the same buffer.
276      retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut,
277                             key->privateKey->size, key->privateKey->buffer,
278                             key->publicKey->size, key->publicKey->buffer);
279
280      // Exponentiation result is stored in-place, thus no space shortage is possible.
281      pAssert(retVal != CRYPT_UNDERFLOW);
282
283      return retVal;
284  }
```

B.12.1.3.5. OaepEncode()

This function performs OAEP padding. The size of the buffer to receive the OAEP padded data must equal the size of the modulus

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	*hashAlg* is not valid
CRYPT_FAIL	message size is too large

```
285  static CRYPT_RESULT
286  OaepEncode(
287      UINT32       paddedSize,     // IN: pad value size
288      BYTE        *padded,         // OUT: the pad data
289      TPM_ALG_ID   hashAlg,        // IN: algorithm to use for padding
290      const char  *label,          // IN: null-terminated string (may be NULL)
```

```
291         UINT32       messageSize,   // IN: the message size
292         BYTE        *message        // IN: the message being padded
293     #ifdef  TEST_RSA                //
294         ,  BYTE        *testSeed    // IN: optional seed used for testing.
295     #endif  // TEST_RSA             //
296     )
297     {
298         UINT32       padLen;
299         UINT32       dbSize;
300         UINT32       i;
301         BYTE         mySeed[MAX_DIGEST_SIZE];
302         BYTE        *seed = mySeed;
303         INT32        hLen = _cpri__GetDigestSize(hashAlg);
304         BYTE         mask[MAX_RSA_KEY_BYTES];
305         BYTE        *pp;
306         BYTE        *pm;
307         UINT32       lSize = 0;
308         CRYPT_RESULT retVal = CRYPT_SUCCESS;
309
310         pAssert(padded != NULL && message != NULL);
311
312         // A value of zero is not allowed because the KDF can't produce a result
313         // if the digest size is zero.
314         if(hLen <= 0)
315             return CRYPT_PARAMETER;
316
317         // If a label is provided, get the length of the string, including the
318         // terminator
319         if(label != NULL)
320             lSize = (UINT32)strlen(label) + 1;
321
322         // Basic size check
323         // messageSize <= k  2hLen  2
324         if(messageSize > paddedSize - 2 * hLen - 2)
325             return CRYPT_FAIL;
326
327         // Hash L even if it is null
328         // Offset into padded leaving room for masked seed and byte of zero
329         pp = &padded[hLen + 1];
330         retVal = _cpri__HashBlock(hashAlg, lSize, (BYTE *)label, hLen, pp);
331
332         // concatenate PS of k  mLen  2hLen  2
333         padLen = paddedSize - messageSize - (2 * hLen) - 2;
334         memset(&pp[hLen], 0, padLen);
335         pp[hLen+padLen] = 0x01;
336         padLen += 1;
337         memcpy(&pp[hLen+padLen], message, messageSize);
338
339         // The total size of db = hLen + pad + mSize;
340         dbSize = hLen+padLen+messageSize;
341
342         // If testing, then use the provided seed. Otherwise, use values
343         // from the RNG
344     #ifdef  TEST_RSA
345         if(testSeed != NULL)
346             seed = testSeed;
347         else
348     #endif  // TEST_RSA
349             _cpri__GenerateRandom(hLen, mySeed);
350
351         // mask = MGF1 (seed, nSize  hLen  1)
352         if((retVal = _cpri__MGF1(dbSize, mask,  hashAlg, hLen, seed)) < 0)
353             return retVal; // Don't expect an error because hash size is not zero
354                            // was detected in the call to _cpri__HashBlock() above.
355
356         // Create the masked db
```

```
357         pm = mask;
358         for(i = dbSize; i > 0; i--)
359             *pp++ ^= *pm++;
360         pp = &padded[hLen + 1];
361 
362         // Run the masked data through MGF1
363         if((retVal = _cpri__MGF1(hLen, &padded[1],  hashAlg, dbSize, pp)) < 0)
364             return retVal; // Don't expect zero here as the only case for zero
365                            // was detected in the call to _cpri__HashBlock() above.
366 
367         // Now XOR the seed to create masked seed
368         pp = &padded[1];
369         pm = seed;
370         for(i = hLen; i > 0; i--)
371             *pp++ ^= *pm++;
372 
373         // Set the first byte to zero
374         *padded = 0x00;
375         return CRYPT_SUCCESS;
376     }
```

B.12.1.3.6. OaepDecode()

This function performs OAEP padding checking. The size of the buffer to receive the recovered data. If the padding is not valid, the *dSize* size is set to zero and the function returns CRYPT_NO_RESULTS.

The *dSize* parameter is used as an input to indicate the size available in the buffer. If insufficient space is available, the size is not changed and the return code is CRYPT_FAIL.

Return Value	Meaning
CRYPT_SUCCESS	decode complete
CRYPT_PARAMETER	the value to decode was larger than the modulus
CRYPT_FAIL	the padding is wrong or the buffer to receive the results is too small

```
377     static CRYPT_RESULT
378     OaepDecode(
379         UINT32          *dataOutSize,   // IN/OUT: the recovered data size
380         BYTE            *dataOut,       // OUT: the recovered data
381         TPM_ALG_ID       hashAlg,       // IN: algorithm to use for padding
382         const char      *label,         // IN: null-terminated string (may be NULL)
383         UINT32           paddedSize,    // IN: the size of the padded data
384         BYTE            *padded         // IN: the padded data
385         )
386     {
387         UINT32       dSizeSave;
388         UINT32       i;
389         BYTE         seedMask[MAX_DIGEST_SIZE];
390         INT32        hLen = _cpri__GetDigestSize(hashAlg);
391 
392         BYTE         mask[MAX_RSA_KEY_BYTES];
393         BYTE        *pp;
394         BYTE        *pm;
395         UINT32       lSize = 0;
396         CRYPT_RESULT retVal = CRYPT_SUCCESS;
397 
398         // Unknown hash
399         pAssert(hLen > 0 && dataOutSize != NULL && dataOut != NULL && padded != NULL);
400 
401         // If there is a label, get its size including the terminating 0x00
402         if(label != NULL)
403             lSize = (UINT32)strlen(label) + 1;
404 
```

```
405         // Set the return size to zero so that it doesn't have to be done on each
406         // failure
407         dSizeSave = *dataOutSize;
408         *dataOutSize = 0;
409 
410         // Strange size (anything smaller can't be an OAEP padded block)
411         // Also check for no leading 0
412         if(paddedSize < (unsigned)((2 * hLen) + 2) || *padded != 0)
413             return CRYPT_FAIL;
414 
415         // Use the hash size to determine what to put through MGF1 in order
416         // to recover the seedMask
417         if((retVal = _cpri__MGF1(hLen, seedMask,  hashAlg,
418                                  paddedSize-hLen-1, &padded[hLen+1])) < 0)
419             return retVal;
420 
421         // Recover the seed into seedMask
422         pp = &padded[1];
423         pm = seedMask;
424         for(i = hLen; i > 0; i--)
425             *pm++ ^= *pp++;
426 
427         // Use the seed to generate the data mask
428         if((retVal = _cpri__MGF1(paddedSize-hLen-1, mask,  hashAlg,
429                                  hLen, seedMask)) < 0)
430             return retVal;
431 
432         // Use the mask generated from seed to recover the padded data
433         pp = &padded[hLen+1];
434         pm = mask;
435         for(i = paddedSize-hLen-1; i > 0; i--)
436             *pm++ ^= *pp++;
437 
438         // Make sure that the recovered data has the hash of the label
439         // Put trial value in the seed mask
440         if((retVal=_cpri__HashBlock(hashAlg, lSize,(BYTE *)label, hLen, seedMask)) < 0)
441             return retVal;
442 
443         if(memcmp(seedMask, mask, hLen) != 0)
444             return CRYPT_FAIL;
445 
446         // find the start of the data
447         pm = &mask[hLen];
448         for(i = paddedSize-(2*hLen)-1; i > 0; i--)
449         {
450             if(*pm++ != 0)
451                 break;
452         }
453         if(i == 0)
454             return CRYPT_PARAMETER;
455 
456         // pm should be pointing at the first part of the data
457         // and i is one greater than the number of bytes to move
458         i--;
459         if(i > dSizeSave)
460         {
461             // Restore dSize
462             *dataOutSize = dSizeSave;
463             return CRYPT_FAIL;
464         }
465         memcpy(dataOut, pm, i);
466         *dataOutSize = i;
467         return CRYPT_SUCCESS;
468     }
```


B.12.1.3.7. PKSC1v1_5Encode()

This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

Return Value	Meaning
CRYPT_SUCCESS	data encoded
CRYPT_PARAMETER	message size is too large

```
469  static CRYPT_RESULT
470  RSAES_PKSC1v1_5Encode(
471      UINT32           paddedSize,    // IN: pad value size
472      BYTE            *padded,        // OUT: the pad data
473      UINT32           messageSize,   // IN: the message size
474      BYTE            *message        // IN: the message being padded
475      )
476  {
477      UINT32      ps = paddedSize - messageSize - 3;
478      if(messageSize > paddedSize - 11)
479          return CRYPT_PARAMETER;
480
481      // move the message to the end of the buffer
482      memcpy(&padded[paddedSize - messageSize], message, messageSize);
483
484      // Set the first byte to 0x00 and the second to 0x02
485      *padded = 0;
486      padded[1] = 2;
487
488      // Fill with random bytes
489      _cpri__GenerateRandom(ps, &padded[2]);
490
491      // Set the delimiter for the random field to 0
492      padded[2+ps] = 0;
493
494      // Now, the only messy part. Make sure that all the ps bytes are non-zero
495      // In this implementation, use the value of the current index
496      for(ps++; ps > 1; ps--)
497      {
498          if(padded[ps] == 0)
499              padded[ps] = 0x55;    // In the < 0.5% of the cases that the random
500                                    // value is 0, just pick a value to put into
501                                    // the spot.
502      }
503      return CRYPT_SUCCESS;
504  }
```

B.12.1.3.8. RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decoding error or results would no fit into provided buffer

```
505  static CRYPT_RESULT
506  RSAES_Decode(
507      UINT32          *messageSize,   // IN/OUT: recovered message size
508      BYTE            *message,       // OUT: the recovered message
509      UINT32           codedSize,     // IN: the encoded message size
510      BYTE            *coded          // IN: the encoded message
511      )
```

```
512 {
513     BOOL        fail = FALSE;
514     UINT32      ps;
515
516     fail = (codedSize < 11);
517     fail |= (coded[0] != 0x00) || (coded[1] != 0x02);
518     for(ps = 2; ps < codedSize; ps++)
519     {
520         if(coded[ps] == 0)
521             break;
522     }
523     ps++;
524
525     // Make sure that ps has not gone over the end and that there are at least 8
526     // bytes of pad data.
527     fail |= ((ps >= codedSize) || ((ps-2) < 8));
528     if((*messageSize < codedSize - ps) || fail)
529         return CRYPT_FAIL;
530
531     *messageSize = codedSize - ps;
532     memcpy(message, &coded[ps], codedSize - ps);
533     return CRYPT_SUCCESS;
534 }
```

B.12.1.3.9. PssEncode()

This function creates an encoded block of data that is the size of modulus. The function uses the maximum salt size that will fit in the encoded block.

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	hashAlg is not a supported hash algorithm

```
535 static CRYPT_RESULT
536 PssEncode    (
537     UINT32       eOutSize,       // IN: size of the encode data buffer
538     BYTE        *eOut,           // OUT: encoded data buffer
539     TPM_ALG_ID   hashAlg,        // IN: hash algorithm to use for the encoding
540     UINT32       hashInSize,     // IN: size of digest to encode
541     BYTE        *hashIn          // IN: the digest
542 #ifdef TEST_RSA                  //
543     ,  BYTE        *saltIn       // IN: optional parameter for testing
544 #endif // TEST_RSA               //
545 )
546 {
547     INT32               hLen = _cpri__GetDigestSize(hashAlg);
548     BYTE                salt[MAX_RSA_KEY_BYTES - 1];
549     UINT16              saltSize;
550     BYTE               *ps = salt;
551     CRYPT_RESULT        retVal;
552     UINT16              mLen;
553     CPRI_HASH_STATE     hashState;
554
555     // These are fatal errors indicating bad TPM firmware
556     pAssert(eOut != NULL && hLen > 0 && hashIn != NULL );
557
558     // Get the size of the mask
559     mLen = (UINT16)(eOutSize - hLen - 1);
560
561     // Maximum possible salt size is mask length - 1
562     saltSize = mLen - 1;
563
```

```
564         // Use the maximum salt size allowed by FIPS 186-4
565         if(saltSize > hLen)
566             saltSize = (UINT16)hLen;
567 
568     //using eOut for scratch space
569         // Set the first 8 bytes to zero
570         memset(eOut, 0, 8);
571 
572         // Get set the salt
573     #ifdef  TEST_RSA
574         if(saltIn != NULL)
575         {
576             saltSize = hLen;
577             memcpy(salt, saltIn, hLen);
578         }
579         else
580     #endif  // TEST_RSA
581             _cpri__GenerateRandom(saltSize, salt);
582 
583         // Create the hash of the pad || input hash || salt
584         _cpri__StartHash(hashAlg, FALSE, &hashState);
585         _cpri__UpdateHash(&hashState, 8, eOut);
586         _cpri__UpdateHash(&hashState, hashInSize, hashIn);
587         _cpri__UpdateHash(&hashState, saltSize, salt);
588         _cpri__CompleteHash(&hashState, hLen, &eOut[eOutSize - hLen - 1]);
589 
590         // Create a mask
591         if((retVal = _cpri__MGF1(mLen, eOut, hashAlg, hLen, &eOut[mLen])) < 0)
592         {
593             // Currently _cpri__MGF1 is not expected to return a CRYPT_RESULT error.
594             pAssert(0);
595         }
596         // Since this implementation uses key sizes that are all even multiples of
597         // 8, just need to make sure that the most significant bit is CLEAR
598         eOut[0] &= 0x7f;
599 
600         // Before we mess up the eOut value, set the last byte to 0xbc
601         eOut[eOutSize - 1] = 0xbc;
602 
603         // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
604         eOut = &eOut[mLen - saltSize - 1];
605         *eOut++ ^= 0x01;
606 
607         // XOR the salt data into the buffer
608         for(; saltSize > 0; saltSize--)
609             *eOut++ ^= *ps++;
610 
611         // and we are done
612         return CRYPT_SUCCESS;
613     }
```

B.12.1.3.10. PssDecode()

This function checks that the PSS encoded block was built from the provided digest. If the check is successful, CRYPT_SUCCESS is returned. Any other value indicates an error.

This implementation of PSS decoding is intended for the reference TPM implementation and is not at all generalized. It is used to check signatures over hashes and assumptions are made about the sizes of values. Those assumptions are enforce by this implementation. This implementation does allow for a variable size salt value to have been used by the creator of the signature.

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_SCHEME	*hashAlg* is not a supported hash algorithm
CRYPT_FAIL	decode operation failed

```
614 static CRYPT_RESULT
615 PssDecode(
616     TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use for the encoding
617     UINT32           dInSize,       // IN: size of the digest to compare
618     BYTE            *dIn,           // In: the digest to compare
619     UINT32           eInSize,       // IN: size of the encoded data
620     BYTE            *eIn,           // IN: the encoded data
621     UINT32           saltSize       // IN: the expected size of the salt
622     )
623 {
624     INT32            hLen = _cpri__GetDigestSize(hashAlg);
625     BYTE             mask[MAX_RSA_KEY_BYTES];
626     BYTE            *pm = mask;
627     BYTE             pad[8] = {0};
628     UINT32           i;
629     UINT32           mLen;
630     BOOL             fail = FALSE;
631     CRYPT_RESULT     retVal;
632     CPRI_HASH_STATE  hashState;
633
634     // These errors are indicative of failures due to programmer error
635     pAssert(dIn != NULL && eIn != NULL);
636
637     // check the hash scheme
638     if(hLen == 0)
639         return CRYPT_SCHEME;
640
641     // most significant bit must be zero
642     fail = ((eIn[0] & 0x80) != 0);
643
644     // last byte must be 0xbc
645     fail |= (eIn[eInSize - 1] != 0xbc);
646
647     // Use the hLen bytes at the end of the buffer to generate a mask
648     // Doesn't start at the end which is a flag byte
649     mLen = eInSize - hLen - 1;
650     if((retVal = _cpri__MGF1(mLen, mask, hashAlg, hLen, &eIn[mLen])) < 0)
651         return retVal;
652     if(retVal == 0)
653         return CRYPT_FAIL;
654
655     // Clear the MSO of the mask to make it consistent with the encoding.
656     mask[0] &= 0x7F;
657
658     // XOR the data into the mask to recover the salt. This sequence
659     // advances eIn so that it will end up pointing to the seed data
660     // which is the hash of the signature data
661     for(i = mLen; i > 0; i--)
662         *pm++ ^= *eIn++;
663
664     // Find the first byte of 0x01 after a string of all 0x00
665     for(pm = mask, i = mLen; i > 0; i--)
666     {
667         if(*pm == 0x01)
668             break;
669         else
670             fail |= (*pm++ != 0);
671     }
```

```
672         fail |= (i == 0);
673
674         // if we have failed, will continue using the entire mask as the salt value so
675         // that the timing attacks will not disclose anything (I don't think that this
676         // is a problem for TPM applications but, usually, we don't fail so this
677         // doesn't cost anything).
678         if(fail)
679         {
680             i = mLen;
681             pm = mask;
682         }
683         else
684         {
685             pm++;
686             i--;
687         }
688         // If the salt size was provided, then the recovered size must match
689         fail |= (saltSize != 0 && i != saltSize);
690
691         // i contains the salt size and pm points to the salt. Going to use the input
692         // hash and the seed to recreate the hash in the lower portion of eIn.
693         _cpri__StartHash(hashAlg, FALSE, &hashState);
694
695         // add the pad of 8 zeros
696         _cpri__UpdateHash(&hashState, 8, pad);
697
698         // add the provided digest value
699         _cpri__UpdateHash(&hashState, dInSize, dIn);
700
701         // and the salt
702         _cpri__UpdateHash(&hashState, i, pm);
703
704         // get the result
705         retVal = _cpri__CompleteHash(&hashState, MAX_DIGEST_SIZE, mask);
706
707         // retVal will be the size of the digest or zero. If not equal to the indicated
708         // digest size, then the signature doesn't match
709         fail |= (retVal != hLen);
710         fail |= (memcmp(mask, eIn, hLen) != 0);
711         if(fail)
712             return CRYPT_FAIL;
713         else
714             return CRYPT_SUCCESS;
715     }
```

B.12.1.3.11. PKSC1v1_5SignEncode()

Encode a message using PKCS1v1().5 method.

Return Value	Meaning
CRYPT_SUCCESS	encode complete
CRYPT_SCHEME	*hashAlg* is not a supported hash algorithm
CRYPT_PARAMETER	*eOutSize* is not large enough or *hInSize* does not match the digest size of *hashAlg*

```
716     static CRYPT_RESULT
717     RSASSA_Encode(
718         UINT32           eOutSize,      // IN: the size of the resulting block
719         BYTE            *eOut,          // OUT: the encoded block
720         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for PKSC1v1_5
721         UINT32           hInSize,       // IN: size of hash to be signed
722         BYTE            *hIn            // IN: hash buffer
```

```
723         )
724     {
725         BYTE            *der;
726         INT32            derSize = _cpri__GetHashDER(hashAlg, &der);
727         INT32            fillSize;
728 
729         pAssert(eOut != NULL && hIn != NULL);
730 
731         // Can't use this scheme if the algorithm doesn't have a DER string defined.
732         if(derSize == 0 )
733             return CRYPT_SCHEME;
734 
735         // If the digest size of 'hashAl' doesn't match the input digest size, then
736         // the DER will misidentify the digest so return an error
737         if((unsigned)_cpri__GetDigestSize(hashAlg) != hInSize)
738             return CRYPT_PARAMETER;
739 
740         fillSize = eOutSize - derSize - hInSize - 3;
741 
742         // Make sure that this combination will fit in the provided space
743         if(fillSize < 8)
744             return CRYPT_PARAMETER;
745         // Start filling
746         *eOut++ = 0; // initial byte of zero
747         *eOut++ = 1; // byte of 0x01
748         for(; fillSize > 0; fillSize--)
749             *eOut++ = 0xff; // bunch of 0xff
750         *eOut++ = 0; // another 0
751         for(; derSize > 0; derSize--)
752             *eOut++ = *der++;    // copy the DER
753         for(; hInSize > 0; hInSize--)
754             *eOut++ = *hIn++;    // copy the hash
755         return CRYPT_SUCCESS;
756     }
```

B.12.1.3.12. RSASSA_Decode()

This function performs the RSASSA decoding of a signature.

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decode unsuccessful
CRYPT_SCHEME	*haslAlg* is not supported

```
757     static CRYPT_RESULT
758     RSASSA_Decode(
759         TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use for the encoding
760         UINT32           hInSize,       // IN: size of the digest to compare
761         BYTE            *hIn,           // In: the digest to compare
762         UINT32           eInSize,       // IN: size of the encoded data
763         BYTE            *eIn            // IN: the encoded data
764         )
765     {
766         BOOL             fail = FALSE;
767         BYTE            *der;
768         INT32            derSize = _cpri__GetHashDER(hashAlg, &der);
769         INT32            hashSize = _cpri__GetDigestSize(hashAlg);
770         INT32            fillSize;
771 
772         pAssert(hIn != NULL && eIn != NULL);
773 
774         // Can't use this scheme if the algorithm doesn't have a DER string
```

```
775         // defined or if the provided hash isn't the right size
776         if(derSize == 0 || (unsigned)hashSize != hInSize)
777             return CRYPT_SCHEME;
778 
779         // Make sure that this combination will fit in the provided space
780         // Since no data movement takes place, can just walk though this
781         // and accept nearly random values. This can only be called from
782         // _cpri__ValidateSignature() so eInSize is known to be in range.
783         fillSize = eInSize - derSize - hashSize - 3;
784 
785         // Start checking
786         fail |= (*eIn++ != 0); // initial byte of zero
787         fail |= (*eIn++ != 1); // byte of 0x01
788         for(; fillSize > 0; fillSize--)
789             fail |= (*eIn++ != 0xff); // bunch of 0xff
790         fail |= (*eIn++ != 0); // another 0
791         for(; derSize > 0; derSize--)
792             fail |= (*eIn++ != *der++); // match the DER
793         for(; hInSize > 0; hInSize--)
794             fail |= (*eIn++ != *hIn++); // match the hash
795         if(fail)
796             return CRYPT_FAIL;
797         return CRYPT_SUCCESS;
798     }
```

B.12.1.4. Externally Accessible Functions

B.12.1.4.1. _cpri__RsaStartup()

Function that is called to initialize the hash service. In this implementation, this function does nothing but it is called by the CryptUtilStartup() function and must be present.

```
799     LIB_EXPORT BOOL
800     _cpri__RsaStartup(
801         void
802         )
803     {
804         return TRUE;
805     }
```

B.12.1.4.2. _cpri__EncryptRSA()

This is the entry point for encryption using RSA. Encryption is use of the public exponent. The padding parameter determines what padding will be used.

The *cOutSize* parameter must be at least as large as the size of the key.

If the padding is RSA_PAD_NONE, *dIn* is treaded as a number. It must be lower in value than the key modulus.

NOTE: If *dIn* has fewer bytes than *cOut*, then we don't add low-order zeros to *dIn* to make it the size of the RSA key for the call to RSAEP. This is because the high order bytes of *dIn* might have a numeric value that is greater than the value of the key modulus. If this had low-order zeros added, it would have a numeric value larger than the modulus even though it started out with a lower numeric value.

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

```
806  LIB_EXPORT CRYPT_RESULT
807  _cpri__EncryptRSA(
808      UINT32          *cOutSize,      // OUT: the size of the encrypted data
809      BYTE            *cOut,          // OUT: the encrypted data
810      RSA_KEY         *key,           // IN: the key to use for encryption
811      TPM_ALG_ID       padType,       // IN: the type of padding
812      UINT32           dInSize,       // IN: the amount of data to encrypt
813      BYTE            *dIn,           // IN: the data to encrypt
814      TPM_ALG_ID       hashAlg,       // IN: in case this is needed
815      const char      *label          // IN: in case it is needed
816      )
817  {
818      CRYPT_RESULT    retVal = CRYPT_SUCCESS;
819
820      pAssert(cOutSize != NULL);
821
822      // All encryption schemes return the same size of data
823      if(*cOutSize < key->publicKey->size)
824          return CRYPT_PARAMETER;
825      *cOutSize = key->publicKey->size;
826
827      switch (padType)
828      {
829      case TPM_ALG_NULL:  // 'raw' encryption
830          {
831              // dIn can have more bytes than cOut as long as the extra bytes
832              // are zero
833              for(; dInSize > *cOutSize; dInSize--)
834              {
835                  if(*dIn++ != 0)
836                      return CRYPT_PARAMETER;
837
838              }
839              // If dIn is smaller than cOut, fill cOut with zeros
840              if(dInSize < *cOutSize)
841                  memset(cOut, 0, *cOutSize - dInSize);
842
843              // Copy the rest of the value
844              memcpy(&cOut[*cOutSize-dInSize], dIn, dInSize);
845              // If the size of dIn is the same as cOut dIn could be larger than
846              // the modulus. If it is, then RSAEP() will catch it.
847          }
848          break;
849      case TPM_ALG_RSAES:
850          retVal = RSAES_PKSC1v1_5Encode(*cOutSize, cOut, dInSize, dIn);
851          break;
852      case TPM_ALG_OAEP:
853          retVal = OaepEncode(*cOutSize, cOut, hashAlg, label, dInSize, dIn
854  #ifdef  TEST_RSA
855                              ,NULL
856  #endif
857                             );
858          break;
```

```
859         default:
860             return CRYPT_SCHEME;
861         }
862         // All the schemes that do padding will come here for the encryption step
863         // Check that the Encoding worked
864         if(retVal != CRYPT_SUCCESS)
865             return retVal;
866
867         // Padding OK so do the encryption
868         return RSAEP(*cOutSize, cOut, key);
869     }
```

B.12.1.4.3. _cpri__DecryptRSA()

This is the entry point for decryption using RSA. Decryption is use of the private exponent. The **padType** parameter determines what padding was used.

Return Value	Meaning
CRYPT_SUCCESS	successful completion
CRYPT_PARAMETER	*cInSize* is not the same as the size of the public modulus of *key*; or numeric value of the encrypted data is greater than the modulus
CRYPT_FAIL	*dOutSize* is not large enough for the result
CRYPT_SCHEME	*padType* is not supported

```
870     LIB_EXPORT CRYPT_RESULT
871     _cpri__DecryptRSA(
872         UINT32          *dOutSize,      // OUT: the size of the decrypted data
873         BYTE            *dOut,          // OUT: the decrypted data
874         RSA_KEY         *key,           // IN: the key to use for decryption
875         TPM_ALG_ID       padType,       // IN: the type of padding
876         UINT32           cInSize,       // IN: the amount of data to decrypt
877         BYTE            *cIn,           // IN: the data to decrypt
878         TPM_ALG_ID       hashAlg,       // IN: in case this is needed for the scheme
879         const char      *label          // IN: in case it is needed for the scheme
880         )
881     {
882         CRYPT_RESULT    retVal;
883
884         // Make sure that the necessary parameters are provided
885         pAssert(cIn != NULL && dOut != NULL && dOutSize != NULL && key != NULL);
886
887         // Size is checked to make sure that the decryption works properly
888         if(cInSize != key->publicKey->size)
889             return CRYPT_PARAMETER;
890
891         // For others that do padding, do the decryption in place and then
892         // go handle the decoding.
893         if((retVal = RSADP(cInSize, cIn, key)) != CRYPT_SUCCESS)
894             return retVal;      // Decryption failed
895
896         // Remove padding
897         switch (padType)
898         {
899         case TPM_ALG_NULL:
900             if(*dOutSize < key->publicKey->size)
901                 return CRYPT_FAIL;
902             *dOutSize = key->publicKey->size;
903             memcpy(dOut, cIn, *dOutSize);
904             return CRYPT_SUCCESS;
905         case TPM_ALG_RSAES:
906             return RSAES_Decode(dOutSize, dOut, cInSize, cIn);
```

```
907             break;
908         case TPM_ALG_OAEP:
909             return OaepDecode(dOutSize, dOut, hashAlg, label, cInSize, cIn);
910             break;
911         default:
912             return CRYPT_SCHEME;
913             break;
914         }
915     }
```

B.12.1.4.4. _cpri__SignRSA()

This function is used to generate an RSA signature of the type indicated in *scheme*.

Return Value	Meaning
CRYPT_SUCCESS	sign operation completed normally
CRYPT_SCHEME	*scheme* or *hashAlg* are not supported
CRYPT_PARAMETER	*hInSize* does not match *hashAlg* (for RSASSA)

```
916     LIB_EXPORT CRYPT_RESULT
917     _cpri__SignRSA(
918         UINT32          *sigOutSize,    // OUT: size of signature
919         BYTE            *sigOut,        // OUT: signature
920         RSA_KEY         *key,           // IN: key to use
921         TPM_ALG_ID       scheme,        // IN: the scheme to use
922         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for PKSC1v1_5
923         UINT32           hInSize,       // IN: size of digest to be signed
924         BYTE            *hIn            // IN: digest buffer
925         )
926     {
927         CRYPT_RESULT    retVal;
928
929         // Parameter checks
930         pAssert(sigOutSize != NULL && sigOut != NULL && key != NULL && hIn != NULL);
931
932         // For all signatures the size is the size of the key modulus
933         *sigOutSize = key->publicKey->size;
934         switch (scheme)
935         {
936         case TPM_ALG_NULL:
937             *sigOutSize = 0;
938             return CRYPT_SUCCESS;
939         case TPM_ALG_RSAPSS:
940             // PssEncode can return CRYPT_PARAMETER
941             retVal = PssEncode(*sigOutSize, sigOut, hashAlg, hInSize, hIn
942     #ifdef  TEST_RSA
943                                , NULL
944     #endif
945                               );
946             break;
947         case TPM_ALG_RSASSA:
948             // RSASSA_Encode can return CRYPT_PARAMETER or CRYPT_SCHEME
949             retVal = RSASSA_Encode(*sigOutSize, sigOut, hashAlg, hInSize, hIn);
950             break;
951         default:
952             return CRYPT_SCHEME;
953         }
954         if(retVal != CRYPT_SUCCESS)
955             return retVal;
956         // Do the encryption using the private key
957         // RSADP can return CRYPT_PARAMETR
958         return RSADP(*sigOutSize,sigOut, key);
```



```
959 }
```

B.12.1.4.5. _cpri__ValidateSignatureRSA()

This function is used to validate an RSA signature. If the signature is valid CRYPT_SUCCESS is returned. If the signature is not valid, CRYPT_FAIL is returned. Other return codes indicate either parameter problems or fatal errors.

Return Value	Meaning
CRYPT_SUCCESS	the signature checks
CRYPT_FAIL	the signature does not check
CRYPT_SCHEME	unsupported scheme or hash algorithm

```
960 LIB_EXPORT CRYPT_RESULT
961 _cpri__ValidateSignatureRSA(
962     RSA_KEY         *key,           // IN: key to use
963     TPM_ALG_ID       scheme,        // IN: the scheme to use
964     TPM_ALG_ID       hashAlg,       // IN: hash algorithm
965     UINT32           hInSize,       // IN: size of digest to be checked
966     BYTE            *hIn,           // IN: digest buffer
967     UINT32           sigInSize,     // IN: size of signature
968     BYTE            *sigIn,         // IN: signature
969     UINT16           saltSize       // IN: salt size for PSS
970     )
971 {
972     CRYPT_RESULT     retVal;
973
974     // Fatal programming errors
975     pAssert(key != NULL && sigIn != NULL && hIn != NULL);
976
977     // Errors that might be caused by calling parameters
978     if(sigInSize != key->publicKey->size)
979         return CRYPT_FAIL;
980     // Decrypt the block
981     if((retVal = RSAEP(sigInSize, sigIn, key)) != CRYPT_SUCCESS)
982         return CRYPT_FAIL;
983     switch (scheme)
984     {
985     case TPM_ALG_NULL:
986         return CRYPT_SCHEME;
987         break;
988     case TPM_ALG_RSAPSS:
989         return PssDecode(hashAlg, hInSize, hIn, sigInSize, sigIn, saltSize);
990         break;
991     case TPM_ALG_RSASSA:
992         return RSASSA_Decode(hashAlg, hInSize, hIn, sigInSize, sigIn);
993         break;
994     default:
995         break;
996     }
997     return CRYPT_SCHEME;
998 }
999 #ifndef RSA_KEY_SIEVE
```

B.12.1.4.6. _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 canceled

```
1000  LIB_EXPORT CRYPT_RESULT
1001  _cpri__GenerateKeyRSA(
1002      TPM2B           *n,             // OUT: The public modulu
1003      TPM2B           *p,             // OUT: One of the prime factors of n
1004      UINT16           keySizeInBits, // IN: Size of the public modulus in bit
1005      UINT32           e,             // IN: The public exponent
1006      TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use in the key
1007                                      //     generation proce
1008      TPM2B           *seed,          // IN: the seed to use
1009      const char      *label,         // IN: A label for the generation process.
1010      TPM2B           *extra,         // IN: Party 1 data for the KDF
1011      UINT32          *counter        // IN/OUT: Counter value to allow KFD iteration
1012                                      //     to be propagated across multiple routine
1013      )
1014  {
1015      UINT32           lLen;          // length of the label
1016                                      // (counting the terminating 0);
1017      UINT16           digestSize = _cpri__GetDigestSize(hashAlg);
1018
1019      TPM2B_HASH_BLOCK    oPadKey;
1020
1021      UINT32          outer;
1022      UINT32          inner;
1023      BYTE            swapped[4];
1024
1025      CRYPT_RESULT    retVal;
1026      int             i, fill;
1027      const static char     defaultLabel[] = "RSA key";
1028      BYTE            *pb;
1029
1030      CPRI_HASH_STATE  h1;            // contains the hash of the
1031                                      //   HMAC key w/ iPad
1032      CPRI_HASH_STATE  h2;            // contains the hash of the
1033                                      //   HMAC key w/ oPad
1034      CPRI_HASH_STATE  h;             // the working hash context
1035
1036      BIGNUM          *bnP;
1037      BIGNUM          *bnQ;
1038      BIGNUM          *bnT;
1039      BIGNUM          *bnE;
1040      BIGNUM          *bnN;
1041      BN_CTX          *context;
1042      UINT32           rem;
1043
1044      // Make sure that hashAlg is valid hash
1045      pAssert(digestSize != 0);
1046
1047      // if present, use externally provided counter
1048      if(counter != NULL)
1049          outer = *counter;
1050      else
1051          outer = 1;
1052
1053      // Validate exponent
1054      UINT32_TO_BYTE_ARRAY(e, swapped);
1055
1056      // Need to check that the exponent is prime and not less than 3
1057      if( e != 0 && (e < 3  || !_math__IsPrime(e)))
```

```
1058            return CRYPT_FAIL;
1059
1060        // Get structures for the big number representations
1061        context = BN_CTX_new();
1062        if(context == NULL)
1063            FAIL(FATAL_ERROR_ALLOCATION);
1064        BN_CTX_start(context);
1065        bnP = BN_CTX_get(context);
1066        bnQ = BN_CTX_get(context);
1067        bnT = BN_CTX_get(context);
1068        bnE = BN_CTX_get(context);
1069        bnN = BN_CTX_get(context);
1070        if(bnN == NULL)
1071            FAIL(FATAL_ERROR_INTERNAL);
1072
1073        // Set Q to zero. This is used as a flag. The prime is computed in P. When a
1074        // new prime is found, Q is checked to see if it is zero.  If so, P is copied
1075        // to Q and a new P is found.  When both P and Q are non-zero, the modulus and
1076        // private exponent are computed and a trial encryption/decryption is
1077        // performed.  If the encrypt/decrypt fails, assume that at least one of the
1078        // primes is composite. Since we don't know which one, set Q to zero and start
1079        // over and find a new pair of primes.
1080        BN_zero(bnQ);
1081
1082        // Need to have some label
1083        if(label == NULL)
1084            label = (const char *)&defaultLabel;
1085        // Get the label size
1086        for(lLen = 0; label[lLen++] != 0;);
1087
1088        // Start the hash using the seed and get the intermediate hash value
1089        _cpri__StartHMAC(hashAlg, FALSE, &h1, seed->size, seed->buffer, &oPadKey.b);
1090        _cpri__StartHash(hashAlg, FALSE, &h2);
1091        _cpri__UpdateHash(&h2, oPadKey.b.size, oPadKey.b.buffer);
1092
1093        n->size = (keySizeInBits +7)/8;
1094        pAssert(n->size <= MAX_RSA_KEY_BYTES);
1095        p->size = n->size / 2;
1096        if(e == 0)
1097            e = RSA_DEFAULT_PUBLIC_EXPONENT;
1098
1099        BN_set_word(bnE, e);
1100
1101        // The first test will increment the counter from zero.
1102        for(outer += 1; outer != 0; outer++)
1103        {
1104            if(_plat__IsCanceled())
1105            {
1106                retVal = CRYPT_CANCEL;
1107                goto Cleanup;
1108            }
1109
1110            // Need to fill in the candidate with the hash
1111            fill = digestSize;
1112            pb = p->buffer;
1113
1114            // Reset the inner counter
1115            inner = 0;
1116            for(i = p->size; i > 0; i -= digestSize)
1117            {
1118                inner++;
1119                // Initialize the HMAC with saved state
1120                _cpri__CopyHashState(&h, &h1);
1121
1122                // Hash the inner counter (the one that changes on each HMAC iteration)
1123                UINT32_TO_BYTE_ARRAY(inner, swapped);
```

```
1124                _cpri__UpdateHash(&h, 4, swapped);
1125                _cpri__UpdateHash(&h, lLen, (BYTE *)label);
1126
1127                // Is there any party 1 data
1128                if(extra != NULL)
1129                    _cpri__UpdateHash(&h, extra->size, extra->buffer);
1130
1131                // Include the outer counter (the one that changes on each prime
1132                // prime candidate generation
1133                UINT32_TO_BYTE_ARRAY(outer, swapped);
1134                _cpri__UpdateHash(&h, 4, swapped);
1135                _cpri__UpdateHash(&h, 2, (BYTE *)&keySizeInBits);
1136                if(i < fill)
1137                    fill = i;
1138                _cpri__CompleteHash(&h, fill, pb);
1139
1140                // Restart the oPad hash
1141                _cpri__CopyHashState(&h, &h2);
1142
1143                // Add the last hashed data
1144                _cpri__UpdateHash(&h, fill, pb);
1145
1146                // gives a completed HMAC
1147                _cpri__CompleteHash(&h, fill, pb);
1148                pb += fill;
1149            }
1150            // Set the Most significant 2 bits and the low bit of the candidate
1151            p->buffer[0] |= 0xC0;
1152            p->buffer[p->size - 1] |= 1;
1153
1154            // Convert the candidate to a BN
1155            BN_bin2bn(p->buffer, p->size, bnP);
1156
1157            // If this is the second prime, make sure that it differs from the
1158            // first prime by at least 2^100
1159            if(!BN_is_zero(bnQ))
1160            {
1161                // bnQ is non-zero if we already found it
1162                if(BN_ucmp(bnP, bnQ) < 0)
1163                    BN_sub(bnT, bnQ, bnP);
1164                else
1165                    BN_sub(bnT, bnP, bnQ);
1166                if(BN_num_bits(bnT) < 100)  // Difference has to be at least 100 bits
1167                    continue;
1168            }
1169            // Make sure that the prime candidate (p) is not divisible by the exponent
1170            // and that (p-1) is not divisible by the exponent
1171            // Get the remainder after dividing by the modulus
1172            rem = BN_mod_word(bnP, e);
1173            if(rem == 0) // evenly divisible so add two keeping the number odd and
1174                // making sure that 1 != p mod e
1175                BN_add_word(bnP, 2);
1176            else if(rem == 1) // leaves a remainder of 1 so subtract two keeping the
1177                // number odd and making (e-1) = p mod e
1178                BN_sub_word(bnP, 2);
1179
1180            // Have a candidate, check for primality
1181            if((retVal = (CRYPT_RESULT)BN_is_prime_ex(bnP,
1182                        BN_prime_checks, NULL, NULL)) < 0)
1183                FAIL(FATAL_ERROR_INTERNAL);
1184
1185            if(retVal != 1)
1186                continue;
1187
1188            // Found a prime, is this the first or second.
1189            if(BN_is_zero(bnQ))
```

```
1190            {
1191                // copy p to q and compute another prime in p
1192                BN_copy(bnQ, bnP);
1193                continue;
1194            }
1195            //Form the public modulus
1196            BN_mul(bnN, bnP, bnQ, context);
1197            if(BN_num_bits(bnN) != keySizeInBits)
1198                FAIL(FATAL_ERROR_INTERNAL);
1199
1200            // Save the public modulus
1201            BnTo2B(n, bnN, n->size);  // Will pad the buffer to the correct size
1202            pAssert((n->buffer[0] & 0x80) != 0);
1203
1204            // And one prime
1205            BnTo2B(p, bnP, p->size);
1206            pAssert((p->buffer[0] & 0x80) != 0);
1207
1208            // Finish by making sure that we can form the modular inverse of PHI
1209            // with respect to the public exponent
1210            // Compute PHI = (p - 1)(q - 1) = n - p - q + 1
1211            // Make sure that we can form the modular inverse
1212            BN_sub(bnT, bnN, bnP);
1213            BN_sub(bnT, bnT, bnQ);
1214            BN_add_word(bnT, 1);
1215
1216            // find d such that (Phi * d) mod e ==1
1217            // If there isn't then we are broken because we took the step
1218            // of making sure that the prime != 1 mod e so the modular inverse
1219            // must exist
1220            if(BN_mod_inverse(bnT, bnE, bnT, context) == NULL || BN_is_zero(bnT))
1221                FAIL(FATAL_ERROR_INTERNAL);
1222
1223            // And, finally, do a trial encryption decryption
1224            {
1225                TPM2B_TYPE(RSA_KEY, MAX_RSA_KEY_BYTES);
1226                TPM2B_RSA_KEY        r;
1227                r.t.size = sizeof(n->size);
1228
1229                // If we are using a seed, then results must be reproducible on each
1230                // call. Otherwise, just get a random number
1231                if(seed == NULL)
1232                    _cpri__GenerateRandom(n->size, r.t.buffer);
1233                else
1234                {
1235                    // this this version does not have a deterministic RNG, XOR the
1236                    // public key and private exponent to get a deterministic value
1237                    // for testing.
1238                    int         i;
1239
1240                    // Generate a random-ish number starting with the public modulus
1241                    // XORed with the MSO of the seed
1242                    for(i = 0; i < n->size; i++)
1243                        r.t.buffer[i] = n->buffer[i] ^ seed->buffer[0];
1244                }
1245                // Make sure that the number is smaller than the public modulus
1246                r.t.buffer[0] &= 0x7F;
1247                      // Convert
1248                if(   BN_bin2bn(r.t.buffer, r.t.size, bnP) == NULL
1249                      // Encrypt with the public exponent
1250                   || BN_mod_exp(bnQ, bnP, bnE, bnN, context) != 1
1251                      // Decrypt with the private exponent
1252                   || BN_mod_exp(bnQ, bnQ, bnT, bnN, context) != 1)
1253                    FAIL(FATAL_ERROR_INTERNAL);
1254                // If the starting and ending values are not the same, start over )-;
1255                if(BN_ucmp(bnP, bnQ) != 0)
```

```
1256                {
1257                    BN_zero(bnQ);
1258                    continue;
1259                }
1260            }
1261            retVal = CRYPT_SUCCESS;
1262            goto Cleanup;
1263        }
1264        retVal = CRYPT_FAIL;
1265
1266    Cleanup:
1267        // Close out the hash sessions
1268        _cpri__CompleteHash(&h2, 0, NULL);
1269        _cpri__CompleteHash(&h1, 0, NULL);
1270
1271        // Free up allocated BN values
1272        BN_CTX_end(context);
1273        BN_CTX_free(context);
1274        if(counter != NULL)
1275            *counter = outer;
1276        return retVal;
1277    }
1278    #endif      // RSA_KEY_SIEVE
1279    #endif // TPM_ALG_RSA
```

B.12.2. Alternative RSA Key Generation

B.12.2.1. 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.14.1 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 filed starting address is chosen at random and a sieve operation is performed on the field using small prime values. After sieving, the bits representing values that are not divisible by the small primes tested, will be checked in a pseudo-random order until a prime is found.

The size of the sieve field is tunable as is the value indicating the number of primes that should be checked. As the size of the prime increases, the density of primes is reduced so the size of the sieve field should be increased to improve the probability that the field will contain at least one prime. In addition, as the sieve field increases the number of small primes that should be checked increases. Eliminating a number from consideration by using division is considerably faster than eliminating the number with a Miller-Rabin test.

B.12.2.2. RSAKeySieve.h

This header file is used to for parameterization of the Sieve and RNG used by the RSA module

```
1    #ifndef     RSA_H
2    #define     RSA_H
```

This value is used to set the size of the table that is searched by the prime iterator. This is used during the generation of different primes. The smaller tables are used when generating smaller primes.

```
3    extern const UINT16   primeTableBytes;
```

The following define determines how large the prime number difference table will be defined. The value of 13 will allocate the maximum size table which allows generation of the first 6542 primes which is all the primes less than 2^16.

```
#define PRIME_DIFF_TABLE_512_BYTE_PAGES     13
```

This set of macros used the value above to set the table size.

```
#ifndef PRIME_DIFF_TABLE_512_BYTE_PAGES
#   define PRIME_DIFF_TABLE_512_BYTE_PAGES   4
#endif
#ifdef PRIME_DIFF_TABLE_512_BYTE_PAGES
#   if PRIME_DIFF_TABLE_512_BYTE_PAGES > 12
#       define PRIME_DIFF_TABLE_BYTES 6542
#   else
#       if  PRIME_DIFF_TABLE_512_BYTE_PAGES <= 0
#           define PRIME_DIFF_TABLE_BYTES 512
#       else
#           define PRIME_DIFF_TABLE_BYTES (PRIME_DIFF_TABLE_512_BYTE_PAGES * 512)
#       endif
#   endif
#endif
extern const BYTE primeDiffTable [PRIME_DIFF_TABLE_BYTES];
```

This determines the number of bits in the sieve field This must be a power of two.

```
#define FIELD_POWER     14  // This is the only value in this group that should be
                            // changed
#define FIELD_BITS      (1 << FIELD_POWER)
#define MAX_FIELD_SIZE      ((FIELD_BITS / 8) + 1)
```

This is the pre-sieved table. It already has the bits for multiples of 3, 5, and 7 cleared.

```
#define SEED_VALUES_SIZE          105
const extern BYTE                 seedValues[SEED_VALUES_SIZE];
```

This allows determination of the number of bits that are set in a byte without having to count them individually.

```
const extern BYTE                 bitsInByte[256];
```

This is the iterator structure for accessing the compressed prime number table. The expectation is that values will need to be accesses sequentially. This tries to save some data access.

```
typedef struct {
    UINT32      lastPrime;
    UINT32      index;
    UINT32      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;
```

```
50    #define NUM_PRIMES          2047
51    extern const __int16        primes[NUM_PRIMES];
52    #else
53    #define defaultFieldSize    MAX_FIELD_SIZE
54    #endif
55    #endif
```

B.12.2.3. RSAKeySieve.c

B.12.2.3.1. Includes and defines

```
#include    "OsslCryptoEngine.h"
#ifdef    TPM_ALG_RSA
```

This file produces no code unless the compile switch is set to cause it to generate code.

```
#ifdef      RSA_KEY_SIEVE               //%
#include    "RsaKeySieve.h"
```

This next line will show up in the header file for this code. It will make the local functions public when debugging.

```
//%#ifdef   RSA_DEBUG
```

B.12.2.3.2. Bit Manipulation Functions

B.12.2.3.2.1. Introduction

These functions operate on a bit array. A bit array is an array of bytes with the 0th byte being the byte with the lowest memory address. Within the byte, bit 0 is the least significant bit.

B.12.2.3.2.2. ClearBit()

This function will CLEAR a bit in a bit array.

```
void
ClearBit(
    unsigned char   *a,             // IN: A pointer to an array of byte
    int              i              // IN: the number of the bit to CLEAR
    )
{
    a[i >> 3] &= 0xff ^ (1 << (i & 7));
}
```

B.12.2.3.2.3. SetBit()

Function to SET a bit in a bit array.

```
void
SetBit(
    unsigned char   *a,             // IN: A pointer to an array of byte
    int              i              // IN: the number of the bit to SET
    )
{
    a[i >> 3] |= (1 << (i & 7));
}
```

B.12.2.3.2.4. IsBitSet()

Function to test if a bit in a bit array is SET.

Return Value	Meaning
0	bit is CLEAR
1	bit is SET

```
22  UINT32
23  IsBitSet(
24      unsigned char   *a,             // IN: A pointer to an array of byte
25      int              i              // IN: the number of the bit to test
26      )
27  {
28      return ((a[i >> 3] & (1 << (i & 7))) != 0);
29  }
```

B.12.2.3.2.5. BitsInArry()

This function counts the number of bits set in an array of bytes.

```
30  int
31  BitsInArray(
32      unsigned char   *a,             // IN: A pointer to an array of byte
33      int              i              // IN: the number of bytes to sum
34      )
35  {
36      int     j = 0;
37      for(; i ; i--)
38          j += bitsInByte[*a++];
39      return j;
40  }
```

B.12.2.3.2.6. FindNthSetBit()

This function finds the nth SET bit in a bit array. The caller should check that the offset of the returned value is not out of range. If called when the array does not have n bits set, it will return a fatal error

```
41  UINT32
42  FindNthSetBit(
43      const UINT16     aSize,         // IN: the size of the array to check
44      const BYTE      *a,             // IN: the array to check
45      const UINT32     n              // IN, the number of the SET bit
46      )
47  {
48      UINT32       i;
49      const BYTE  *pA = a;
50      UINT32       retValue;
51      BYTE         sel;
52
53      (aSize);
54
55      //find the bit
56      for(i = 0; i < n; i += bitsInByte[*pA++]);
57
58      // The chosen bit is in the byte that was just accessed
59      // Compute the offset to the start of that byte
60      pA--;
61      retValue = (UINT32)(pA - a) * 8;
62
63      // Subtract the bits in the last byte added.
64      i -= bitsInByte[*pA];
65
66      // Now process the byte, one bit at a time.
```



```
67      for(sel = *pA; sel != 0 ; sel = sel >> 1)
68      {
69          if(sel & 1)
70          {
71              i += 1;
72              if(i == n)
73                  return retValue;
74          }
75          retValue += 1;
76      }
77      FAIL(FATAL_ERROR_INTERNAL);
78  }
```

B.12.2.3.3. Miscellaneous Functions

B.12.2.3.3.1. 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.

```
79  void
80  RandomForRsa(
81      KDFa_CONTEXT    *ktx,           // IN: a context for the KDF
82      const char      *label,         // IN: a use qualifying label
83      TPM2B           *p              // OUT: the pseudo random result
84      )
85  {
86      INT16                   i;
87      UINT32                  inner;
88      BYTE                    swapped[4];
89      UINT16                  fill;
90      BYTE                    *pb;
91      UINT16                  lLen = 0;
92      UINT16                  digestSize = _cpri__GetDigestSize(ktx->hashAlg);
93      CPRI_HASH_STATE         h;      // the working hash context
94
95      if(label != NULL)
96          for(lLen = 0; label[lLen++];);
97      fill = digestSize;
98      pb = p->buffer;
99      inner = 0;
100     *(ktx->outer) += 1;
101     for(i = p->size; i > 0; i -= digestSize)
102     {
103         inner++;
104
105         // Initialize the HMAC with saved state
106         _cpri__CopyHashState(&h, &(ktx->iPadCtx));
107
108         // Hash the inner counter (the one that changes on each HMAC iteration)
109         UINT32_TO_BYTE_ARRAY(inner, swapped);
110         _cpri__UpdateHash(&h, 4, swapped);
111         if(lLen != 0)
112             _cpri__UpdateHash(&h, lLen, (BYTE *)label);
113
114         // Is there any party 1 data
115         if(ktx->extra != NULL)
116             _cpri__UpdateHash(&h, ktx->extra->size, ktx->extra->buffer);
117
```

```
118             // Include the outer counter (the one that changes on each prime
119             // prime candidate generation
120             UINT32_TO_BYTE_ARRAY(*(ktx->outer), swapped);
121             _cpri__UpdateHash(&h, 4, swapped);
122             _cpri__UpdateHash(&h, 2, (BYTE *)&ktx->keySizeInBits);
123             if(i < fill)
124                 fill = i;
125             _cpri__CompleteHash(&h, fill, pb);
126
127             // Restart the oPad hash
128             _cpri__CopyHashState(&h, &(ktx->oPadCtx));
129
130             // Add the last hashed data
131             _cpri__UpdateHash(&h, fill, pb);
132
133             // gives a completed HMAC
134             _cpri__CompleteHash(&h, fill, pb);
135             pb += fill;
136         }
137         return;
138     }
```

B.12.2.3.3.2. MillerRabinRounds()

Function returns the number of Miller-Rabin rounds necessary to give an error probability equal to the security strength of the prime. These values are from FIPS 186-3.

```
139     UINT32
140     MillerRabinRounds(
141         UINT32          bits            // IN: Number of bits in the RSA prime
142         )
143     {
144         if(bits < 511) return 8;    // don't really expect this
145         if(bits < 1536) return 5;   // for 512 and 1K primes
146         return 4;                   // for 3K public modulus and greater
147     }
```

B.12.2.3.3.3. 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.

Return Value	Meaning
TRUE	probably prime
FALSE	composite

```
148     BOOL
149     MillerRabin(
150         BIGNUM          *bnW,
151         int              iterations,
152         KDFa_CONTEXT    *ktx,
153         BN_CTX          *context
154         )
155     {
156         BIGNUM      *bnWm1;
157         BIGNUM      *bnM;
158         BIGNUM      *bnB;
159         BIGNUM      *bnZ;
```

```
160         BOOL         ret = FALSE;   // Assumed composite for easy exit
161         TPM2B_TYPE(MAX_PRIME, MAX_RSA_KEY_BYTES/2);
162         TPM2B_MAX_PRIME   b;
163         int          a;
164         int          j;
165         int          wLen;
166         int          i;
167
168         pAssert(BN_is_bit_set(bnW, 0));
169         INSTRUMENT_INC(MillerRabinTrials);  // Instrumentation
170
171         BN_CTX_start(context);
172         bnWm1 = BN_CTX_get(context);
173         bnB = BN_CTX_get(context);
174         bnZ = BN_CTX_get(context);
175         bnM = BN_CTX_get(context);
176         if(bnM == NULL)
177             FAIL(FATAL_ERROR_ALLOCATION);
178
179     // Let a be the largest integer such that 2^a divides w1.
180         BN_copy(bnWm1, bnW);
181         BN_sub_word(bnWm1, 1);
182         // Since w is odd (w-1) is even so start at bit number 1 rather than 0
183         for(a = 1; !BN_is_bit_set(bnWm1, a); a++);
184
185     // 2. m = (w1) / 2^a
186         BN_rshift(bnM, bnWm1, a);
187
188     // 3. wlen = len (w).
189         wLen = BN_num_bits(bnW);
190         pAssert((wLen & 7) == 0);
191
192         // Set the size for the random number
193         b.b.size = (UINT16)(wLen + 7)/8;
194
195     // 4. For i = 1 to iterations do
196         for(i = 0; i < iterations ; i++)
197         {
198
199     // 4.1 Obtain a string b of wlen bits from an RBG.
200     step4point1:
201             // In the reference implementation, wLen is always a multiple of 8
202             if(ktx != NULL)
203                 RandomForRsa(ktx, "Miller-Rabin witness", &b.b);
204             else
205                 _cpri__GenerateRandom(b.t.size, b.t.buffer);
206
207             if(BN_bin2bn(b.t.buffer, b.t.size, bnB) == NULL)
208                 FAIL(FATAL_ERROR_ALLOCATION);
209
210     // 4.2 If ((b  1) or (b  w1)), then go to step 4.1.
211             if(BN_is_zero(bnB))
212                 goto step4point1;
213             if(BN_is_one(bnB))
214                 goto step4point1;
215             if(BN_ucmp(bnB, bnWm1) >= 0)
216                 goto step4point1;
217
218     // 4.3 z = b^m mod w.
219             if(BN_mod_exp(bnZ, bnB, bnM, bnW, context) != 1)
220                 FAIL(FATAL_ERROR_ALLOCATION);
221
222     // 4.4 If ((z = 1) or (z = w  1)), then go to step 4.7.
223             if(BN_is_one(bnZ) || BN_ucmp(bnZ, bnWm1) == 0)
224                 goto step4point7;
225
```

```
226     // 4.5 For j = 1 to a  1 do.
227             for(j = 1; j < a; j++)
228             {
229     // 4.5.1 z = z^2 mod w.
230                 if(BN_mod_mul(bnZ, bnZ, bnZ, bnW, context) != 1)
231                     FAIL(FATAL_ERROR_ALLOCATION);
232
233     // 4.5.2 If (z = w1), then go to step 4.7.
234                 if(BN_ucmp(bnZ, bnWm1) == 0)
235                     goto step4point7;
236
237     // 4.5.3 If (z = 1), then go to step 4.6.
238                 if(BN_is_one(bnZ))
239                     goto step4point6;
240             }
241     // 4.6 Return COMPOSITE.
242     step4point6:
243             if(i > 9)
244                 INSTRUMENT_INC(failedAtIteration[9]);
245             else
246                 INSTRUMENT_INC(failedAtIteration[i]);
247             goto end;
248
249     // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
250     step4point7:
251             continue;
252         }
253     // 5. Return PROBABLY PRIME
254         ret = TRUE;
255
256     end:
257         BN_CTX_end(context);
258         return ret;
259     }
```

B.12.2.3.3.4. NextPrime()

This function is used to access the next prime number in the sequence of primes. It requires a pre-initialized iterator.

```
260     UINT32
261     NextPrime(
262         PRIME_ITERATOR  *iter
263         )
264     {
265         if(iter->index >= iter->final)
266             return (iter->lastPrime = 0);
267         return (iter->lastPrime += primeDiffTable[iter->index++]);
268     }
```

B.12.2.3.3.5. 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.

```
269     UINT32
270     AdjustNumberOfPrimes(
271         UINT32           p
272         )
273     {
274         p = ((p + 511) / 512) * 512;
```

```
275         if(p == 0 || p > PRIME_DIFF_TABLE_BYTES)
276             p = PRIME_DIFF_TABLE_BYTES;
277         return p;
278     }
```

B.12.2.3.3.6. 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.

```
279     UINT32
280     PrimeInit(
281         UINT32           first,         // IN: the initial prime
282         PRIME_ITERATOR  *iter,          // IN/OUT: the iterator structure
283         UINT32           primes         // IN: the table length
284         )
285     {
286
287         iter->lastPrime = 1;
288         iter->index = 0;
289         iter->final = AdjustNumberOfPrimes(primes);
290         while(iter->lastPrime < first)
291             NextPrime(iter);
292         return iter->lastPrime;
293     }
```

B.12.2.3.3.7. SetDefaultNumberOfPrimes()

This macro sets the default number of primes to the indicated value.

```
294     //%#define SetDefaultNumberOfPrimes(p) (primeTableBytes = AdjustNumberOfPrimes(p))
```

B.12.2.3.3.8. IsPrimeWord()

Checks to see if a UINT32 is prime

Return Value	Meaning
TRUE	number is prime
FAIL	number is not prime

```
295     BOOL
296     IsPrimeWord(
297         UINT32           p              // IN: number to test
298         )
299     {
300     #if defined RSA_KEY_SIEVE && (PRIME_DIFF_TABLE_BYTES >= 6542)
301
302         UINT32      test;
303         UINT32      index;
304         UINT32      stop;
305
306         if((p & 1) == 0)
307             return FALSE;
308         if(p == 1 || p == 3)
309             return TRUE;
310
311         // Get a high value for the stopping point
312         for(index = p, stop = 0; index; index >>= 2)
```

```
313             stop = (stop << 1) + 1;
314         stop++;
315
316         // If the full prime difference value table is present, can check here
317
318         test = 3;
319         for(index = 1; index < PRIME_DIFF_TABLE_BYTES; index += 1)
320         {
321             if((p % test) == 0)
322                 return (p == test);
323             if(test > stop)
324                 return TRUE;
325             test += primeDiffTable[index];
326         }
327         return TRUE;
328
329     #else
330
331         BYTE        b[4];
332         if(p == RSA_DEFAULT_PUBLIC_EXPONENT || p == 1 || p == 3 )
333             return TRUE;
334         if((p & 1) == 0)
335             return FALSE;
336         UINT32_TO_BYTE_ARRAY(p,b);
337         return _math__IsPrime(p);
338     #endif
339     }
340     typedef struct  {
341         UINT16      prime;
342         UINT16      count;
343     } SIEVE_MARKS;
344     const SIEVE_MARKS sieveMarks[5] = {
345         {31, 7}, {73, 5}, {241, 4}, {1621, 3}, {UINT16_MAX, 2}};
```

B.12.2.3.3.9. 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).

```
346     UINT32
347     PrimeSieve(
348         BIGNUM      *bnN,       // IN/OUT: number to sieve
349         UINT32       fieldSize, // IN: size of the field area in bytes
350         BYTE        *field,     // IN: field
351         UINT32       primes     // IN: the number of primes to use
352         )
353     {
354         UINT32          i;
355         UINT32          j;
356         UINT32          fieldBits = fieldSize * 8;
357         UINT32          r;
358         const BYTE      *p1;
359         BYTE            *p2;
360         PRIME_ITERATOR   iter;
361         UINT32          adjust;
362         UINT32          mark = 0;
363         UINT32          count = sieveMarks[0].count;
364         UINT32          stop = sieveMarks[0].prime;
365         UINT32          composite;
366
367     //    UINT64          test;      //DEBUG
```

```
368
369         pAssert(field != NULL && bnN != NULL);
370         // Need to have a field that has a size of 2^n + 1 bytes
371         pAssert(BitsInArray((BYTE *)&fieldSize, 2) == 2);
372
373         primes = AdjustNumberOfPrimes(primes);
374
375         // If the remainder is odd, then subtracting the value
376         // will give an even number, but we want an odd number,
377         // so subtract the 105+rem. Otherwise, just subtract
378         // the even remainder.
379         adjust = BN_mod_word(bnN,105);
380         if(adjust & 1)
381             adjust += 105;
382
383         // seed the field
384         // This starts the pointer at the nearest byte to the input value
385         p1 = &seedValues[adjust/16];
386
387         // Reduce the number of bytes to transfer by the amount skipped
388         j = sizeof(seedValues) - adjust/16;
389         adjust = adjust % 16;
390         BN_sub_word(bnN, adjust);
391         adjust >>= 1;
392
393         // This offsets the field
394         p2 = field;
395         for(i = fieldSize; i > 0; i--)
396         {
397             *p2++ = *p1++;
398             if(--j == 0)
399             {
400                 j = sizeof(seedValues);
401                 p1 = seedValues;
402             }
403         }
404         // Mask the first bits in the field and the last byte in order to eliminate
405         // bytes not in the field from consideration.
406         field[0] &= 0xff << adjust;
407         field[fieldSize-1] &= 0xff >> (8 - adjust);
408
409         // Cycle through the primes, clearing bits
410         // Have already done 3, 5, and 7
411         PrimeInit(7, &iter, primes);
412
413         // Get the next N primes where N is determined by the mark in the sieveMarks
414         while((composite = NextPrime(&iter)) != 0)
415         {
416             UINT32  pList[8];
417             UINT32   next = 0;
418             i = count;
419             pList[i--] = composite;
420             for(; i > 0; i--)
421             {
422                 next = NextPrime(&iter);
423                 pList[i] = next;
424                 if(next != 0)
425                     composite *= next;
426             }
427             composite = BN_mod_word(bnN, composite);
428             for(i = count; i > 0; i--)
429             {
430                 next = pList[i];
431                 if(next == 0)
432                     goto done;
433                 r = composite % next;
```

```
434             if(r & 1)           j = (next - r)/2;
435             else if(r == 0)     j = 0;
436             else                j = next - r/2;
437             for(; j < fieldBits; j += next)
438                 ClearBit(field, j);
439         }
440         if(next >= stop)
441         {
442             mark++;
443             count = sieveMarks[mark].count;
444             stop = sieveMarks[mark].prime;
445         }
446     }
447 done:
448     INSTRUMENT_INC(totalFieldsSieved);
449     i = BitsInArray(field, fieldSize);
450     if(i == 0) INSTRUMENT_INC(emptyFieldsSieved);
451     return i;
452 }
```

B.12.2.3.3.10. 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.

```
453 BOOL
454 PrimeSelectWithSieve(
455     BIGNUM          *bnP,               // IN/OUT: The candidate to filter
456     KDFa_CONTEXT    *ktx,               // IN: KDFa iterator structure
457     UINT32           e,                 // IN: the exponent
458     BN_CTX          *context            // IN: the big number context to play in
459 #ifdef RSA_DEBUG                        //%
460    ,UINT16           fieldSize,         // IN: number of bytes in the field, as
461                                         //     determined by the caller
462     UINT16           primes             // IN: number of primes to use.
463 #endif                                  //%
464 )
465 {
466     BYTE             field[MAX_FIELD_SIZE];
467     UINT32           first;
468     UINT32           ones;
469     INT32            chosen;
470     UINT32           rounds = MillerRabinRounds(BN_num_bits(bnP));
471 #ifndef RSA_DEBUG
472     UINT32           primes;
473     UINT32           fieldSize;
474     // Adjust the field size and prime table list to fit the size of the prime
475     // being tested.
476     primes = BN_num_bits(bnP);
477     if(primes <= 512)
478     {
479         primes = AdjustNumberOfPrimes(2048);
480         fieldSize = 65;
481     }
482     else if(primes <= 1024)
483     {
484         primes = AdjustNumberOfPrimes(4096);
485         fieldSize = 129;
486     }
```

```
487         else
488         {
489             primes = AdjustNumberOfPrimes(0);  // Set to the maximum
490             fieldSize = MAX_FIELD_SIZE;
491         }
492         if(fieldSize > MAX_FIELD_SIZE)
493             fieldSize = MAX_FIELD_SIZE;
494     #endif
495
496         // Save the low-order word to use as a search generator and make sure that
497         // it has some interesting range to it
498         first = bnP->d[0] | 0x80000000;
499
500         // Align to field boundary
501         bnP->d[0] &= ~((UINT32)(fieldSize-3));
502         pAssert(BN_is_bit_set(bnP, 0));
503         bnP->d[0] &= (UINT32_MAX << (FIELD_POWER + 1)) + 1;
504         ones = PrimeSieve(bnP, fieldSize, field, primes);
505     #ifdef  RSA_FILTER_DEBUG
506         pAssert(ones == BitsInArray(field, defaultFieldSize));
507     #endif
508         for(; ones > 0; ones--)
509         {
510     #ifdef  RSA_FILTER_DEBUG
511             if(ones != BitsInArray(field, defaultFieldSize))
512                 FAIL(FATAL_ERROR_INTERNAL);
513     #endif
514             // Decide which bit to look at and find its offset
515             if(ones == 1)
516                 ones = ones;
517             chosen = FindNthSetBit(defaultFieldSize, field,((first % ones) + 1));
518             if(chosen >= ((defaultFieldSize) * 8))
519                 FAIL(FATAL_ERROR_INTERNAL);
520
521             // Set this as the trial prime
522             BN_add_word(bnP, chosen * 2);
523
524             // Use MR to see if this is prime
525             if(MillerRabin(bnP, rounds, ktx, context))
526             {
527                 // Final check is to make sure that 0 != (p-1) mod e
528                 // This is the same as -1 != p mod e ; or
529                 // (e - 1) != p mod e
530                 if((e <= 3) || (BN_mod_word(bnP, e) != (e-1)))
531                     return TRUE;
532             }
533             // Back out the bit number
534             BN_sub_word(bnP, chosen * 2);
535
536             // Clear the bit just tested
537             ClearBit(field, chosen);
538     }
539         // Ran out of bits and couldn't find a prime in this field
540         INSTRUMENT_INC(noPrimeFields);
541         return FALSE;
542     }
```

B.12.2.3.3.11. AdjustPrimeCandiate()

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.

```
543  void
544  AdjustPrimeCandidate(
545      BYTE            *a,
546      UINT16           len
547      )
548  {
549      UINT16  highBytes;
550
551      highBytes = BYTE_ARRAY_TO_UINT16(a);
552      // This is fixed point arithmetic on 16-bit values
553      highBytes = ((UINT32)highBytes * (UINT32)0x4AFB) >> 16;
554      highBytes += 0xB505;
555      UINT16_TO_BYTE_ARRAY(highBytes, a);
556      a[len-1] |= 1;
557  }
```

B.12.2.3.3.12. GeneratateRamdomPrime()

```
558  void
559  GenerateRandomPrime(
560      TPM2B   *p,
561      BN_CTX  *ctx
562  #ifdef  RSA_DEBUG       //%
563     ,UINT16   field,
564      UINT16   primes
565  #endif                  //%
566      )
567  {
568      BIGNUM  *bnP;
569      BN_CTX  *context;
570
571      if(ctx == NULL) context = BN_CTX_new();
572      else context = ctx;
573      if(context == NULL)
574          FAIL(FATAL_ERROR_ALLOCATION);
575      BN_CTX_start(context);
576      bnP = BN_CTX_get(context);
577
578      while(TRUE)
579      {
580          _cpri__GenerateRandom(p->size, p->buffer);
581          p->buffer[p->size-1] |= 1;
582          p->buffer[0] |= 0x80;
583          BN_bin2bn(p->buffer, p->size, bnP);
584  #ifdef  RSA_DEBUG
585          if(PrimeSelectWithSieve(bnP, NULL, 0, context, field, primes))
586  #else
587          if(PrimeSelectWithSieve(bnP, NULL, 0, context))
588  #endif
589              break;
590      }
591      BnTo2B(p, bnP, (UINT16)BN_num_bytes(bnP));
592      BN_CTX_end(context);
593      if(ctx == NULL)
594          BN_CTX_free(context);
595      return;
596  }
597  KDFa_CONTEXT *
598  KDFaContextStart(
```

```
599         KDFa_CONTEXT    *ktx,           // IN/OUT: the context structure to initialize
600         TPM2B           *seed,          // IN: the seed for the digest proce
601         TPM_ALG_ID       hashAlg,       // IN: the hash algorithm
602         TPM2B           *extra,         // IN: the extra data
603         UINT32          *outer,         // IN: the outer iteration counter
604         UINT16           keySizeInBit
605         )
606     {
607         UINT16               digestSize = _cpri__GetDigestSize(hashAlg);
608         TPM2B_HASH_BLOCK     oPadKey;
609
610         if(seed == NULL)
611             return NULL;
612
613         pAssert(ktx != NULL && outer != NULL && digestSize != 0);
614
615        // Start the hash using the seed and get the intermediate hash value
616        _cpri__StartHMAC(hashAlg, FALSE, &(ktx->iPadCtx), seed->size, seed->buffer,
617                         &oPadKey.b);
618        _cpri__StartHash(hashAlg, FALSE, &(ktx->oPadCtx));
619        _cpri__UpdateHash(&(ktx->oPadCtx), oPadKey.b.size, oPadKey.b.buffer);
620        ktx->extra = extra;
621        ktx->hashAlg = hashAlg;
622        ktx->outer = outer;
623        ktx->keySizeInBits = keySizeInBits;
624        return ktx;
625    }
626    void
627    KDFaContextEnd(
628        KDFa_CONTEXT    *ktx            // IN/OUT: the context structure to close
629        )
630    {
631        if(ktx != NULL)
632        {
633            // Close out the hash sessions
634            _cpri__CompleteHash(&(ktx->iPadCtx), 0, NULL);
635            _cpri__CompleteHash(&(ktx->oPadCtx), 0, NULL);
636        }
637    }
638    //%#endif
```

B.12.2.3.4. Public Function

B.12.2.3.4.1. Introduction

This is the external entry for this replacement function. All this file provides is the substitute function to generate an RSA key. If the compiler settings are set appropriately, this this function will be used instead of the similarly named function in CpriRSA.c.

B.12.2.3.4.2. _cpri__GenerateKeyRSA()

Generate an RSA key from a provided seed

Return Value	Meaning
CRYPT_FAIL	exponent is not prime or is less than 3; or could not find a prime using the provided parameters
CRYPT_CANCEL	operation was canceled

```
639    LIB_EXPORT CRYPT_RESULT
640    _cpri__GenerateKeyRSA(
```

```
641         TPM2B           *n,             // OUT: The public modulus
642         TPM2B           *p,             // OUT: One of the prime factors of n
643         UINT16           keySizeInBits, // IN: Size of the public modulus in bits
644         UINT32           e,             // IN: The public exponent
645         TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use in the key
646                                         //     generation process
647         TPM2B           *seed,          // IN: the seed to use
648         const char      *label,         // IN: A label for the generation process.
649         TPM2B           *extra,         // IN: Party 1 data for the KDF
650         UINT32          *counter        // IN/OUT: Counter value to allow KDF
651                                         //         iteration to be propagated across
652                                         //         multiple routines
653     #ifdef  RSA_DEBUG                   //%
654        ,UINT16           primes,       // IN: number of primes to test
655         UINT16           fieldSize     // IN: the field size to use
656     #endif                              //%
657         )
658     {
659         CRYPT_RESULT         retVal;
660         UINT32               myCounter = 0;
661         UINT32              *pCtr = (counter == NULL) ? &myCounter : counter;
662
663         KDFa_CONTEXT         ktx;
664         KDFa_CONTEXT        *ktxPtr;
665         UINT32               i;
666         BIGNUM              *bnP;
667         BIGNUM              *bnQ;
668         BIGNUM              *bnT;
669         BIGNUM              *bnE;
670         BIGNUM              *bnN;
671         BN_CTX              *context;
672
673         // Make sure that the required pointers are provided
674         pAssert(n != NULL && p != NULL);
675
676         // If the seed is provided, then use KDFa for generation of the 'random'
677         // values
678         ktxPtr = KDFaContextStart(&ktx, seed, hashAlg, extra, pCtr, keySizeInBits);
679
680         n->size = keySizeInBits/8;
681         p->size = n->size / 2;
682
683         // Validate exponent
684         if(e == 0 || e == RSA_DEFAULT_PUBLIC_EXPONENT)
685             e = RSA_DEFAULT_PUBLIC_EXPONENT;
686         else
687             if(!IsPrimeWord(e))
688                 return CRYPT_FAIL;
689
690         // Get structures for the big number representations
691         context = BN_CTX_new();
692         BN_CTX_start(context);
693         bnP = BN_CTX_get(context);
694         bnQ = BN_CTX_get(context);
695         bnT = BN_CTX_get(context);
696         bnE = BN_CTX_get(context);
697         bnN = BN_CTX_get(context);
698         if(bnN == NULL)
699             FAIL(FATAL_ERROR_INTERNAL);
700
701         // Set Q to zero. This is used as a flag. The prime is computed in P. When a
702         // new prime is found, Q is checked to see if it is zero.  If so, P is copied
703         // to Q and a new P is found.  When both P and Q are non-zero, the modulus and
704         // private exponent are computed and a trial encryption/decryption is
705         // performed.  If the encrypt/decrypt fails, assume that at least one of the
706         // primes is composite. Since we don't know which one, set Q to zero and start
```

```
707         // over and find a new pair of primes.
708         BN_zero(bnQ);
709         BN_set_word(bnE, e);
710 
711         // Each call to generate a random value will increment ktx.outer
712         // it doesn't matter if ktx.outer wraps. This lets the caller
713         // use the initial value of the counter for additional entropy.
714         for(i = 0; i < UINT32_MAX; i++)
715         {
716             if(_plat__IsCanceled())
717             {
718                 retVal = CRYPT_CANCEL;
719                 goto end;
720             }
721             // Get a random prime candidate.
722             if(seed == NULL)
723                 _cpri__GenerateRandom(p->size, p->buffer);
724             else
725                 RandomForRsa(&ktx, label, p);
726             AdjustPrimeCandidate(p->buffer, p->size);
727 
728             // Convert the candidate to a BN
729             if(BN_bin2bn(p->buffer, p->size, bnP) == NULL)
730                 FAIL(FATAL_ERROR_INTERNAL);
731             // If this is the second prime, make sure that it differs from the
732             // first prime by at least 2^100. Since BIGNUMS use words, the check
733             // below will make sure they are different by at least 128 bits
734             if(!BN_is_zero(bnQ))
735             { // bnQ is non-zero, we have a first value
736                 UINT32       *pP = (UINT32 *)(&bnP->d[4]);
737                 UINT32       *pQ = (UINT32 *)(&bnQ->d[4]);
738                 INT32        k = ((INT32)bnP->top) - 4;
739                 for(;k > 0; k--)
740                     if(*pP++ != *pQ++)
741                         break;
742                 // Didn't find any difference so go get a new value
743                 if(k == 0)
744                     continue;
745             }
746             // If PrimeSelectWithSieve   returns success, bnP is a prime,
747 #ifdef  RSA_DEBUG
748             if(!PrimeSelectWithSieve(bnP, ktxPtr, e, context, fieldSize, primes))
749 #else
750             if(!PrimeSelectWithSieve(bnP, ktxPtr, e, context))
751 #endif
752                 continue;   // If not, get another
753 
754             // Found a prime, is this the first or second.
755             if(BN_is_zero(bnQ))
756             {   // copy p to q and compute another prime in p
757                 BN_copy(bnQ, bnP);
758                 continue;
759             }
760             //Form the public modulus
761            if(   BN_mul(bnN, bnP, bnQ, context) != 1
762               || BN_num_bits(bnN) != keySizeInBits)
763                 FAIL(FATAL_ERROR_INTERNAL);
764             // Save the public modulus
765             BnTo2B(n, bnN, n->size);
766             // And one prime
767             BnTo2B(p, bnP, p->size);
768 
769 #ifdef EXTENDED_CHECKS
770             // Finish by making sure that we can form the modular inverse of PHI
771             // with respect to the public exponent
772             // Compute PHI = (p - 1)(q - 1) = n - p - q + 1
```

```
773             // Make sure that we can form the modular inverse
774             if(   BN_sub(bnT, bnN, bnP) != 1
775                || BN_sub(bnT, bnT, bnQ) != 1
776                || BN_add_word(bnT, 1) != 1)
777                 FAIL(FATAL_ERROR_INTERNAL);
778 
779             // find d such that (Phi * d) mod e ==1
780             // If there isn't then we are broken because we took the step
781             // of making sure that the prime != 1 mod e so the modular inverse
782             // must exist
783             if(   BN_mod_inverse(bnT, bnE, bnT, context) == NULL
784                || BN_is_zero(bnT))
785                 FAIL(FATAL_ERROR_INTERNAL);
786 
787             // And, finally, do a trial encryption decryption
788             {
789                 TPM2B_TYPE(RSA_KEY, MAX_RSA_KEY_BYTES);
790                 TPM2B_RSA_KEY        r;
791                 r.t.size = sizeof(r.t.buffer);
792                 // If we are using a seed, then results must be reproducible on each
793                 // call. Otherwise, just get a random number
794                 if(seed == NULL)
795                     _cpri__GenerateRandom(keySizeInBits/8, r.t.buffer);
796                 else
797                     RandomForRsa(&ktx, label, &r.b);
798 
799                 // Make sure that the number is smaller than the public modulus
800                 r.t.buffer[0] &= 0x7F;
801                       // Convert
802                 if(   BN_bin2bn(r.t.buffer, r.t.size, bnP) == NULL
803                       // Encrypt with the public exponent
804                    || BN_mod_exp(bnQ, bnP, bnE, bnN, context) != 1
805                       // Decrypt with the private exponent
806                    || BN_mod_exp(bnQ, bnQ, bnT, bnN, context) != 1)
807                     FAIL(FATAL_ERROR_INTERNAL);
808                 // If the starting and ending values are not the same, start over )-;
809                 if(BN_ucmp(bnP, bnQ) != 0)
810                 {
811                     BN_zero(bnQ);
812                     continue;
813                 }
814             }
815 #endif // EXTENDED_CHECKS
816             retVal = CRYPT_SUCCESS;
817             goto end;
818         }
819         retVal = CRYPT_FAIL;
820 
821 end:
822         KDFaContextEnd(&ktx);
823 
824         // Free up allocated BN values
825         BN_CTX_end(context);
826         BN_CTX_free(context);
827         return retVal;
828 }
829 #else
830 static void noFuntion(
831     void
832     )
833 {
834     pAssert(1);
835 }
836 #endif              //%
837 #endif // TPM_ALG_RSA
```

B.12.2.4. RSAData.c

```
#include "OsslCryptoEngine.h"
#ifdef  RSA_KEY_SIEVE
#include "RsaKeySieve.h"
#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,
    0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
    0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
    0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
    0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
    0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
    0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0x69, 0xc9, 0x34,
    0x25, 0xda, 0x22, 0x65, 0x99, 0xb4, 0x4c, 0x1b,
    0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
    0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6,
    0x25, 0xd3, 0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2,
    0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
    0xd1};
const BYTE bitsInByte[256] = {
    0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08
};
```

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.

```
56   const UINT16      primeTableBytes = PRIME_DIFF_TABLE_BYTES;
57   #if PRIME_DIFF_TABLE_BYTES > 0
58   const BYTE primeDiffTable [PRIME_DIFF_TABLE_BYTES] = {
59      0x02,0x02,0x02,0x04,0x02,0x04,0x02,0x04,0x06,0x02,0x06,0x04,0x02,0x04,0x06,0x06,
60      0x02,0x06,0x04,0x02,0x06,0x04,0x06,0x08,0x04,0x02,0x04,0x02,0x04,0x0E,0x04,0x06,
61      0x02,0x0A,0x02,0x06,0x06,0x04,0x06,0x06,0x02,0x0A,0x02,0x04,0x02,0x0C,0x0C,0x04,
62      0x02,0x04,0x06,0x02,0x0A,0x06,0x06,0x06,0x02,0x06,0x04,0x02,0x0A,0x0E,0x04,0x02,
63      0x04,0x0E,0x06,0x0A,0x02,0x04,0x06,0x08,0x06,0x06,0x04,0x06,0x08,0x04,0x08,0x0A,
64      0x02,0x0A,0x02,0x06,0x04,0x06,0x08,0x04,0x02,0x04,0x0C,0x08,0x04,0x08,0x04,0x06,
65      0x0C,0x02,0x12,0x06,0x0A,0x06,0x06,0x02,0x06,0x0A,0x06,0x06,0x02,0x06,0x06,0x04,
66      0x02,0x0C,0x0A,0x02,0x04,0x06,0x06,0x02,0x0C,0x04,0x06,0x08,0x0A,0x08,0x0A,0x08,
67      0x06,0x06,0x04,0x08,0x06,0x04,0x08,0x04,0x0E,0x0A,0x0C,0x02,0x0A,0x02,0x04,0x02,
68      0x0A,0x0E,0x04,0x02,0x04,0x0E,0x04,0x02,0x04,0x14,0x04,0x08,0x0A,0x08,0x04,0x06,
69      0x06,0x0E,0x04,0x06,0x06,0x08,0x06,0x0C,0x04,0x06,0x02,0x0A,0x02,0x06,0x0A,0x02,
70      0x0A,0x02,0x06,0x12,0x04,0x02,0x04,0x06,0x06,0x08,0x06,0x06,0x16,0x02,0x0A,0x08,
71      0x0A,0x06,0x06,0x08,0x0C,0x04,0x06,0x06,0x02,0x06,0x0C,0x0A,0x12,0x02,0x04,0x06,
72      0x02,0x06,0x04,0x02,0x04,0x0C,0x02,0x06,0x22,0x06,0x06,0x08,0x12,0x0A,0x0E,0x04,
73      0x02,0x04,0x06,0x08,0x04,0x02,0x06,0x0C,0x0A,0x02,0x04,0x02,0x04,0x06,0x0C,0x0C,
74      0x08,0x0C,0x06,0x04,0x06,0x08,0x04,0x08,0x04,0x0E,0x04,0x06,0x02,0x04,0x06,0x02
75   #endif
76   // 256
77   #if PRIME_DIFF_TABLE_BYTES > 256
78     ,0x06,0x0A,0x14,0x06,0x04,0x02,0x18,0x04,0x02,0x0A,0x0C,0x02,0x0A,0x08,0x06,0x06,
79      0x06,0x12,0x06,0x04,0x02,0x0C,0x0A,0x0C,0x08,0x10,0x0E,0x06,0x04,0x02,0x04,0x02,
80      0x0A,0x0C,0x06,0x06,0x12,0x02,0x10,0x02,0x16,0x06,0x08,0x06,0x04,0x02,0x04,0x08,
81      0x06,0x0A,0x02,0x0A,0x0E,0x0A,0x06,0x0C,0x02,0x04,0x02,0x0A,0x0C,0x02,0x10,0x02,
82      0x06,0x04,0x02,0x0A,0x08,0x12,0x18,0x04,0x06,0x08,0x10,0x02,0x04,0x08,0x10,0x02,
83      0x04,0x08,0x06,0x06,0x04,0x0C,0x02,0x16,0x06,0x02,0x06,0x04,0x06,0x0E,0x06,0x04,
84      0x02,0x06,0x04,0x06,0x0C,0x06,0x06,0x0E,0x04,0x06,0x0C,0x08,0x06,0x04,0x1A,0x12,
85      0x0A,0x08,0x04,0x06,0x02,0x06,0x16,0x0C,0x02,0x10,0x08,0x04,0x0C,0x0E,0x0A,0x02,
86      0x04,0x08,0x06,0x06,0x04,0x02,0x04,0x06,0x08,0x04,0x02,0x06,0x0A,0x02,0x0A,0x08,
87      0x04,0x0E,0x0A,0x0C,0x02,0x06,0x04,0x02,0x10,0x0E,0x04,0x06,0x08,0x06,0x04,0x12,
88      0x08,0x0A,0x06,0x06,0x08,0x0A,0x0C,0x0E,0x04,0x06,0x06,0x02,0x1C,0x02,0x0A,0x08,
89      0x04,0x0E,0x04,0x08,0x0C,0x06,0x0C,0x04,0x06,0x14,0x0A,0x02,0x10,0x1A,0x04,0x02,
90      0x0C,0x06,0x04,0x0C,0x06,0x08,0x04,0x08,0x16,0x02,0x04,0x02,0x0C,0x1C,0x02,0x06,
91      0x06,0x06,0x04,0x06,0x02,0x0C,0x04,0x0C,0x02,0x0A,0x02,0x10,0x02,0x10,0x06,0x14,
92      0x10,0x08,0x04,0x02,0x04,0x02,0x16,0x08,0x0C,0x06,0x0A,0x02,0x04,0x06,0x02,0x06,
93      0x0A,0x02,0x0C,0x0A,0x02,0x0A,0x0E,0x06,0x04,0x06,0x08,0x06,0x06,0x10,0x0C,0x02
94   #endif
95   // 512
96   #if PRIME_DIFF_TABLE_BYTES > 512
97     ,0x04,0x0E,0x06,0x04,0x08,0x0A,0x08,0x06,0x06,0x16,0x06,0x02,0x0A,0x0E,0x04,0x06,
98      0x12,0x02,0x0A,0x0E,0x04,0x02,0x0A,0x0E,0x04,0x08,0x12,0x04,0x06,0x02,0x04,0x06,
99      0x02,0x0C,0x04,0x14,0x16,0x0C,0x02,0x04,0x06,0x06,0x02,0x06,0x16,0x02,0x06,0x10,
100     0x06,0x0C,0x02,0x06,0x0C,0x10,0x02,0x04,0x06,0x0E,0x04,0x02,0x12,0x18,0x0A,0x06,
101     0x02,0x0A,0x02,0x0A,0x02,0x0A,0x06,0x02,0x0A,0x02,0x0A,0x06,0x08,0x1E,0x0A,0x02,
102     0x0A,0x08,0x06,0x0A,0x12,0x06,0x0C,0x0C,0x02,0x12,0x06,0x04,0x06,0x06,0x12,0x02,
103     0x0A,0x0E,0x06,0x04,0x02,0x04,0x18,0x02,0x0C,0x06,0x10,0x08,0x06,0x06,0x12,0x10,
104     0x02,0x04,0x06,0x02,0x06,0x06,0x0A,0x06,0x0C,0x0C,0x12,0x02,0x06,0x04,0x12,0x08,
105     0x18,0x04,0x02,0x04,0x06,0x02,0x0C,0x04,0x0E,0x1E,0x0A,0x06,0x0C,0x0E,0x06,0x0A,
106     0x0C,0x02,0x04,0x06,0x08,0x06,0x0A,0x02,0x04,0x0E,0x06,0x06,0x04,0x06,0x02,0x0A,
107     0x02,0x10,0x0C,0x08,0x12,0x04,0x06,0x0C,0x02,0x06,0x06,0x06,0x1C,0x06,0x0E,0x04,
108     0x08,0x0A,0x08,0x0C,0x12,0x04,0x02,0x04,0x18,0x0C,0x06,0x02,0x10,0x06,0x06,0x0E,
109     0x0A,0x0E,0x04,0x1E,0x06,0x06,0x06,0x08,0x06,0x04,0x02,0x0C,0x06,0x04,0x02,0x06,
110     0x16,0x06,0x02,0x04,0x12,0x02,0x04,0x0C,0x02,0x06,0x04,0x1A,0x06,0x06,0x04,0x08,
111     0x0A,0x20,0x10,0x02,0x06,0x04,0x02,0x04,0x02,0x0A,0x0E,0x06,0x04,0x08,0x0A,0x06,
112     0x14,0x04,0x02,0x06,0x1E,0x04,0x08,0x0A,0x06,0x06,0x08,0x06,0x0C,0x04,0x06,0x02
113  #endif
```



```
114   // 768
115   #if PRIME_DIFF_TABLE_BYTES > 768
116     ,0x06,0x04,0x06,0x02,0x0A,0x02,0x10,0x06,0x14,0x04,0x0C,0x0E,0x1C,0x06,0x14,0x04,
117      0x12,0x08,0x06,0x04,0x06,0x0E,0x06,0x06,0x0A,0x02,0x0A,0x0C,0x08,0x0A,0x02,0x0A,
118      0x08,0x0C,0x0A,0x18,0x02,0x04,0x08,0x06,0x04,0x08,0x12,0x0A,0x06,0x06,0x02,0x06,
119      0x0A,0x0C,0x02,0x0A,0x06,0x06,0x06,0x08,0x06,0x0A,0x06,0x02,0x06,0x06,0x06,0x0A,
120      0x08,0x18,0x06,0x16,0x02,0x12,0x04,0x08,0x0A,0x1E,0x08,0x12,0x04,0x02,0x0A,0x06,
121      0x02,0x06,0x04,0x12,0x08,0x0C,0x12,0x10,0x06,0x02,0x0C,0x06,0x0A,0x02,0x0A,0x02,
122      0x06,0x0A,0x0E,0x04,0x18,0x02,0x10,0x02,0x0A,0x02,0x0A,0x14,0x04,0x02,0x04,0x08,
123      0x10,0x06,0x06,0x02,0x0C,0x10,0x08,0x04,0x06,0x1E,0x02,0x0A,0x02,0x06,0x04,0x06,
124      0x06,0x08,0x06,0x04,0x0C,0x06,0x08,0x0C,0x04,0x0E,0x0C,0x0A,0x18,0x06,0x0C,0x06,
125      0x02,0x16,0x08,0x12,0x0A,0x06,0x0E,0x04,0x02,0x06,0x0A,0x08,0x06,0x04,0x06,0x1E,
126      0x0E,0x0A,0x02,0x0C,0x0A,0x02,0x10,0x02,0x12,0x18,0x12,0x06,0x10,0x12,0x06,0x02,
127      0x12,0x04,0x06,0x02,0x0A,0x08,0x0A,0x06,0x06,0x08,0x04,0x06,0x02,0x0A,0x02,0x0C,
128      0x04,0x06,0x06,0x02,0x0C,0x04,0x0E,0x12,0x04,0x06,0x14,0x04,0x08,0x06,0x04,0x08,
129      0x04,0x0E,0x06,0x04,0x0E,0x0C,0x04,0x02,0x1E,0x04,0x18,0x06,0x06,0x0C,0x0C,0x0E,
130      0x06,0x04,0x02,0x04,0x12,0x06,0x0C,0x08,0x06,0x04,0x0C,0x02,0x0C,0x1E,0x10,0x02,
131      0x06,0x16,0x0E,0x06,0x0A,0x0C,0x06,0x02,0x04,0x08,0x0A,0x06,0x06,0x18,0x0E,0x06
132   #endif
133   // 1024
134   #if PRIME_DIFF_TABLE_BYTES > 1024
135     ,0x04,0x08,0x0C,0x12,0x0A,0x02,0x0A,0x02,0x04,0x06,0x14,0x06,0x04,0x0E,0x04,0x02,
136      0x04,0x0E,0x06,0x0C,0x18,0x0A,0x06,0x08,0x0A,0x02,0x1E,0x04,0x06,0x02,0x0C,0x04,
137      0x0E,0x06,0x22,0x0C,0x08,0x06,0x0A,0x02,0x04,0x14,0x0A,0x08,0x10,0x02,0x0A,0x0E,
138      0x04,0x02,0x0C,0x06,0x10,0x06,0x08,0x04,0x08,0x04,0x06,0x08,0x06,0x06,0x0C,0x06,
139      0x04,0x06,0x06,0x08,0x12,0x04,0x14,0x04,0x0C,0x02,0x0A,0x06,0x02,0x0A,0x0C,0x02,
140      0x04,0x14,0x06,0x1E,0x06,0x04,0x08,0x0A,0x0C,0x06,0x02,0x1C,0x02,0x06,0x04,0x02,
141      0x10,0x0C,0x02,0x06,0x0A,0x08,0x18,0x0C,0x06,0x12,0x06,0x04,0x0E,0x06,0x04,0x0C,
142      0x08,0x06,0x0C,0x04,0x06,0x0C,0x06,0x0C,0x02,0x10,0x14,0x04,0x02,0x0A,0x12,0x08,
143      0x04,0x0E,0x04,0x02,0x06,0x16,0x06,0x0E,0x06,0x06,0x0A,0x06,0x02,0x0A,0x02,0x04,
144      0x02,0x16,0x02,0x04,0x06,0x06,0x0C,0x06,0x0E,0x0A,0x0C,0x06,0x08,0x04,0x24,0x0E,
145      0x0C,0x06,0x04,0x06,0x02,0x0C,0x06,0x0C,0x10,0x02,0x0A,0x08,0x16,0x02,0x0C,0x06,
146      0x04,0x06,0x12,0x02,0x0C,0x06,0x04,0x0C,0x08,0x06,0x0C,0x04,0x06,0x0C,0x06,0x02,
147      0x0C,0x0C,0x04,0x0E,0x06,0x10,0x06,0x02,0x0A,0x08,0x12,0x06,0x22,0x02,0x1C,0x02,
148      0x16,0x06,0x02,0x0A,0x0C,0x02,0x06,0x04,0x08,0x16,0x06,0x02,0x0A,0x08,0x04,0x06,
149      0x08,0x04,0x0C,0x12,0x0C,0x14,0x04,0x06,0x06,0x08,0x04,0x02,0x10,0x0C,0x02,0x0A,
150      0x08,0x0A,0x02,0x04,0x06,0x0E,0x0C,0x16,0x08,0x1C,0x02,0x04,0x14,0x04,0x02,0x04
151   #endif
152   // 1280
153   #if PRIME_DIFF_TABLE_BYTES > 1280
154     ,0x0E,0x0A,0x0C,0x02,0x0C,0x10,0x02,0x1C,0x08,0x16,0x08,0x04,0x06,0x06,0x0E,0x04,
155      0x08,0x0C,0x06,0x06,0x04,0x14,0x04,0x12,0x02,0x0C,0x06,0x04,0x06,0x0E,0x12,0x0A,
156      0x08,0x0A,0x20,0x06,0x0A,0x06,0x06,0x02,0x06,0x10,0x06,0x02,0x0C,0x06,0x1C,0x02,
157      0x0A,0x08,0x10,0x06,0x08,0x06,0x0A,0x18,0x14,0x0A,0x02,0x0A,0x02,0x0C,0x04,0x06,
158      0x14,0x04,0x02,0x0C,0x12,0x0A,0x02,0x0A,0x02,0x04,0x14,0x10,0x1A,0x04,0x08,0x06,
159      0x04,0x0C,0x06,0x08,0x0C,0x0C,0x06,0x04,0x08,0x16,0x02,0x10,0x0E,0x0A,0x06,0x0C,
160      0x0C,0x0E,0x06,0x04,0x14,0x04,0x0C,0x06,0x02,0x06,0x06,0x10,0x08,0x16,0x02,0x1C,
161      0x08,0x06,0x04,0x14,0x04,0x0C,0x18,0x14,0x04,0x08,0x0A,0x02,0x10,0x02,0x0C,0x0C,
162      0x22,0x02,0x04,0x06,0x0C,0x06,0x06,0x08,0x06,0x04,0x02,0x06,0x18,0x04,0x14,0x0A,
163      0x06,0x06,0x0E,0x04,0x06,0x06,0x02,0x0C,0x06,0x0A,0x02,0x0A,0x06,0x14,0x04,0x1A,
164      0x04,0x02,0x06,0x16,0x02,0x18,0x04,0x06,0x02,0x04,0x06,0x18,0x06,0x08,0x04,0x02,
165      0x22,0x06,0x08,0x10,0x0C,0x02,0x0A,0x02,0x0A,0x06,0x08,0x04,0x08,0x0C,0x16,0x06,
166      0x0E,0x04,0x1A,0x04,0x02,0x0C,0x0A,0x08,0x04,0x08,0x0C,0x04,0x0E,0x06,0x10,0x06,
167      0x08,0x04,0x06,0x06,0x08,0x06,0x0A,0x0C,0x02,0x06,0x06,0x10,0x08,0x06,0x06,0x0C,
168      0x0A,0x02,0x06,0x12,0x04,0x06,0x06,0x06,0x0C,0x12,0x08,0x06,0x0A,0x08,0x12,0x04,
169      0x0E,0x06,0x12,0x0A,0x08,0x0A,0x0C,0x02,0x06,0x0C,0x0C,0x24,0x04,0x06,0x08,0x04
170   #endif
171   // 1536
172   #if PRIME_DIFF_TABLE_BYTES > 1536
173     ,0x06,0x02,0x04,0x12,0x0C,0x06,0x08,0x06,0x06,0x04,0x12,0x02,0x04,0x02,0x18,0x04,
174      0x06,0x06,0x0E,0x1E,0x06,0x04,0x06,0x0C,0x06,0x14,0x04,0x08,0x04,0x08,0x06,0x06,
175      0x04,0x1E,0x02,0x0A,0x0C,0x08,0x0A,0x08,0x18,0x06,0x0C,0x04,0x0E,0x04,0x06,0x02,
176      0x1C,0x0E,0x10,0x02,0x0C,0x06,0x04,0x14,0x0A,0x06,0x06,0x06,0x08,0x0A,0x0C,0x0E,
177      0x0A,0x0E,0x10,0x0E,0x0A,0x0E,0x06,0x10,0x06,0x08,0x06,0x10,0x14,0x0A,0x02,0x06,
178      0x04,0x02,0x04,0x0C,0x02,0x0A,0x02,0x06,0x16,0x06,0x02,0x04,0x12,0x08,0x0A,0x08,
179      0x16,0x02,0x0A,0x12,0x0E,0x04,0x02,0x04,0x12,0x02,0x04,0x06,0x08,0x0A,0x02,0x1E,
```

```
    0x04,0x1E,0x02,0x0A,0x02,0x12,0x04,0x12,0x06,0x0E,0x0A,0x02,0x04,0x14,0x24,0x06,
    0x04,0x06,0x0E,0x04,0x14,0x0A,0x0E,0x16,0x06,0x02,0x1E,0x0C,0x0A,0x12,0x02,0x04,
    0x0E,0x06,0x16,0x12,0x02,0x0C,0x06,0x04,0x08,0x04,0x08,0x06,0x0A,0x02,0x0C,0x12,
    0x0A,0x0E,0x10,0x0E,0x04,0x06,0x06,0x02,0x06,0x04,0x02,0x1C,0x02,0x1C,0x06,0x02,
    0x04,0x06,0x0E,0x04,0x0C,0x0E,0x10,0x0E,0x04,0x06,0x08,0x06,0x04,0x06,0x06,0x06,
    0x08,0x04,0x08,0x04,0x0E,0x10,0x08,0x06,0x04,0x0C,0x08,0x10,0x02,0x0A,0x08,0x04,
    0x06,0x1A,0x06,0x0A,0x08,0x04,0x06,0x0C,0x0E,0x1E,0x04,0x0E,0x16,0x08,0x0C,0x04,
    0x06,0x08,0x0A,0x06,0x0E,0x0A,0x06,0x02,0x0A,0x0C,0x0C,0x0E,0x06,0x06,0x12,0x0A,
    0x06,0x08,0x12,0x04,0x06,0x02,0x06,0x0A,0x02,0x0A,0x08,0x06,0x06,0x0A,0x02,0x12
#endif
// 1792
#if PRIME_DIFF_TABLE_BYTES > 1792
  ,0x0A,0x02,0x0C,0x04,0x06,0x08,0x0A,0x0C,0x0E,0x0C,0x04,0x08,0x0A,0x06,0x06,0x14,
    0x04,0x0E,0x10,0x0E,0x0A,0x08,0x0A,0x0C,0x02,0x12,0x06,0x0C,0x0A,0x0C,0x02,0x04,
    0x02,0x0C,0x06,0x04,0x08,0x04,0x2C,0x04,0x02,0x04,0x02,0x0A,0x0C,0x06,0x06,0x0E,
    0x04,0x06,0x06,0x06,0x08,0x06,0x24,0x12,0x04,0x06,0x02,0x0C,0x06,0x06,0x06,0x04,
    0x0E,0x16,0x0C,0x02,0x12,0x0A,0x06,0x1A,0x18,0x04,0x02,0x04,0x02,0x04,0x0E,0x04,
    0x06,0x06,0x08,0x10,0x0C,0x02,0x2A,0x04,0x02,0x04,0x18,0x06,0x06,0x02,0x12,0x04,
    0x0E,0x06,0x1C,0x12,0x0E,0x06,0x0A,0x0C,0x02,0x06,0x0C,0x1E,0x06,0x04,0x06,0x06,
    0x0E,0x04,0x02,0x18,0x04,0x06,0x06,0x1A,0x0A,0x12,0x06,0x08,0x06,0x06,0x1E,0x04,
    0x0C,0x0C,0x02,0x10,0x02,0x06,0x04,0x0C,0x12,0x02,0x06,0x04,0x1A,0x0C,0x06,0x0C,
    0x04,0x18,0x18,0x0C,0x06,0x02,0x0C,0x1C,0x08,0x04,0x06,0x0C,0x02,0x12,0x06,0x04,
    0x06,0x06,0x14,0x10,0x02,0x06,0x06,0x12,0x0A,0x06,0x02,0x04,0x08,0x06,0x06,0x18,
    0x10,0x06,0x08,0x0A,0x06,0x0E,0x16,0x08,0x10,0x06,0x02,0x0C,0x04,0x02,0x16,0x08,
    0x12,0x22,0x02,0x06,0x12,0x04,0x06,0x06,0x08,0x0A,0x08,0x12,0x06,0x04,0x02,0x04,
    0x08,0x10,0x02,0x0C,0x0C,0x06,0x12,0x04,0x06,0x06,0x06,0x02,0x06,0x0C,0x0A,0x14,
    0x0C,0x12,0x04,0x06,0x02,0x10,0x02,0x0A,0x0E,0x04,0x1E,0x02,0x0A,0x0C,0x02,0x18,
    0x06,0x10,0x08,0x0A,0x02,0x0C,0x16,0x06,0x02,0x10,0x14,0x0A,0x02,0x0C,0x0C,0x00
#endif
// 2048
#if PRIME_DIFF_TABLE_BYTES > 2048
  ,0x12,0x0A,0x0C,0x06,0x02,0x0A,0x02,0x06,0x0A,0x12,0x02,0x0C,0x06,0x04,0x06,0x02,
    0x18,0x1C,0x02,0x04,0x02,0x0A,0x02,0x10,0x0C,0x08,0x16,0x02,0x06,0x04,0x02,0x0A,
    0x06,0x14,0x0C,0x0A,0x08,0x0C,0x06,0x06,0x06,0x04,0x12,0x02,0x04,0x0C,0x12,0x02,
    0x0C,0x06,0x04,0x02,0x10,0x0C,0x0C,0x0E,0x04,0x08,0x12,0x04,0x0C,0x0E,0x06,0x06,
    0x04,0x08,0x06,0x04,0x14,0x0C,0x0A,0x0E,0x04,0x02,0x10,0x02,0x0C,0x1E,0x04,0x06,
    0x18,0x14,0x18,0x0A,0x08,0x0C,0x0A,0x0C,0x06,0x0C,0x0C,0x06,0x08,0x10,0x0E,0x06,
    0x04,0x06,0x24,0x14,0x0A,0x1E,0x0C,0x02,0x04,0x02,0x1C,0x0C,0x0E,0x06,0x16,0x08,
    0x04,0x12,0x06,0x0E,0x12,0x04,0x06,0x02,0x06,0x22,0x12,0x02,0x10,0x06,0x12,0x02,
    0x18,0x04,0x02,0x06,0x0C,0x06,0x0C,0x0A,0x08,0x06,0x10,0x0C,0x08,0x0A,0x0E,0x28,
    0x06,0x02,0x06,0x04,0x0C,0x0E,0x04,0x02,0x04,0x02,0x04,0x08,0x06,0x0A,0x06,0x06,
    0x02,0x06,0x06,0x06,0x0C,0x06,0x18,0x0A,0x02,0x0A,0x06,0x0C,0x06,0x06,0x0E,0x06,
    0x06,0x34,0x14,0x06,0x0A,0x02,0x0A,0x08,0x0A,0x0C,0x0C,0x02,0x06,0x04,0x0E,0x10,
    0x08,0x0C,0x06,0x16,0x02,0x0A,0x08,0x06,0x16,0x02,0x16,0x06,0x08,0x0A,0x0C,0x0C,
    0x02,0x0A,0x06,0x0C,0x02,0x04,0x0E,0x0A,0x02,0x06,0x12,0x04,0x0C,0x08,0x12,0x0C,
    0x06,0x06,0x04,0x06,0x06,0x0E,0x04,0x02,0x0C,0x0C,0x04,0x06,0x12,0x12,0x0C,0x02,
    0x10,0x0C,0x08,0x12,0x0A,0x1A,0x04,0x06,0x08,0x06,0x06,0x04,0x02,0x0A,0x14,0x04
#endif
// 2304
#if PRIME_DIFF_TABLE_BYTES > 2304
  ,0x06,0x08,0x04,0x14,0x0A,0x02,0x22,0x02,0x04,0x18,0x02,0x0C,0x0C,0x0A,0x06,0x02,
    0x0C,0x1E,0x06,0x0C,0x10,0x0C,0x02,0x16,0x12,0x0C,0x0E,0x0A,0x02,0x0C,0x0C,0x04,
    0x02,0x04,0x06,0x0C,0x02,0x10,0x12,0x02,0x28,0x08,0x10,0x06,0x08,0x0A,0x02,0x04,
    0x12,0x08,0x0A,0x08,0x0C,0x04,0x12,0x02,0x12,0x0A,0x02,0x04,0x02,0x04,0x08,0x1C,
    0x02,0x06,0x16,0x0C,0x06,0x0E,0x12,0x04,0x06,0x08,0x06,0x06,0x0A,0x08,0x04,0x02,
    0x12,0x0A,0x06,0x14,0x16,0x08,0x06,0x1E,0x04,0x02,0x04,0x12,0x06,0x1E,0x02,0x04,
    0x08,0x06,0x04,0x06,0x0C,0x0E,0x22,0x0E,0x06,0x04,0x02,0x06,0x04,0x0E,0x04,0x02,
    0x06,0x1C,0x02,0x04,0x06,0x08,0x0A,0x02,0x0A,0x02,0x0A,0x02,0x04,0x1E,0x02,0x0C,
    0x0C,0x0A,0x12,0x0C,0x0E,0x0A,0x02,0x0C,0x06,0x0A,0x06,0x0E,0x0C,0x04,0x0E,0x04,
    0x12,0x02,0x0A,0x08,0x04,0x08,0x0A,0x0C,0x12,0x12,0x08,0x06,0x12,0x10,0x0E,0x06,
    0x06,0x0A,0x0E,0x04,0x06,0x02,0x0C,0x0C,0x04,0x06,0x06,0x0C,0x02,0x10,0x02,0x0C,
    0x06,0x04,0x0E,0x06,0x04,0x02,0x0C,0x12,0x04,0x24,0x12,0x0C,0x0C,0x02,0x04,0x02,
    0x04,0x08,0x0C,0x04,0x24,0x06,0x12,0x02,0x0C,0x0A,0x06,0x0C,0x18,0x08,0x06,0x06,
    0x10,0x0C,0x02,0x12,0x0A,0x14,0x0A,0x02,0x06,0x12,0x04,0x02,0x28,0x06,0x02,0x10,
    0x02,0x04,0x08,0x12,0x0A,0x0C,0x06,0x02,0x0A,0x08,0x04,0x06,0x0C,0x02,0x0A,0x12,
    0x08,0x06,0x04,0x14,0x04,0x06,0x24,0x06,0x02,0x0A,0x06,0x18,0x06,0x0E,0x10,0x06
```

```
246   #endif
247   // 2560
248   #if PRIME_DIFF_TABLE_BYTES > 2560
249     ,0x12,0x02,0x0A,0x14,0x0A,0x08,0x06,0x04,0x06,0x02,0x0A,0x02,0x0C,0x04,0x02,0x04,
250      0x08,0x0A,0x06,0x0C,0x12,0x0E,0x0C,0x10,0x08,0x06,0x10,0x08,0x04,0x02,0x06,0x12,
251      0x18,0x12,0x0A,0x0C,0x02,0x04,0x0E,0x0A,0x06,0x06,0x06,0x12,0x0C,0x02,0x1C,0x12,
252      0x0E,0x10,0x0C,0x0E,0x18,0x0C,0x16,0x06,0x02,0x0A,0x08,0x04,0x02,0x04,0x0E,0x0C,
253      0x06,0x04,0x06,0x0E,0x04,0x02,0x04,0x1E,0x06,0x02,0x06,0x0A,0x02,0x1E,0x16,0x02,
254      0x04,0x06,0x08,0x06,0x06,0x10,0x0C,0x0C,0x06,0x08,0x04,0x02,0x18,0x0C,0x04,0x06,
255      0x08,0x06,0x06,0x0A,0x02,0x06,0x0C,0x1C,0x0E,0x06,0x04,0x0C,0x08,0x06,0x0C,0x04,
256      0x06,0x0E,0x06,0x0C,0x0A,0x06,0x06,0x08,0x06,0x06,0x04,0x02,0x04,0x08,0x0C,0x04,
257      0x0E,0x12,0x0A,0x02,0x10,0x06,0x14,0x06,0x0A,0x08,0x04,0x1E,0x24,0x0C,0x08,0x16,
258      0x0C,0x02,0x06,0x0C,0x10,0x06,0x06,0x02,0x12,0x04,0x1A,0x04,0x08,0x12,0x0A,0x08,
259      0x0A,0x06,0x0E,0x04,0x14,0x16,0x12,0x0C,0x08,0x1C,0x0C,0x06,0x06,0x08,0x06,0x0C,
260      0x18,0x10,0x0E,0x04,0x0E,0x0C,0x06,0x0A,0x0C,0x14,0x06,0x04,0x08,0x12,0x0C,0x12,
261      0x0A,0x02,0x04,0x14,0x0A,0x0E,0x04,0x06,0x02,0x0A,0x18,0x12,0x02,0x04,0x14,0x10,
262      0x0E,0x0A,0x0E,0x06,0x04,0x06,0x14,0x06,0x0A,0x06,0x02,0x0C,0x06,0x1E,0x0A,0x08,
263      0x06,0x04,0x06,0x08,0x28,0x02,0x04,0x02,0x0C,0x12,0x04,0x06,0x08,0x0A,0x06,0x12,
264      0x12,0x02,0x0C,0x10,0x08,0x06,0x04,0x06,0x06,0x02,0x34,0x0E,0x04,0x14,0x10,0x02
265   #endif
266   // 2816
267   #if PRIME_DIFF_TABLE_BYTES > 2816
268     ,0x04,0x06,0x0C,0x02,0x06,0x0C,0x0C,0x06,0x04,0x0E,0x0A,0x06,0x06,0x0E,0x0A,0x0E,
269      0x10,0x08,0x06,0x0C,0x04,0x08,0x16,0x06,0x02,0x12,0x16,0x06,0x02,0x12,0x06,0x10,
270      0x0E,0x0A,0x06,0x0C,0x02,0x06,0x04,0x08,0x12,0x0C,0x10,0x02,0x04,0x0E,0x04,0x08,
271      0x0C,0x0C,0x1E,0x10,0x08,0x04,0x02,0x06,0x16,0x0C,0x08,0x0A,0x06,0x06,0x06,0x0E,
272      0x06,0x12,0x0A,0x0C,0x02,0x0A,0x02,0x04,0x1A,0x04,0x0C,0x08,0x04,0x12,0x08,0x0A,
273      0x0E,0x10,0x06,0x06,0x08,0x0A,0x06,0x08,0x06,0x0C,0x0A,0x14,0x0A,0x08,0x04,0x0C,
274      0x1A,0x12,0x04,0x0C,0x12,0x06,0x1E,0x06,0x08,0x06,0x16,0x0C,0x02,0x04,0x06,0x06,
275      0x02,0x0A,0x02,0x04,0x06,0x06,0x02,0x06,0x16,0x12,0x06,0x12,0x0C,0x08,0x0C,0x06,
276      0x0A,0x0C,0x02,0x10,0x02,0x0A,0x02,0x0A,0x12,0x06,0x14,0x04,0x02,0x06,0x16,0x06,
277      0x06,0x12,0x06,0x0E,0x0C,0x10,0x02,0x06,0x06,0x04,0x0E,0x0C,0x04,0x02,0x12,0x10,
278      0x24,0x0C,0x06,0x0E,0x1C,0x02,0x0C,0x06,0x0C,0x06,0x04,0x02,0x10,0x1E,0x08,0x18,
279      0x06,0x1E,0x0A,0x02,0x12,0x04,0x06,0x0C,0x08,0x16,0x02,0x06,0x16,0x12,0x02,0x0A,
280      0x02,0x0A,0x1E,0x02,0x1C,0x06,0x0E,0x10,0x06,0x14,0x10,0x02,0x06,0x04,0x20,0x04,
281      0x02,0x04,0x06,0x02,0x0C,0x04,0x06,0x06,0x0C,0x02,0x06,0x04,0x06,0x08,0x06,0x04,
282      0x14,0x04,0x20,0x0A,0x08,0x10,0x02,0x16,0x02,0x04,0x06,0x08,0x06,0x10,0x0E,0x04,
283      0x12,0x08,0x04,0x14,0x06,0x0C,0x0C,0x06,0x0A,0x02,0x0A,0x02,0x0C,0x1C,0x0C,0x12
284   #endif
285   // 3072
286   #if PRIME_DIFF_TABLE_BYTES > 3072
287     ,0x02,0x12,0x0A,0x08,0x0A,0x30,0x02,0x04,0x06,0x08,0x0A,0x02,0x0A,0x1E,0x02,0x24,
288      0x06,0x0A,0x06,0x02,0x12,0x04,0x06,0x08,0x10,0x0E,0x10,0x06,0x0E,0x04,0x14,0x04,
289      0x06,0x02,0x0A,0x0C,0x02,0x06,0x0C,0x06,0x06,0x04,0x0C,0x02,0x06,0x04,0x0C,0x06,
290      0x08,0x04,0x02,0x06,0x12,0x0A,0x06,0x08,0x0C,0x06,0x16,0x02,0x06,0x0C,0x12,0x04,
291      0x0E,0x06,0x04,0x14,0x06,0x10,0x08,0x04,0x08,0x16,0x08,0x0C,0x06,0x06,0x10,0x0C,
292      0x12,0x1E,0x08,0x04,0x02,0x04,0x06,0x1A,0x04,0x0E,0x18,0x16,0x06,0x02,0x06,0x0A,
293      0x06,0x0E,0x06,0x06,0x0C,0x0A,0x06,0x02,0x0C,0x0A,0x0C,0x08,0x12,0x12,0x0A,0x06,
294      0x08,0x10,0x06,0x06,0x08,0x10,0x14,0x04,0x02,0x0A,0x02,0x0A,0x0C,0x06,0x08,0x06,
295      0x0A,0x14,0x0A,0x12,0x1A,0x04,0x06,0x1E,0x02,0x04,0x08,0x06,0x0C,0x0C,0x12,0x04,
296      0x08,0x16,0x06,0x02,0x0C,0x22,0x06,0x12,0x0C,0x06,0x02,0x1C,0x0E,0x10,0x0E,0x04,
297      0x0E,0x0C,0x04,0x06,0x06,0x02,0x24,0x04,0x06,0x14,0x0C,0x18,0x06,0x16,0x02,0x10,
298      0x12,0x0C,0x0C,0x12,0x02,0x06,0x06,0x06,0x04,0x06,0x0E,0x04,0x02,0x16,0x08,0x0C,
299      0x06,0x0A,0x06,0x08,0x0C,0x12,0x0C,0x06,0x0A,0x02,0x16,0x0E,0x06,0x06,0x04,0x12,
300      0x06,0x14,0x16,0x02,0x0C,0x18,0x04,0x12,0x12,0x02,0x16,0x02,0x04,0x0C,0x08,0x0C,
301      0x0A,0x0E,0x04,0x02,0x12,0x10,0x26,0x06,0x06,0x06,0x0C,0x0A,0x06,0x0C,0x08,0x06,
302      0x04,0x06,0x0E,0x1E,0x06,0x0A,0x08,0x16,0x06,0x08,0x0C,0x0A,0x02,0x0A,0x02,0x06
303   #endif
304   // 3328
305   #if PRIME_DIFF_TABLE_BYTES > 3328
306     ,0x0A,0x02,0x0A,0x0C,0x12,0x14,0x06,0x04,0x08,0x16,0x06,0x06,0x1E,0x06,0x0E,0x06,
307      0x0C,0x0C,0x06,0x0A,0x02,0x0A,0x1E,0x02,0x10,0x08,0x04,0x02,0x06,0x12,0x04,0x02,
308      0x06,0x04,0x1A,0x04,0x08,0x06,0x0A,0x02,0x04,0x06,0x08,0x04,0x06,0x1E,0x0C,0x02,
309      0x06,0x06,0x04,0x14,0x16,0x08,0x04,0x02,0x04,0x48,0x08,0x04,0x08,0x16,0x02,0x04,
310      0x0E,0x0A,0x02,0x04,0x14,0x06,0x0A,0x12,0x06,0x14,0x10,0x06,0x08,0x06,0x04,0x14,
311      0x0C,0x16,0x02,0x04,0x02,0x0C,0x0A,0x12,0x02,0x16,0x06,0x12,0x1E,0x02,0x0A,0x0E,
```

```
312       0x0A,0x08,0x10,0x32,0x06,0x0A,0x08,0x0A,0x0C,0x06,0x12,0x02,0x16,0x06,0x02,0x04,
313       0x06,0x08,0x06,0x06,0x0A,0x12,0x02,0x16,0x02,0x10,0x0E,0x0A,0x06,0x02,0x0C,0x0A,
314       0x14,0x04,0x0E,0x06,0x04,0x24,0x02,0x04,0x06,0x0C,0x02,0x04,0x0E,0x0C,0x06,0x04,
315       0x06,0x02,0x06,0x04,0x14,0x0A,0x02,0x0A,0x06,0x0C,0x02,0x18,0x0C,0x0C,0x06,0x06,
316       0x04,0x18,0x02,0x04,0x18,0x02,0x06,0x04,0x06,0x08,0x10,0x06,0x02,0x0A,0x0C,0x0E,
317       0x06,0x22,0x06,0x0E,0x06,0x04,0x02,0x1E,0x16,0x08,0x04,0x06,0x08,0x04,0x02,0x1C,
318       0x02,0x06,0x04,0x1A,0x12,0x16,0x02,0x06,0x10,0x06,0x02,0x10,0x0C,0x02,0x0C,0x04,
319       0x06,0x06,0x0E,0x0A,0x06,0x08,0x0C,0x04,0x12,0x02,0x0A,0x08,0x10,0x06,0x06,0x1E,
320       0x02,0x0A,0x12,0x02,0x0A,0x08,0x04,0x08,0x0C,0x18,0x28,0x02,0x0C,0x0A,0x06,0x0C,
321       0x02,0x0C,0x04,0x02,0x04,0x06,0x12,0x0E,0x0C,0x06,0x04,0x0E,0x1E,0x04,0x08,0x0A
322   #endif
323   // 3584
324   #if PRIME_DIFF_TABLE_BYTES > 3584
325     ,0x08,0x06,0x0A,0x12,0x08,0x04,0x0E,0x10,0x06,0x08,0x04,0x06,0x02,0x0A,0x02,0x0C,
326       0x04,0x02,0x04,0x06,0x08,0x04,0x06,0x20,0x18,0x0A,0x08,0x12,0x0A,0x02,0x06,0x0A,
327       0x02,0x04,0x12,0x06,0x0C,0x02,0x10,0x02,0x16,0x06,0x06,0x08,0x12,0x04,0x12,0x0C,
328       0x08,0x06,0x04,0x14,0x06,0x1E,0x16,0x0C,0x02,0x06,0x12,0x04,0x3E,0x04,0x02,0x0C,
329       0x06,0x0A,0x02,0x0C,0x0C,0x1C,0x02,0x04,0x0E,0x16,0x06,0x02,0x06,0x06,0x0A,0x0E,
330       0x04,0x02,0x0A,0x06,0x08,0x0A,0x0E,0x0A,0x06,0x02,0x0C,0x16,0x12,0x08,0x0A,0x12,
331       0x0C,0x02,0x0C,0x04,0x0C,0x02,0x0A,0x02,0x06,0x12,0x06,0x06,0x22,0x06,0x02,0x0C,
332       0x04,0x06,0x12,0x12,0x02,0x10,0x06,0x06,0x08,0x06,0x0A,0x12,0x08,0x0A,0x08,0x0A,
333       0x02,0x04,0x12,0x1A,0x0C,0x16,0x02,0x04,0x02,0x16,0x06,0x06,0x0E,0x10,0x06,0x14,
334       0x0A,0x0C,0x02,0x12,0x2A,0x04,0x18,0x02,0x06,0x0A,0x0C,0x02,0x06,0x0A,0x08,0x04,
335       0x06,0x0C,0x0C,0x08,0x04,0x06,0x0C,0x1E,0x14,0x06,0x18,0x06,0x0A,0x0C,0x02,0x0A,
336       0x14,0x06,0x06,0x04,0x0C,0x0E,0x0A,0x12,0x0C,0x08,0x06,0x0C,0x04,0x0E,0x0A,0x02,
337       0x0C,0x1E,0x10,0x02,0x0C,0x06,0x04,0x02,0x04,0x06,0x1A,0x04,0x12,0x02,0x04,0x06,
338       0x0E,0x36,0x06,0x34,0x02,0x10,0x06,0x06,0x0C,0x1A,0x04,0x02,0x06,0x16,0x06,0x02,
339       0x0C,0x0C,0x06,0x0A,0x12,0x02,0x0C,0x0C,0x0A,0x12,0x0C,0x06,0x08,0x06,0x0A,0x06,
340       0x08,0x04,0x02,0x04,0x14,0x18,0x06,0x06,0x0A,0x0E,0x0A,0x02,0x16,0x06,0x0E,0x0A
341   #endif
342   // 3840
343   #if PRIME_DIFF_TABLE_BYTES > 3840
344     ,0x1A,0x04,0x12,0x08,0x0C,0x0C,0x0A,0x0C,0x06,0x08,0x10,0x06,0x08,0x06,0x06,0x16,
345       0x02,0x0A,0x14,0x0A,0x06,0x2C,0x12,0x06,0x0A,0x02,0x04,0x06,0x0E,0x04,0x1A,0x04,
346       0x02,0x0C,0x0A,0x08,0x04,0x08,0x0C,0x04,0x0C,0x08,0x16,0x08,0x06,0x0A,0x12,0x06,
347       0x06,0x08,0x06,0x0C,0x04,0x08,0x12,0x0A,0x0C,0x06,0x0C,0x02,0x06,0x04,0x02,0x10,
348       0x0C,0x0C,0x0E,0x0A,0x0E,0x06,0x0A,0x0C,0x02,0x0C,0x06,0x04,0x06,0x02,0x0C,0x04,
349       0x1A,0x06,0x12,0x06,0x0A,0x06,0x02,0x12,0x0A,0x08,0x04,0x1A,0x0A,0x14,0x06,0x10,
350       0x14,0x0C,0x0A,0x08,0x0A,0x02,0x10,0x06,0x14,0x0A,0x14,0x04,0x1E,0x02,0x04,0x08,
351       0x10,0x02,0x12,0x04,0x02,0x06,0x0A,0x12,0x0C,0x0E,0x12,0x06,0x10,0x14,0x06,0x04,
352       0x08,0x06,0x04,0x06,0x0C,0x08,0x0A,0x02,0x0C,0x06,0x04,0x02,0x06,0x0A,0x02,0x10,
353       0x0C,0x0E,0x0A,0x06,0x08,0x06,0x1C,0x02,0x06,0x12,0x1E,0x22,0x02,0x10,0x0C,0x02,
354       0x12,0x10,0x06,0x08,0x0A,0x08,0x0A,0x08,0x0A,0x2C,0x06,0x06,0x04,0x14,0x04,0x02,
355       0x04,0x0E,0x1C,0x08,0x06,0x10,0x0E,0x1E,0x06,0x1E,0x04,0x0E,0x0A,0x06,0x06,0x08,
356       0x04,0x12,0x0C,0x06,0x02,0x16,0x0C,0x08,0x06,0x0C,0x04,0x0E,0x04,0x06,0x02,0x04,
357       0x12,0x14,0x06,0x10,0x26,0x10,0x02,0x04,0x06,0x02,0x28,0x2A,0x0E,0x04,0x06,0x02,
358       0x18,0x0A,0x06,0x02,0x12,0x0A,0x0C,0x02,0x10,0x02,0x06,0x10,0x06,0x08,0x04,0x02,
359       0x0A,0x06,0x08,0x0A,0x02,0x12,0x10,0x08,0x0C,0x12,0x0C,0x06,0x0C,0x0A,0x06,0x06
360   #endif
361   // 4096
362   #if PRIME_DIFF_TABLE_BYTES > 4096
363     ,0x12,0x0C,0x0E,0x04,0x02,0x0A,0x14,0x06,0x0C,0x06,0x10,0x1A,0x04,0x12,0x02,0x04,
364       0x20,0x0A,0x08,0x06,0x04,0x06,0x06,0x0E,0x06,0x12,0x04,0x02,0x12,0x0A,0x08,0x0A,
365       0x08,0x0A,0x02,0x04,0x06,0x02,0x0A,0x2A,0x08,0x0C,0x04,0x06,0x12,0x02,0x10,0x08,
366       0x04,0x02,0x0A,0x0E,0x0C,0x0A,0x14,0x04,0x08,0x0A,0x26,0x04,0x06,0x02,0x0A,0x14,
367       0x0A,0x0C,0x06,0x0C,0x1A,0x0C,0x04,0x08,0x1C,0x08,0x04,0x08,0x18,0x06,0x0A,0x08,
368       0x06,0x10,0x0C,0x08,0x0A,0x0C,0x08,0x16,0x06,0x02,0x0A,0x02,0x06,0x0A,0x06,0x06,
369       0x08,0x06,0x04,0x0E,0x1C,0x08,0x10,0x12,0x08,0x04,0x06,0x14,0x04,0x12,0x06,0x02,
370       0x18,0x18,0x06,0x06,0x0C,0x0C,0x04,0x02,0x16,0x02,0x0A,0x06,0x08,0x0C,0x04,0x14,
371       0x12,0x06,0x04,0x0C,0x18,0x06,0x06,0x36,0x08,0x06,0x04,0x1A,0x24,0x04,0x02,0x04,
372       0x1A,0x0C,0x0C,0x04,0x06,0x06,0x08,0x0C,0x0A,0x02,0x0C,0x10,0x12,0x06,0x08,0x06,
373       0x0C,0x12,0x0A,0x02,0x36,0x04,0x02,0x0A,0x1E,0x0C,0x08,0x04,0x08,0x10,0x0E,0x0C,
374       0x06,0x04,0x06,0x0C,0x06,0x02,0x04,0x0E,0x0C,0x04,0x0E,0x06,0x18,0x06,0x06,0x0A,
375       0x0C,0x0C,0x14,0x12,0x06,0x06,0x10,0x08,0x04,0x06,0x14,0x04,0x20,0x04,0x0E,0x0A,
376       0x02,0x06,0x0C,0x10,0x02,0x04,0x06,0x0C,0x02,0x0A,0x08,0x06,0x04,0x02,0x0A,0x0E,
377       0x06,0x06,0x0C,0x12,0x22,0x08,0x0A,0x06,0x18,0x06,0x02,0x0A,0x0C,0x02,0x1E,0x0A,
```



```
378       0x0E,0x0C,0x0C,0x10,0x06,0x06,0x02,0x12,0x04,0x06,0x1E,0x0E,0x04,0x06,0x06,0x02
379   #endif
380   // 4352
381   #if PRIME_DIFF_TABLE_BYTES > 4352
382     ,0x06,0x04,0x06,0x0E,0x06,0x04,0x08,0x0A,0x0C,0x06,0x20,0x0A,0x08,0x16,0x02,0x0A,
383      0x06,0x18,0x08,0x04,0x1E,0x06,0x02,0x0C,0x10,0x08,0x06,0x04,0x06,0x08,0x10,0x0E,
384      0x06,0x06,0x04,0x02,0x0A,0x0C,0x02,0x10,0x0E,0x04,0x02,0x04,0x14,0x12,0x0A,0x02,
385      0x0A,0x06,0x0C,0x1E,0x08,0x12,0x0C,0x0A,0x02,0x06,0x06,0x04,0x0C,0x0C,0x02,0x04,
386      0x0C,0x12,0x18,0x02,0x0A,0x06,0x08,0x10,0x08,0x06,0x0C,0x0A,0x0E,0x06,0x0C,0x06,
387      0x06,0x04,0x02,0x18,0x04,0x06,0x08,0x06,0x04,0x02,0x04,0x06,0x0E,0x04,0x08,0x0A,
388      0x18,0x18,0x0C,0x02,0x06,0x0C,0x16,0x1E,0x02,0x06,0x12,0x0A,0x06,0x06,0x08,0x04,
389      0x02,0x06,0x0A,0x08,0x0A,0x06,0x08,0x10,0x06,0x0E,0x06,0x04,0x18,0x08,0x0A,0x02,
390      0x0C,0x06,0x04,0x24,0x02,0x16,0x06,0x08,0x06,0x0A,0x08,0x06,0x0C,0x0A,0x0E,0x0A,
391      0x06,0x12,0x0C,0x02,0x0C,0x04,0x1A,0x0A,0x0E,0x10,0x12,0x08,0x12,0x0C,0x0C,0x06,
392      0x10,0x0E,0x18,0x0A,0x0C,0x08,0x16,0x06,0x02,0x0A,0x3C,0x06,0x02,0x04,0x08,0x10,
393      0x0E,0x0A,0x06,0x18,0x06,0x0C,0x12,0x18,0x02,0x1E,0x04,0x02,0x0C,0x06,0x0A,0x02,
394      0x04,0x0E,0x06,0x10,0x02,0x0A,0x08,0x16,0x14,0x06,0x04,0x20,0x06,0x12,0x04,0x02,
395      0x04,0x02,0x04,0x08,0x34,0x0E,0x16,0x02,0x16,0x14,0x0A,0x08,0x0A,0x02,0x06,0x04,
396      0x0E,0x04,0x06,0x14,0x04,0x06,0x02,0x0C,0x0C,0x06,0x0C,0x10,0x02,0x0C,0x0A,0x08,
397      0x04,0x06,0x02,0x1C,0x0C,0x08,0x0A,0x0C,0x02,0x04,0x0E,0x1C,0x08,0x06,0x04,0x02
398   #endif
399   // 4608
400   #if PRIME_DIFF_TABLE_BYTES > 4608
401     ,0x04,0x06,0x02,0x0C,0x3A,0x06,0x0E,0x0A,0x02,0x06,0x1C,0x20,0x04,0x1E,0x08,0x06,
402      0x04,0x06,0x0C,0x0C,0x02,0x04,0x06,0x06,0x0E,0x10,0x08,0x1E,0x04,0x02,0x0A,0x08,
403      0x06,0x04,0x06,0x1A,0x04,0x0C,0x02,0x0A,0x12,0x0C,0x0C,0x12,0x02,0x04,0x0C,0x08,
404      0x0C,0x0A,0x14,0x04,0x08,0x10,0x0C,0x08,0x06,0x10,0x08,0x0A,0x0C,0x0E,0x06,0x04,
405      0x08,0x0C,0x04,0x14,0x06,0x28,0x08,0x10,0x06,0x24,0x02,0x06,0x04,0x06,0x02,0x16,
406      0x12,0x02,0x0A,0x06,0x24,0x0E,0x0C,0x04,0x12,0x08,0x04,0x0E,0x0A,0x02,0x0A,0x08,
407      0x04,0x02,0x12,0x10,0x0C,0x0E,0x0A,0x0E,0x06,0x06,0x2A,0x0A,0x06,0x06,0x14,0x0A,
408      0x08,0x0C,0x04,0x0C,0x12,0x02,0x0A,0x0E,0x12,0x0A,0x12,0x08,0x06,0x04,0x0E,0x06,
409      0x0A,0x1E,0x0E,0x06,0x06,0x04,0x0C,0x26,0x04,0x02,0x04,0x06,0x08,0x0C,0x0A,0x06,
410      0x12,0x06,0x32,0x06,0x04,0x06,0x0C,0x08,0x0A,0x20,0x06,0x16,0x02,0x0A,0x0C,0x12,
411      0x02,0x06,0x04,0x1E,0x08,0x06,0x06,0x12,0x0A,0x02,0x04,0x0C,0x14,0x0A,0x08,0x18,
412      0x0A,0x02,0x06,0x16,0x06,0x02,0x12,0x0A,0x0C,0x02,0x1E,0x12,0x0C,0x1C,0x02,0x06,
413      0x04,0x06,0x0E,0x06,0x0C,0x0A,0x08,0x04,0x0C,0x1A,0x0A,0x08,0x06,0x10,0x02,0x0A,
414      0x12,0x0E,0x06,0x04,0x06,0x0E,0x10,0x02,0x06,0x04,0x0C,0x14,0x04,0x14,0x04,0x06,
415      0x0C,0x02,0x24,0x04,0x06,0x02,0x0A,0x02,0x16,0x08,0x06,0x0A,0x0C,0x0C,0x12,0x0E,
416      0x18,0x24,0x04,0x14,0x18,0x0A,0x06,0x02,0x1C,0x06,0x12,0x08,0x04,0x06,0x08,0x06
417   #endif
418   // 4864
419   #if PRIME_DIFF_TABLE_BYTES > 4864
420     ,0x04,0x02,0x0C,0x1C,0x12,0x0E,0x10,0x0E,0x12,0x0A,0x08,0x06,0x04,0x06,0x06,0x08,
421      0x16,0x0C,0x02,0x0A,0x12,0x06,0x02,0x12,0x0A,0x02,0x0C,0x0A,0x12,0x20,0x06,0x04,
422      0x06,0x06,0x08,0x06,0x06,0x0A,0x14,0x06,0x0C,0x0A,0x08,0x0A,0x0E,0x06,0x0A,0x0E,
423      0x04,0x02,0x16,0x12,0x02,0x0A,0x02,0x04,0x14,0x04,0x02,0x22,0x02,0x0C,0x06,0x0A,
424      0x02,0x0A,0x12,0x06,0x0E,0x0C,0x0C,0x16,0x08,0x06,0x10,0x06,0x08,0x04,0x0C,0x06,
425      0x08,0x04,0x24,0x06,0x06,0x14,0x18,0x06,0x0C,0x12,0x0A,0x02,0x0A,0x1A,0x06,0x10,
426      0x08,0x06,0x04,0x18,0x12,0x08,0x0C,0x0C,0x0A,0x12,0x0C,0x02,0x18,0x04,0x0C,0x12,
427      0x0C,0x0E,0x0A,0x02,0x04,0x18,0x0C,0x0E,0x0A,0x06,0x02,0x06,0x04,0x06,0x1A,0x04,
428      0x06,0x06,0x02,0x16,0x08,0x12,0x04,0x12,0x08,0x04,0x18,0x02,0x0C,0x0C,0x04,0x02,
429      0x34,0x02,0x12,0x06,0x04,0x06,0x0C,0x02,0x06,0x0C,0x0A,0x08,0x04,0x02,0x18,0x0A,
430      0x02,0x0A,0x02,0x0C,0x06,0x12,0x28,0x06,0x14,0x10,0x02,0x0C,0x06,0x0A,0x0C,0x02,
431      0x04,0x06,0x0E,0x0C,0x0C,0x16,0x06,0x08,0x04,0x02,0x10,0x12,0x0C,0x02,0x06,0x10,
432      0x06,0x02,0x06,0x04,0x0C,0x1E,0x08,0x10,0x02,0x12,0x0A,0x18,0x02,0x06,0x18,0x04,
433      0x02,0x16,0x02,0x10,0x02,0x06,0x0C,0x04,0x12,0x08,0x04,0x0E,0x04,0x12,0x18,0x06,
434      0x02,0x06,0x0A,0x02,0x0A,0x26,0x06,0x0A,0x0E,0x06,0x06,0x18,0x04,0x02,0x0C,0x10,
435      0x0E,0x10,0x0C,0x02,0x06,0x0A,0x1A,0x04,0x02,0x0C,0x06,0x04,0x0C,0x08,0x0C,0x0A
436   #endif
437   // 5120
438   #if PRIME_DIFF_TABLE_BYTES > 5120
439     ,0x12,0x06,0x0E,0x1C,0x02,0x06,0x0A,0x02,0x04,0x0E,0x22,0x02,0x06,0x16,0x02,0x0A,
440      0x0E,0x04,0x02,0x10,0x08,0x0A,0x06,0x08,0x0A,0x08,0x04,0x06,0x02,0x10,0x06,0x06,
441      0x12,0x1E,0x0E,0x06,0x04,0x1E,0x02,0x0A,0x0E,0x04,0x14,0x0A,0x08,0x04,0x08,0x12,
442      0x04,0x0E,0x06,0x04,0x18,0x06,0x06,0x12,0x12,0x02,0x24,0x06,0x0A,0x0E,0x0C,0x04,
443      0x06,0x02,0x1E,0x06,0x04,0x02,0x06,0x1C,0x14,0x04,0x14,0x0C,0x18,0x10,0x12,0x0C,
```

```
444       0x0E,0x06,0x04,0x0C,0x20,0x0C,0x06,0x0A,0x08,0x0A,0x06,0x12,0x02,0x10,0x0E,0x06,
445       0x16,0x06,0x0C,0x02,0x12,0x04,0x08,0x1E,0x0C,0x04,0x0C,0x02,0x0A,0x26,0x16,0x02,
446       0x04,0x0E,0x06,0x0C,0x18,0x04,0x02,0x04,0x0E,0x0C,0x0A,0x02,0x10,0x06,0x14,0x04,
447       0x14,0x16,0x0C,0x02,0x04,0x02,0x0C,0x16,0x18,0x06,0x06,0x02,0x06,0x04,0x06,0x02,
448       0x0A,0x0C,0x0C,0x06,0x02,0x06,0x10,0x08,0x06,0x04,0x12,0x0C,0x0C,0x0E,0x04,0x0C,
449       0x06,0x08,0x06,0x12,0x06,0x0A,0x0C,0x0E,0x06,0x04,0x08,0x16,0x06,0x02,0x1C,0x12,
450       0x02,0x12,0x0A,0x06,0x0E,0x0A,0x02,0x0A,0x0E,0x06,0x0A,0x02,0x16,0x06,0x08,0x06,
451       0x10,0x0C,0x08,0x16,0x02,0x04,0x0E,0x12,0x0C,0x06,0x18,0x06,0x0A,0x02,0x0C,0x16,
452       0x12,0x06,0x14,0x06,0x0A,0x0E,0x04,0x02,0x06,0x0C,0x16,0x0E,0x0C,0x04,0x06,0x08,
453       0x16,0x02,0x0A,0x0C,0x08,0x28,0x02,0x06,0x0A,0x08,0x04,0x2A,0x14,0x04,0x20,0x0C,
454       0x0A,0x06,0x0C,0x0C,0x02,0x0A,0x08,0x06,0x04,0x08,0x04,0x1A,0x12,0x04,0x08,0x1C
455   #endif
456   // 5376
457   #if PRIME_DIFF_TABLE_BYTES > 5376
458     ,0x06,0x12,0x06,0x0C,0x02,0x0A,0x06,0x06,0x0E,0x0A,0x0C,0x0E,0x18,0x06,0x04,0x14,
459       0x16,0x02,0x12,0x04,0x06,0x0C,0x02,0x10,0x12,0x0E,0x06,0x06,0x04,0x06,0x08,0x12,
460       0x04,0x0E,0x1E,0x04,0x12,0x08,0x0A,0x02,0x04,0x08,0x0C,0x04,0x0C,0x12,0x02,0x0C,
461       0x0A,0x02,0x10,0x08,0x04,0x1E,0x02,0x06,0x1C,0x02,0x0A,0x02,0x12,0x0A,0x0E,0x04,
462       0x1A,0x06,0x12,0x04,0x14,0x06,0x04,0x08,0x12,0x04,0x0C,0x1A,0x18,0x04,0x14,0x16,
463       0x02,0x12,0x16,0x02,0x04,0x0C,0x02,0x06,0x06,0x06,0x04,0x06,0x0E,0x04,0x18,0x0C,
464       0x06,0x12,0x02,0x0C,0x1C,0x0E,0x04,0x06,0x08,0x16,0x06,0x0C,0x12,0x08,0x04,0x14,
465       0x06,0x04,0x06,0x02,0x12,0x06,0x04,0x0C,0x0C,0x08,0x1C,0x06,0x08,0x0A,0x02,0x18,
466       0x0C,0x0A,0x18,0x08,0x0A,0x14,0x0C,0x06,0x0C,0x0C,0x04,0x0E,0x0C,0x18,0x22,0x12,
467       0x08,0x0A,0x06,0x12,0x08,0x04,0x08,0x10,0x0E,0x06,0x04,0x06,0x18,0x02,0x06,0x04,
468       0x06,0x02,0x10,0x06,0x06,0x14,0x18,0x04,0x02,0x04,0x0E,0x04,0x12,0x02,0x06,0x0C,
469       0x04,0x0E,0x04,0x02,0x12,0x10,0x06,0x06,0x02,0x10,0x14,0x06,0x06,0x1E,0x04,0x08,
470       0x06,0x18,0x10,0x06,0x06,0x08,0x0C,0x1E,0x04,0x12,0x12,0x08,0x04,0x1A,0x0A,0x02,
471       0x16,0x08,0x0A,0x0E,0x06,0x04,0x12,0x08,0x0C,0x1C,0x02,0x06,0x04,0x0C,0x06,0x18,
472       0x06,0x08,0x0A,0x14,0x10,0x08,0x1E,0x06,0x06,0x04,0x02,0x0A,0x0E,0x06,0x0A,0x20,
473       0x16,0x12,0x02,0x04,0x02,0x04,0x08,0x16,0x08,0x12,0x0C,0x1C,0x02,0x10,0x0C,0x12
474   #endif
475   // 5632
476   #if PRIME_DIFF_TABLE_BYTES > 5632
477     ,0x0E,0x0A,0x12,0x0C,0x06,0x20,0x0A,0x0E,0x06,0x0A,0x02,0x0A,0x02,0x06,0x16,0x02,
478       0x04,0x06,0x08,0x0A,0x06,0x0E,0x06,0x04,0x0C,0x1E,0x18,0x06,0x06,0x08,0x06,0x04,
479       0x02,0x04,0x06,0x08,0x06,0x06,0x16,0x12,0x08,0x04,0x02,0x12,0x06,0x04,0x02,0x10,
480       0x12,0x14,0x0A,0x06,0x06,0x1E,0x02,0x0C,0x1C,0x06,0x06,0x06,0x02,0x0C,0x0A,0x08,
481       0x12,0x12,0x04,0x08,0x12,0x0A,0x02,0x1C,0x02,0x0A,0x0E,0x04,0x02,0x1E,0x0C,0x16,
482       0x1A,0x0A,0x08,0x06,0x0A,0x08,0x10,0x0E,0x06,0x06,0x0A,0x0E,0x06,0x04,0x02,0x0A,
483       0x0C,0x02,0x06,0x0A,0x08,0x04,0x02,0x0A,0x1A,0x16,0x06,0x02,0x0C,0x12,0x04,0x1A,
484       0x04,0x08,0x0A,0x06,0x0E,0x0A,0x02,0x12,0x06,0x0A,0x14,0x06,0x06,0x04,0x18,0x02,
485       0x04,0x08,0x06,0x10,0x0E,0x10,0x12,0x02,0x04,0x0C,0x02,0x0A,0x02,0x06,0x0C,0x0A,
486       0x06,0x06,0x14,0x06,0x04,0x06,0x26,0x04,0x06,0x0C,0x0E,0x04,0x0C,0x08,0x0A,0x0C,
487       0x0C,0x08,0x04,0x06,0x0E,0x0A,0x06,0x0C,0x02,0x0A,0x12,0x02,0x12,0x0A,0x08,0x0A,
488       0x02,0x0C,0x04,0x0E,0x1C,0x02,0x10,0x02,0x12,0x06,0x0A,0x06,0x08,0x10,0x0E,0x1E,
489       0x0A,0x14,0x06,0x0A,0x18,0x02,0x1C,0x02,0x0C,0x10,0x06,0x08,0x24,0x04,0x08,0x04,
490       0x0E,0x0C,0x0A,0x08,0x0C,0x04,0x06,0x08,0x04,0x06,0x0E,0x16,0x08,0x06,0x04,0x02,
491       0x0A,0x06,0x14,0x0A,0x08,0x06,0x06,0x16,0x12,0x02,0x10,0x06,0x14,0x04,0x1A,0x04,
492       0x0E,0x16,0x0E,0x04,0x0C,0x06,0x08,0x04,0x06,0x06,0x1A,0x0A,0x02,0x12,0x12,0x04
493   #endif
494   // 5888
495   #if PRIME_DIFF_TABLE_BYTES > 5888
496     ,0x02,0x10,0x02,0x12,0x04,0x06,0x08,0x04,0x06,0x0C,0x02,0x06,0x06,0x1C,0x26,0x04,
497       0x08,0x10,0x1A,0x04,0x02,0x0A,0x0C,0x02,0x0A,0x08,0x06,0x0A,0x0C,0x02,0x0A,0x02,
498       0x18,0x04,0x1E,0x1A,0x06,0x06,0x12,0x06,0x06,0x16,0x02,0x0A,0x12,0x1A,0x04,0x12,
499       0x08,0x06,0x06,0x0C,0x10,0x06,0x08,0x10,0x06,0x08,0x10,0x02,0x2A,0x3A,0x08,0x04,
500       0x06,0x02,0x04,0x08,0x10,0x06,0x14,0x04,0x0C,0x0C,0x06,0x0C,0x02,0x0A,0x02,0x06,
501       0x16,0x02,0x0A,0x06,0x08,0x06,0x0A,0x0E,0x06,0x06,0x04,0x12,0x08,0x0A,0x08,0x10,
502       0x0E,0x0A,0x02,0x0A,0x02,0x0C,0x06,0x04,0x14,0x0A,0x08,0x34,0x08,0x0A,0x06,0x02,
503       0x0A,0x08,0x0A,0x06,0x06,0x08,0x0A,0x02,0x16,0x02,0x04,0x06,0x0E,0x04,0x02,0x18,
504       0x0C,0x04,0x1A,0x12,0x04,0x06,0x0E,0x1E,0x06,0x04,0x06,0x02,0x16,0x08,0x04,0x06,
505       0x02,0x16,0x06,0x08,0x10,0x06,0x0E,0x04,0x06,0x12,0x08,0x0C,0x06,0x0C,0x18,0x1E,
506       0x10,0x08,0x22,0x08,0x16,0x06,0x0E,0x0A,0x12,0x0E,0x04,0x0C,0x08,0x04,0x24,0x06,
507       0x06,0x02,0x0A,0x02,0x04,0x14,0x06,0x06,0x0A,0x0C,0x06,0x02,0x28,0x08,0x06,0x1C,
508       0x06,0x02,0x0C,0x12,0x04,0x18,0x0E,0x06,0x06,0x0A,0x14,0x0A,0x0E,0x10,0x0E,0x10,
509       0x06,0x08,0x24,0x04,0x0C,0x0C,0x06,0x0C,0x32,0x0C,0x06,0x04,0x06,0x06,0x08,0x06,
```

```
510       0x0A,0x02,0x0A,0x02,0x12,0x0A,0x0E,0x10,0x08,0x06,0x04,0x14,0x04,0x02,0x0A,0x06,
511       0x0E,0x12,0x0A,0x26,0x0A,0x12,0x02,0x0A,0x02,0x0C,0x04,0x02,0x04,0x0E,0x06,0x0A
512   #endif
513   // 6144
514   #if PRIME_DIFF_TABLE_BYTES > 6144
515     ,0x08,0x28,0x06,0x14,0x04,0x0C,0x08,0x06,0x22,0x08,0x16,0x08,0x0C,0x0A,0x02,0x10,
516       0x2A,0x0C,0x08,0x16,0x08,0x16,0x08,0x06,0x22,0x02,0x06,0x04,0x0E,0x06,0x10,0x02,
517       0x16,0x06,0x08,0x18,0x16,0x06,0x02,0x0C,0x04,0x06,0x0E,0x04,0x08,0x18,0x04,0x06,
518       0x06,0x02,0x16,0x14,0x06,0x04,0x0E,0x04,0x06,0x06,0x08,0x06,0x0A,0x06,0x08,0x06,
519       0x10,0x0E,0x06,0x06,0x16,0x06,0x18,0x20,0x06,0x12,0x06,0x12,0x0A,0x08,0x1E,0x12,
520       0x06,0x10,0x0C,0x06,0x0C,0x02,0x06,0x04,0x0C,0x08,0x06,0x16,0x08,0x06,0x04,0x0E,
521       0x0A,0x12,0x14,0x0A,0x02,0x06,0x04,0x02,0x1C,0x12,0x02,0x0A,0x06,0x06,0x06,0x0E,
522       0x28,0x18,0x02,0x04,0x08,0x0C,0x04,0x14,0x04,0x20,0x12,0x10,0x06,0x24,0x08,0x06,
523       0x04,0x06,0x0E,0x04,0x06,0x1A,0x06,0x0A,0x0E,0x12,0x0A,0x06,0x06,0x0E,0x0A,0x06,
524       0x06,0x0E,0x06,0x18,0x04,0x0E,0x16,0x08,0x0C,0x0A,0x08,0x0C,0x12,0x0A,0x12,0x08,
525       0x18,0x0A,0x08,0x04,0x18,0x06,0x12,0x06,0x02,0x0A,0x1E,0x02,0x0A,0x02,0x04,0x02,
526       0x28,0x02,0x1C,0x08,0x06,0x06,0x12,0x06,0x0A,0x0E,0x04,0x12,0x1E,0x12,0x02,0x0C,
527       0x1E,0x06,0x1E,0x04,0x12,0x0C,0x02,0x04,0x0E,0x06,0x0A,0x06,0x08,0x06,0x0A,0x0C,
528       0x02,0x06,0x0C,0x0A,0x02,0x12,0x04,0x14,0x04,0x06,0x0E,0x06,0x06,0x16,0x06,0x06,
529       0x08,0x12,0x12,0x0A,0x02,0x0A,0x02,0x06,0x04,0x06,0x0C,0x12,0x02,0x0A,0x08,0x04,
530       0x12,0x02,0x06,0x06,0x06,0x0A,0x08,0x0A,0x06,0x12,0x0C,0x08,0x0C,0x06,0x04,0x06
531   #endif
532   // 6400
533   #if PRIME_DIFF_TABLE_BYTES > 6400
534     ,0x0E,0x10,0x02,0x0C,0x04,0x06,0x26,0x06,0x06,0x10,0x14,0x1C,0x14,0x0A,0x06,0x06,
535       0x0E,0x04,0x1A,0x04,0x0E,0x0A,0x12,0x0E,0x1C,0x02,0x04,0x0E,0x10,0x02,0x1C,0x06,
536       0x08,0x06,0x22,0x08,0x04,0x12,0x02,0x10,0x08,0x06,0x28,0x08,0x12,0x04,0x1E,0x06,
537       0x0C,0x02,0x1E,0x06,0x0A,0x0E,0x28,0x0E,0x0A,0x02,0x0C,0x0A,0x08,0x04,0x08,0x06,
538       0x06,0x1C,0x02,0x04,0x0C,0x0E,0x10,0x08,0x1E,0x10,0x12,0x02,0x0A,0x12,0x06,0x20,
539       0x04,0x12,0x06,0x02,0x0C,0x0A,0x12,0x02,0x06,0x0A,0x0E,0x12,0x1C,0x06,0x08,0x10,
540       0x02,0x04,0x14,0x0A,0x08,0x12,0x0A,0x02,0x0A,0x08,0x04,0x06,0x0C,0x06,0x14,0x04,
541       0x02,0x06,0x04,0x14,0x0A,0x1A,0x12,0x0A,0x02,0x12,0x06,0x10,0x0E,0x04,0x1A,0x04,
542       0x0E,0x0A,0x0C,0x0E,0x06,0x06,0x04,0x0E,0x0A,0x02,0x1E,0x12,0x16,0x02
543   #endif
544   // 6542
545   #if PRIME_DIFF_TABLE_BYTES > 0
546      };
547   #endif
548   #if defined  RSA_INSTRUMENT || defined RSA_DEBUG
549   UINT32  failedAtIteration[10];
550   UINT32  MillerRabinTrials;
551   UINT32  totalFields;
552   UINT32  emptyFields;
553   UINT32  noPrimeFields;
554   UINT16  lastSievePrime;
555   UINT32  primesChecked;
556   #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

```
557   #ifdef  RSA_DEBUG
558   const __int16 primes[NUM_PRIMES]=
559       {
560             3,   5,   7,  11,  13,  17,  19,  23,  29,  31,  37,  41,  43,  47,  53,
561        59,  61,  67,  71,  73,  79,  83,  89,  97, 101, 103, 107, 109, 113, 127, 131,
562       137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
563       227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311,
564       313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
565       419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,
566       509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613,
567       617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719,
568       727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827,
569       829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
570       947, 953, 967, 971, 977, 983, 991, 997,1009,1013,1019,1021,1031,1033,1039,1049,
```

```
571     1051,1061,1063,1069,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,1163,
572     1171,1181,1187,1193,1201,1213,1217,1223,1229,1231,1237,1249,1259,1277,1279,1283,
573     1289,1291,1297,1301,1303,1307,1319,1321,1327,1361,1367,1373,1381,1399,1409,1423,
574     1427,1429,1433,1439,1447,1451,1453,1459,1471,1481,1483,1487,1489,1493,1499,1511,
575     1523,1531,1543,1549,1553,1559,1567,1571,1579,1583,1597,1601,1607,1609,1613,1619,
576     1621,1627,1637,1657,1663,1667,1669,1693,1697,1699,1709,1721,1723,1733,1741,1747,
577     1753,1759,1777,1783,1787,1789,1801,1811,1823,1831,1847,1861,1867,1871,1873,1877,
578     1879,1889,1901,1907,1913,1931,1933,1949,1951,1973,1979,1987,1993,1997,1999,2003,
579     2011,2017,2027,2029,2039,2053,2063,2069,2081,2083,2087,2089,2099,2111,2113,2129,
580     2131,2137,2141,2143,2153,2161,2179,2203,2207,2213,2221,2237,2239,2243,2251,2267,
581     2269,2273,2281,2287,2293,2297,2309,2311,2333,2339,2341,2347,2351,2357,2371,2377,
582     2381,2383,2389,2393,2399,2411,2417,2423,2437,2441,2447,2459,2467,2473,2477,2503,
583     2521,2531,2539,2543,2549,2551,2557,2579,2591,2593,2609,2617,2621,2633,2647,2657,
584     2659,2663,2671,2677,2683,2687,2689,2693,2699,2707,2711,2713,2719,2729,2731,2741,
585     2749,2753,2767,2777,2789,2791,2797,2801,2803,2819,2833,2837,2843,2851,2857,2861,
586     2879,2887,2897,2903,2909,2917,2927,2939,2953,2957,2963,2969,2971,2999,3001,3011,
587     3019,3023,3037,3041,3049,3061,3067,3079,3083,3089,3109,3119,3121,3137,3163,3167,
588     3169,3181,3187,3191,3203,3209,3217,3221,3229,3251,3253,3257,3259,3271,3299,3301,
589     3307,3313,3319,3323,3329,3331,3343,3347,3359,3361,3371,3373,3389,3391,3407,3413,
590     3433,3449,3457,3461,3463,3467,3469,3491,3499,3511,3517,3527,3529,3533,3539,3541,
591     3547,3557,3559,3571,3581,3583,3593,3607,3613,3617,3623,3631,3637,3643,3659,3671,
592     3673,3677,3691,3697,3701,3709,3719,3727,3733,3739,3761,3767,3769,3779,3793,3797,
593     3803,3821,3823,3833,3847,3851,3853,3863,3877,3881,3889,3907,3911,3917,3919,3923,
594     3929,3931,3943,3947,3967,3989,4001,4003,4007,4013,4019,4021,4027,4049,4051,4057,
595     4073,4079,4091,4093,4099,4111,4127,4129,4133,4139,4153,4157,4159,4177,4201,4211,
596     4217,4219,4229,4231,4241,4243,4253,4259,4261,4271,4273,4283,4289,4297,4327,4337,
597     4339,4349,4357,4363,4373,4391,4397,4409,4421,4423,4441,4447,4451,4457,4463,4481,
598     4483,4493,4507,4513,4517,4519,4523,4547,4549,4561,4567,4583,4591,4597,4603,4621,
599     4637,4639,4643,4649,4651,4657,4663,4673,4679,4691,4703,4721,4723,4729,4733,4751,
600     4759,4783,4787,4789,4793,4799,4801,4813,4817,4831,4861,4871,4877,4889,4903,4909,
601     4919,4931,4933,4937,4943,4951,4957,4967,4969,4973,4987,4993,4999,5003,5009,5011,
602     5021,5023,5039,5051,5059,5077,5081,5087,5099,5101,5107,5113,5119,5147,5153,5167,
603     5171,5179,5189,5197,5209,5227,5231,5233,5237,5261,5273,5279,5281,5297,5303,5309,
604     5323,5333,5347,5351,5381,5387,5393,5399,5407,5413,5417,5419,5431,5437,5441,5443,
605     5449,5471,5477,5479,5483,5501,5503,5507,5519,5521,5527,5531,5557,5563,5569,5573,
606     5581,5591,5623,5639,5641,5647,5651,5653,5657,5659,5669,5683,5689,5693,5701,5711,
607     5717,5737,5741,5743,5749,5779,5783,5791,5801,5807,5813,5821,5827,5839,5843,5849,
608     5851,5857,5861,5867,5869,5879,5881,5897,5903,5923,5927,5939,5953,5981,5987,6007,
609     6011,6029,6037,6043,6047,6053,6067,6073,6079,6089,6091,6101,6113,6121,6131,6133,
610     6143,6151,6163,6173,6197,6199,6203,6211,6217,6221,6229,6247,6257,6263,6269,6271,
611     6277,6287,6299,6301,6311,6317,6323,6329,6337,6343,6353,6359,6361,6367,6373,6379,
612     6389,6397,6421,6427,6449,6451,6469,6473,6481,6491,6521,6529,6547,6551,6553,6563,
613     6569,6571,6577,6581,6599,6607,6619,6637,6653,6659,6661,6673,6679,6689,6691,6701,
614     6703,6709,6719,6733,6737,6761,6763,6779,6781,6791,6793,6803,6823,6827,6829,6833,
615     6841,6857,6863,6869,6871,6883,6899,6907,6911,6917,6947,6949,6959,6961,6967,6971,
616     6977,6983,6991,6997,7001,7013,7019,7027,7039,7043,7057,7069,7079,7103,7109,7121,
617     7127,7129,7151,7159,7177,7187,7193,7207,7211,7213,7219,7229,7237,7243,7247,7253,
618     7283,7297,7307,7309,7321,7331,7333,7349,7351,7369,7393,7411,7417,7433,7451,7457,
619     7459,7477,7481,7487,7489,7499,7507,7517,7523,7529,7537,7541,7547,7549,7559,7561,
620     7573,7577,7583,7589,7591,7603,7607,7621,7639,7643,7649,7669,7673,7681,7687,7691,
621     7699,7703,7717,7723,7727,7741,7753,7757,7759,7789,7793,7817,7823,7829,7841,7853,
622     7867,7873,7877,7879,7883,7901,7907,7919,7927,7933,7937,7949,7951,7963,7993,8009,
623     8011,8017,8039,8053,8059,8069,8081,8087,8089,8093,8101,8111,8117,8123,8147,8161,
624     8167,8171,8179,8191,8209,8219,8221,8231,8233,8237,8243,8263,8269,8273,8287,8291,
625     8293,8297,8311,8317,8329,8353,8363,8369,8377,8387,8389,8419,8423,8429,8431,8443,
626     8447,8461,8467,8501,8513,8521,8527,8537,8539,8543,8563,8573,8581,8597,8599,8609,
627     8623,8627,8629,8641,8647,8663,8669,8677,8681,8689,8693,8699,8707,8713,8719,8731,
628     8737,8741,8747,8753,8761,8779,8783,8803,8807,8819,8821,8831,8837,8839,8849,8861,
629     8863,8867,8887,8893,8923,8929,8933,8941,8951,8963,8969,8971,8999,9001,9007,9011,
630     9013,9029,9041,9043,9049,9059,9067,9091,9103,9109,9127,9133,9137,9151,9157,9161,
631     9173,9181,9187,9199,9203,9209,9221,9227,9239,9241,9257,9277,9281,9283,9293,9311,
632     9319,9323,9337,9341,9343,9349,9371,9377,9391,9397,9403,9413,9419,9421,9431,9433,
633     9437,9439,9461,9463,9467,9473,9479,9491,9497,9511,9521,9533,9539,9547,9551,9587,
634     9601,9613,9619,9623,9629,9631,9643,9649,9661,9677,9679,9689,9697,9719,9721,9733,
635     9739,9743,9749,9767,9769,9781,9787,9791,9803,9811,9817,9829,9833,9839,9851,9857,
636     9859, 9871, 9883, 9887, 9901, 9907, 9923, 9929,
```

```
637      9931, 9941, 9949, 9967, 9973,10007,10009,10037,
638      10039,10061,10067,10069,10079,10091,10093,10099,
639      10103,10111,10133,10139,10141,10151,10159,10163,
640      10169,10177,10181,10193,10211,10223,10243,10247,
641      10253,10259,10267,10271,10273,10289,10301,10303,
642      10313,10321,10331,10333,10337,10343,10357,10369,
643      10391,10399,10427,10429,10433,10453,10457,10459,
644      10463,10477,10487,10499,10501,10513,10529,10531,
645      10559,10567,10589,10597,10601,10607,10613,10627,
646      10631,10639,10651,10657,10663,10667,10687,10691,
647      10709,10711,10723,10729,10733,10739,10753,10771,
648      10781,10789,10799,10831,10837,10847,10853,10859,
649      10861,10867,10883,10889,10891,10903,10909,10937,
650      10939,10949,10957,10973,10979,10987,10993,11003,
651      11027,11047,11057,11059,11069,11071,11083,11087,
652      11093,11113,11117,11119,11131,11149,11159,11161,
653      11171,11173,11177,11197,11213,11239,11243,11251,
654      11257,11261,11273,11279,11287,11299,11311,11317,
655      11321,11329,11351,11353,11369,11383,11393,11399,
656      11411,11423,11437,11443,11447,11467,11471,11483,
657      11489,11491,11497,11503,11519,11527,11549,11551,
658      11579,11587,11593,11597,11617,11621,11633,11657,
659      11677,11681,11689,11699,11701,11717,11719,11731,
660      11743,11777,11779,11783,11789,11801,11807,11813,
661      11821,11827,11831,11833,11839,11863,11867,11887,
662      11897,11903,11909,11923,11927,11933,11939,11941,
663      11953,11959,11969,11971,11981,11987,12007,12011,
664      12037,12041,12043,12049,12071,12073,12097,12101,
665      12107,12109,12113,12119,12143,12149,12157,12161,
666      12163,12197,12203,12211,12227,12239,12241,12251,
667      12253,12263,12269,12277,12281,12289,12301,12323,
668      12329,12343,12347,12373,12377,12379,12391,12401,
669      12409,12413,12421,12433,12437,12451,12457,12473,
670      12479,12487,12491,12497,12503,12511,12517,12527,
671      12539,12541,12547,12553,12569,12577,12583,12589,
672      12601,12611,12613,12619,12637,12641,12647,12653,
673      12659,12671,12689,12697,12703,12713,12721,12739,
674      12743,12757,12763,12781,12791,12799,12809,12821,
675      12823,12829,12841,12853,12889,12893,12899,12907,
676      12911,12917,12919,12923,12941,12953,12959,12967,
677      12973,12979,12983,13001,13003,13007,13009,13033,
678      13037,13043,13049,13063,13093,13099,13103,13109,
679      13121,13127,13147,13151,13159,13163,13171,13177,
680      13183,13187,13217,13219,13229,13241,13249,13259,
681      13267,13291,13297,13309,13313,13327,13331,13337,
682      13339,13367,13381,13397,13399,13411,13417,13421,
683      13441,13451,13457,13463,13469,13477,13487,13499,
684      13513,13523,13537,13553,13567,13577,13591,13597,
685      13613,13619,13627,13633,13649,13669,13679,13681,
686      13687,13691,13693,13697,13709,13711,13721,13723,
687      13729,13751,13757,13759,13763,13781,13789,13799,
688      13807,13829,13831,13841,13859,13873,13877,13879,
689      13883,13901,13903,13907,13913,13921,13931,13933,
690      13963,13967,13997,13999,14009,14011,14029,14033,
691      14051,14057,14071,14081,14083,14087,14107,14143,
692      14149,14153,14159,14173,14177,14197,14207,14221,
693      14243,14249,14251,14281,14293,14303,14321,14323,
694      14327,14341,14347,14369,14387,14389,14401,14407,
695      14411,14419,14423,14431,14437,14447,14449,14461,
696      14479,14489,14503,14519,14533,14537,14543,14549,
697      14551,14557,14561,14563,14591,14593,14621,14627,
698      14629,14633,14639,14653,14657,14669,14683,14699,
699      14713,14717,14723,14731,14737,14741,14747,14753,
700      14759,14767,14771,14779,14783,14797,14813,14821,
701      14827,14831,14843,14851,14867,14869,14879,14887,
702      14891,14897,14923,14929,14939,14947,14951,14957,
```

```
703       14969,14983,15013,15017,15031,15053,15061,15073,
704       15077,15083,15091,15101,15107,15121,15131,15137,
705       15139,15149,15161,15173,15187,15193,15199,15217,
706       15227,15233,15241,15259,15263,15269,15271,15277,
707       15287,15289,15299,15307,15313,15319,15329,15331,
708       15349,15359,15361,15373,15377,15383,15391,15401,
709       15413,15427,15439,15443,15451,15461,15467,15473,
710       15493,15497,15511,15527,15541,15551,15559,15569,
711       15581,15583,15601,15607,15619,15629,15641,15643,
712       15647,15649,15661,15667,15671,15679,15683,15727,
713       15731,15733,15737,15739,15749,15761,15767,15773,
714       15787,15791,15797,15803,15809,15817,15823,15859,
715       15877,15881,15887,15889,15901,15907,15913,15919,
716       15923,15937,15959,15971,15973,15991,16001,16007,
717       16033,16057,16061,16063,16067,16069,16073,16087,
718       16091,16097,16103,16111,16127,16139,16141,16183,
719       16187,16189,16193,16217,16223,16229,16231,16249,
720       16253,16267,16273,16301,16319,16333,16339,16349,
721       16361,16363,16369,16381,16411,16417,16421,16427,
722       16433,16447,16451,16453,16477,16481,16487,16493,
723       16519,16529,16547,16553,16561,16567,16573,16603,
724       16607,16619,16631,16633,16649,16651,16657,16661,
725       16673,16691,16693,16699,16703,16729,16741,16747,
726       16759,16763,16787,16811,16823,16829,16831,16843,
727       16871,16879,16883,16889,16901,16903,16921,16927,
728       16931,16937,16943,16963,16979,16981,16987,16993,
729       17011,17021,17027,17029,17033,17041,17047,17053,
730       17077,17093,17099,17107,17117,17123,17137,17159,
731       17167,17183,17189,17191,17203,17207,17209,17231,
732       17239,17257,17291,17293,17299,17317,17321,17327,
733       17333,17341,17351,17359,17377,17383,17387,17389,
734       17393,17401,17417,17419,17431,17443,17449,17467,
735       17471,17477,17483,17489,17491,17497,17509,17519,
736       17539,17551,17569,17573,17579,17581,17597,17599,
737       17609,17623,17627,17657,17659,17669,17681,17683,
738       17707,17713,17729,17737,17747,17749,17761,17783,
739       17789,17791,17807,17827,17837,17839,17851,17863
740   };
741   #endif
742   #endif
```

B.13 Elliptic Curve Files

B.13.1. CpriDataEcc.h

```
#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;
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.13.2. CpriDataEcc.c

Defines for the sizes of ECC parameters

```
#include    "TPMB.h"
TPM2B_BYTE_VALUE(1);
TPM2B_BYTE_VALUE(16);
TPM2B_BYTE_VALUE(2);
TPM2B_BYTE_VALUE(24);
TPM2B_BYTE_VALUE(28);
TPM2B_BYTE_VALUE(32);
TPM2B_BYTE_VALUE(4);
TPM2B_BYTE_VALUE(48);
TPM2B_BYTE_VALUE(64);
TPM2B_BYTE_VALUE(66);
TPM2B_BYTE_VALUE(8);
TPM2B_BYTE_VALUE(80);
#if defined ECC_NIST_P192 && ECC_NIST_P192 == YES
const TPM2B_24_BYTE_VALUE NIST_P192_p = {24,
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
         0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
         0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
const TPM2B_24_BYTE_VALUE NIST_P192_a = {24,
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
         0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
         0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC}};
const TPM2B_24_BYTE_VALUE NIST_P192_b = {24,
        {0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7,
         0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49,
         0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1}};
const TPM2B_24_BYTE_VALUE NIST_P192_gX = {24,
        {0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6,
         0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00,
         0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12}};
const TPM2B_24_BYTE_VALUE NIST_P192_gY = {24,
        {0x07, 0x19, 0x2B, 0x95, 0xFFC, 0x8D, 0xA7, 0x86,
         0x31, 0x01, 0x1ED, 0x6B, 0x24, 0xCD, 0xD5, 0x73,
         0xF9, 0x77, 0xA1, 0x1E, 0x79, 0x48, 0x11}};
const TPM2B_24_BYTE_VALUE NIST_P192_n = {24,
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
         0xFF, 0xFF, 0xFF, 0xFF, 0x99, 0xDE, 0xF8, 0x36,
         0x14, 0x6B, 0xC9, 0xB1, 0xB4, 0xD2, 0x28, 0x31}};
const TPM2B_1_BYTE_VALUE NIST_P192_h = {1,{1}};
const ECC_CURVE_DATA NIST_P192 = {&NIST_P192_p.b, &NIST_P192_a.b, &NIST_P192_b.b,
                                  &NIST_P192_gX.b, &NIST_P192_gY.b, &NIST_P192_n.b,
                                  &NIST_P192_h.b};
#endif // ECC_NIST_P192
#if defined ECC_NIST_P224 && ECC_NIST_P224 == YES
const TPM2B_28_BYTE_VALUE NIST_P224_p = {28,
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
         0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00, 0x01}};
const TPM2B_28_BYTE_VALUE NIST_P224_a = {28,
        {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
         0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
         0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
         0xFF, 0xFF, 0xFF, 0xFE}};
const TPM2B_28_BYTE_VALUE NIST_P224_b = {28,
        {0xB4, 0x05, 0x0A, 0x85, 0x0C, 0x04, 0xB3, 0xAB,
         0xF5, 0x41, 0x32, 0x56, 0x50, 0x44, 0xB0, 0xB7,
         0xD7, 0xBF, 0xD8, 0xBA, 0x27, 0x0B, 0x39, 0x43,
         0x23, 0x55, 0xFF, 0xB4}};
const TPM2B_28_BYTE_VALUE NIST_P224_gX = {28,
        {0xB7, 0x0E, 0x0C, 0xBD, 0x6B, 0xB4, 0xBF, 0x7F,
```

```
62            0x32, 0x13, 0x90, 0xB9, 0x4A, 0x03, 0xC1, 0xD3,
63            0x56, 0xC2, 0x11, 0x22, 0x34, 0x32, 0x80, 0xD6,
64            0x11, 0x5C, 0x1D, 0x21}};
65    const TPM2B_28_BYTE_VALUE NIST_P224_gY = {28,
66           {0xBD, 0x37, 0x63, 0x88, 0xB5, 0xF7, 0x23, 0xFB,
67            0x4C, 0x22, 0xDF, 0xE6, 0xCD, 0x43, 0x75, 0xA0,
68            0x5A, 0x07, 0x47, 0x64, 0x44, 0xD5, 0x81, 0x99,
69            0x85, 0x00, 0x7E, 0x34}};
70    const TPM2B_28_BYTE_VALUE NIST_P224_n = {28,
71           {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
72            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x16, 0xA2,
73            0xE0, 0xB8, 0xF0, 0x3E, 0x13, 0xDD, 0x29, 0x45,
74            0x5C, 0x5C, 0x2A, 0x3D}};
75    const TPM2B_1_BYTE_VALUE NIST_P224_h = {1,{1}};
76    const ECC_CURVE_DATA NIST_P224 = {&NIST_P224_p.b, &NIST_P224_a.b, &NIST_P224_b.b,
77                                      &NIST_P224_gX.b, &NIST_P224_gY.b, &NIST_P224_n.b,
78                                      &NIST_P224_h.b};
79    #endif // ECC_NIST_P224
80    #if defined ECC_NIST_P256 && ECC_NIST_P256 == YES
81    const TPM2B_32_BYTE_VALUE NIST_P256_p = {32,
82           {0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01,
83            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
84            0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
85            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
86    const TPM2B_32_BYTE_VALUE NIST_P256_a = {32,
87           {0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01,
88            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
89            0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
90            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC}};
91    const TPM2B_32_BYTE_VALUE NIST_P256_b = {32,
92           {0x5A, 0xC6, 0x35, 0xD8, 0xAA, 0x3A, 0x93, 0xE7,
93            0xB3, 0xEB, 0xBD, 0x55, 0x76, 0x98, 0x86, 0xBC,
94            0x65, 0x1D, 0x06, 0xB0, 0xCC, 0x53, 0xB0, 0xF6,
95            0x3B, 0xCE, 0x3C, 0x3E, 0x27, 0xD2, 0x60, 0x4B}};
96    const TPM2B_32_BYTE_VALUE NIST_P256_gX = {32,
97           {0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47,
98            0xF8, 0xBC, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2,
99            0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0,
100           0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96}};
101   const TPM2B_32_BYTE_VALUE NIST_P256_gY = {32,
102          {0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B,
103           0x8E, 0xE7, 0xEB, 0x4A, 0x7C, 0x0F, 0x9E, 0x16,
104           0x2B, 0xCE, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE,
105           0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5}};
106   const TPM2B_32_BYTE_VALUE NIST_P256_n = {32,
107          {0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
108           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
109           0xBC, 0xE6, 0xFA, 0xAD, 0xA7, 0x17, 0x9E, 0x84,
110           0xF3, 0xB9, 0xCA, 0xC2, 0xFC, 0x63, 0x25, 0x51}};
111   const TPM2B_1_BYTE_VALUE NIST_P256_h = {1,{1}};
112   const ECC_CURVE_DATA NIST_P256 = {&NIST_P256_p.b, &NIST_P256_a.b, &NIST_P256_b.b,
113                                     &NIST_P256_gX.b, &NIST_P256_gY.b, &NIST_P256_n.b,
114                                     &NIST_P256_h.b};
115   #endif // ECC_NIST_P256
116   #if defined ECC_NIST_P384 && ECC_NIST_P384 == YES
117   const TPM2B_48_BYTE_VALUE NIST_P384_p = {48,
118          {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
119           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
120           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
121           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
122           0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
123           0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF}};
124   const TPM2B_48_BYTE_VALUE NIST_P384_a = {48,
125          {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
126           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
127           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
```

```
128             0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
129             0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
130             0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFC}};
131     const TPM2B_48_BYTE_VALUE NIST_P384_b = {48,
132             {0xB3, 0x31, 0x2F, 0xA7, 0xE2, 0x3E, 0xE7, 0xE4,
133              0x98, 0x8E, 0x05, 0x6B, 0xE3, 0xF8, 0x2D, 0x19,
134              0x18, 0x1D, 0x9C, 0x6E, 0xFE, 0x81, 0x41, 0x12,
135              0x03, 0x14, 0x08, 0x8F, 0x50, 0x13, 0x87, 0x5A,
136              0xC6, 0x56, 0x39, 0x8D, 0x8A, 0x2E, 0xD1, 0x9D,
137              0x2A, 0x85, 0xC8, 0xED, 0xD3, 0xEC, 0x2A, 0xEF}};
138     const TPM2B_48_BYTE_VALUE NIST_P384_gX = {48,
139             {0xAA, 0x87, 0xCA, 0x22, 0xBE, 0x8B, 0x05, 0x37,
140              0x8E, 0xB1, 0xC7, 0x1E, 0xF3, 0x20, 0xAD, 0x74,
141              0x6E, 0x1D, 0x3B, 0x62, 0x8B, 0xA7, 0x9B, 0x98,
142              0x59, 0xF7, 0x41, 0xE0, 0x82, 0x54, 0x2A, 0x38,
143              0x55, 0x02, 0xF2, 0x5D, 0xBF, 0x55, 0x29, 0x6C,
144              0x3A, 0x54, 0x5E, 0x38, 0x72, 0x76, 0x0A, 0xB7}};
145     const TPM2B_48_BYTE_VALUE NIST_P384_gY = {48,
146             {0x36, 0x17, 0xDE, 0x4A, 0x96, 0x26, 0x2C, 0x6F,
147              0x5D, 0x9E, 0x98, 0xBF, 0x92, 0x92, 0xDC, 0x29,
148              0xF8, 0xF4, 0x1D, 0xBD, 0x28, 0x9A, 0x14, 0x7C,
149              0xE9, 0xDA, 0x31, 0x13, 0xB5, 0xF0, 0xB8, 0xC0,
150              0x0A, 0x60, 0xB1, 0xCE, 0x1D, 0x7E, 0x81, 0x9D,
151              0x7A, 0x43, 0x1D, 0x7C, 0x90, 0xEA, 0x0E, 0x5F}};
152     const TPM2B_48_BYTE_VALUE NIST_P384_n = {48,
153             {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
154              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
155              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
156              0xC7, 0x63, 0x4D, 0x81, 0xF4, 0x37, 0x2D, 0xDF,
157              0x58, 0x1A, 0x0D, 0xB2, 0x48, 0xB0, 0xA7, 0x7A,
158              0xEC, 0xEC, 0x19, 0x6A, 0xCC, 0xC5, 0x29, 0x73}};
159     const TPM2B_1_BYTE_VALUE NIST_P384_h = {1,{1}};
160     const ECC_CURVE_DATA NIST_P384 = {&NIST_P384_p.b, &NIST_P384_a.b, &NIST_P384_b.b,
161                                       &NIST_P384_gX.b, &NIST_P384_gY.b, &NIST_P384_n.b,
162                                       &NIST_P384_h.b};
163     #endif // ECC_NIST_P384
164     #if defined ECC_NIST_P521 && ECC_NIST_P521 == YES
165     const TPM2B_66_BYTE_VALUE NIST_P521_p = {66,
166             {0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
167              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
168              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
169              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
170              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
171              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
172              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
173              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
174              0xFF, 0xFF}};
175     const TPM2B_66_BYTE_VALUE NIST_P521_a = {66,
176             {0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
177              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
178              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
179              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
180              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
181              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
182              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
183              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
184              0xFF, 0xFC}};
185     const TPM2B_66_BYTE_VALUE NIST_P521_b = {66,
186             {0x00, 0x51, 0x95, 0x3E, 0xB9, 0x61, 0x8E, 0x1C,
187              0x9A, 0x1F, 0x92, 0x9A, 0x21, 0xA0, 0xB6, 0x85,
188              0x40, 0xEE, 0xA2, 0xDA, 0x72, 0x5B, 0x99, 0xB3,
189              0x15, 0xF3, 0xB8, 0xB4, 0x89, 0x91, 0x8E, 0xF1,
190              0x09, 0xE1, 0x56, 0x19, 0x39, 0x51, 0xEC, 0x7E,
191              0x93, 0x7B, 0x16, 0x52, 0xC0, 0xBD, 0x3B, 0xB1,
192              0xBF, 0x07, 0x35, 0x73, 0xDF, 0x88, 0x3D, 0x2C,
193              0x34, 0xF1, 0xEF, 0x45, 0x1F, 0xD4, 0x6B, 0x50,
```

```
194             0x3F, 0x00}};
195     const TPM2B_66_BYTE_VALUE NIST_P521_gX = {66,
196             {0x00, 0xC6, 0x85, 0x8E, 0x06, 0xB7, 0x04, 0x04,
197              0xE9, 0xCD, 0x9E, 0x3E, 0xCB, 0x66, 0x23, 0x95,
198              0xB4, 0x42, 0x9C, 0x64, 0x81, 0x39, 0x05, 0x3F,
199              0xB5, 0x21, 0xF8, 0x28, 0xAF, 0x60, 0x6B, 0x4D,
200              0x3D, 0xBA, 0xA1, 0x4B, 0x5E, 0x77, 0xEF, 0xE7,
201              0x59, 0x28, 0xFE, 0x1D, 0xC1, 0x27, 0xA2, 0xFF,
202              0xA8, 0xDE, 0x33, 0x48, 0xB3, 0xC1, 0x85, 0x6A,
203              0x42, 0x9B, 0xF9, 0x7E, 0x7E, 0x31, 0xC2, 0xE5,
204              0xBD, 0x66}};
205     const TPM2B_66_BYTE_VALUE NIST_P521_gY = {66,
206             {0x01, 0x18, 0x39, 0x29, 0x6A, 0x78, 0x9A, 0x3B,
207              0xC0, 0x04, 0x5C, 0x8A, 0x5F, 0xB4, 0x2C, 0x7D,
208              0x1B, 0xD9, 0x98, 0xF5, 0x44, 0x49, 0x57, 0x9B,
209              0x44, 0x68, 0x17, 0xAF, 0xBD, 0x17, 0x27, 0x3E,
210              0x66, 0x2C, 0x97, 0xEE, 0x72, 0x99, 0x5E, 0xF4,
211              0x26, 0x40, 0xC5, 0x50, 0xB9, 0x01, 0x3F, 0xAD,
212              0x07, 0x61, 0x35, 0x3C, 0x70, 0x86, 0xA2, 0x72,
213              0xC2, 0x40, 0x88, 0xBE, 0x94, 0x76, 0x9F, 0xD1,
214              0x66, 0x50}};
215     const TPM2B_66_BYTE_VALUE NIST_P521_n = {66,
216             {0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
217              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
218              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
219              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
220              0xFF, 0xFA, 0x51, 0x86, 0x87, 0x83, 0xBF, 0x2F,
221              0x96, 0x6B, 0x7F, 0xCC, 0x01, 0x48, 0xF7, 0x09,
222              0xA5, 0xD0, 0x3B, 0xB5, 0xC9, 0xB8, 0x89, 0x9C,
223              0x47, 0xAE, 0xBB, 0x6F, 0xB7, 0x1E, 0x91, 0x38,
224              0x64, 0x09}};
225     const TPM2B_1_BYTE_VALUE NIST_P521_h = {1,{1}};
226     const ECC_CURVE_DATA NIST_P521 = {&NIST_P521_p.b, &NIST_P521_a.b, &NIST_P521_b.b,
227                                       &NIST_P521_gX.b, &NIST_P521_gY.b, &NIST_P521_n.b,
228                                       &NIST_P521_h.b};
229     #endif // ECC_NIST_P521
230     #if defined ECC_BN_P256 && ECC_BN_P256 == YES
231     const TPM2B_32_BYTE_VALUE BN_P256_p = {32,
232             {0xFF, 0XFF, 0XFF, 0XFF, 0XFF, 0XFC, 0XF0, 0XCD,
233              0X46, 0XE5, 0XF2, 0X5E, 0XEE, 0X71, 0XA4, 0X9F,
234              0X0C, 0XDC, 0X65, 0XFB, 0X12, 0X98, 0X0A, 0X82,
235              0XD3, 0X29, 0X2D, 0XDB, 0XAE, 0XD3, 0X30, 0X13}};
236     const TPM2B_1_BYTE_VALUE BN_P256_a = {1,{0}};
237     const TPM2B_1_BYTE_VALUE BN_P256_b = {1,{3}};
238     const TPM2B_1_BYTE_VALUE BN_P256_gX = {1,{1}};
239     const TPM2B_1_BYTE_VALUE BN_P256_gY = {1,{2}};;
240     const TPM2B_32_BYTE_VALUE BN_P256_n = {32,
241             {0xFF, 0XFF, 0XFF, 0XFF, 0XFF, 0XFC, 0XF0, 0XCD,
242              0X46, 0XE5, 0XF2, 0X5E, 0XEE, 0X71, 0XA4, 0X9E,
243              0X0C, 0XDC, 0X65, 0XFB, 0X12, 0X99, 0X92, 0X1A,
244              0XF6, 0X2D, 0X53, 0X6C, 0XD1, 0X0B, 0X50, 0X0D}};
245     const TPM2B_1_BYTE_VALUE BN_P256_h = {1,{1}};
246     const ECC_CURVE_DATA BN_P256 = {&BN_P256_p.b, &BN_P256_a.b, &BN_P256_b.b,
247                                     &BN_P256_gX.b, &BN_P256_gY.b, &BN_P256_n.b,
248                                     &BN_P256_h.b};
249     #endif // ECC_BN_P256
250     #if defined ECC_BN_P638 && ECC_BN_P638 == YES
251     const TPM2B_80_BYTE_VALUE BN_P638_p = {80,
252             {0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D,
253              0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3,
254              0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E,
255              0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F,
256              0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55,
257              0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B,
258              0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80,
259              0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD,
```

```
260             0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0,
261             0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67}};
262     const TPM2B_1_BYTE_VALUE BN_P638_a = {1,{0}};
263     const TPM2B_2_BYTE_VALUE BN_P638_b = {2,{0x01,0x01}};
264     const TPM2B_80_BYTE_VALUE BN_P638_gX = {80,
265             {0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D,
266              0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3,
267              0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E,
268              0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F,
269              0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55,
270              0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B,
271              0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80,
272              0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD,
273              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0,
274              0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66}};
275     const TPM2B_1_BYTE_VALUE BN_P638_gY = {1,{0x10}};
276     const TPM2B_80_BYTE_VALUE BN_P638_n = {80,
277             {0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D,
278              0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3,
279              0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E,
280              0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F,
281              0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55,
282              0x60, 0x00, 0x86, 0x55, 0x00, 0x21, 0xE5, 0x55,
283              0xFF, 0xFF, 0xF5, 0x4F, 0xFF, 0xF4, 0xEA, 0xC0,
284              0x00, 0x00, 0x00, 0x49, 0x80, 0x01, 0x54, 0xD9,
285              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xA0,
286              0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61}};
287     const TPM2B_1_BYTE_VALUE BN_P638_h = {1,{1}};
288     const ECC_CURVE_DATA BN_P638 = {&BN_P638_p.b, &BN_P638_a.b, &BN_P638_b.b,
289                                     &BN_P638_gX.b, &BN_P638_gY.b, &BN_P638_n.b,
290                                     &BN_P638_h.b};
291     #endif // ECC_BN_P638
292     #if defined ECC_SM2_P256 && ECC_SM2_P256 == YES
293     const TPM2B_32_BYTE_VALUE SM2_P256_p = {32,
294             {0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF,
295              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
296              0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
297              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
298     const TPM2B_32_BYTE_VALUE SM2_P256_a = {32,
299             {0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF,
300              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
301              0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
302              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC}};
303     const TPM2B_32_BYTE_VALUE SM2_P256_b = {32,
304             {0x28, 0xE9, 0xFA, 0x9E, 0x9D, 0x9F, 0x5E, 0x34,
305              0x4D, 0x5A, 0x9E, 0x4B, 0xCF, 0x65, 0x09, 0xA7,
306              0xF3, 0x97, 0x89, 0xF5, 0x15, 0xAB, 0x8F, 0x92,
307              0xDD, 0xBC, 0xBD, 0x41, 0x4D, 0x94, 0x0E, 0x93}};
308     const TPM2B_32_BYTE_VALUE SM2_P256_gX = {32,
309             {0x32, 0xC4, 0xAE, 0x2C, 0x1F, 0x19, 0x81, 0x19,
310              0x5F, 0x99, 0x04, 0x46, 0x6A, 0x39, 0xC9, 0x94,
311              0x8F, 0xE3, 0x0B, 0xBF, 0xF2, 0x66, 0x0B, 0xE1,
312              0x71, 0x5A, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7}};
313     const TPM2B_32_BYTE_VALUE SM2_P256_gY = {32,
314             {0xBC, 0x37, 0x36, 0xA2, 0xF4, 0xF6, 0x77, 0x9C,
315              0x59, 0xBD, 0xCE, 0xE3, 0x6B, 0x69, 0x21, 0x53,
316              0xD0, 0xA9, 0x87, 0x7C, 0xC6, 0x2A, 0x47, 0x40,
317              0x02, 0xDF, 0x32, 0xE5, 0x21, 0x39, 0xF0, 0xA0}};
318     const TPM2B_32_BYTE_VALUE SM2_P256_n = {32,
319             {0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF,
320              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
321              0x72, 0x03, 0xDF, 0x6B, 0x21, 0xC6, 0x05, 0x2B,
322              0x53, 0xBB, 0xF4, 0x09, 0x39, 0xD5, 0x41, 0x23}};
323     const TPM2B_1_BYTE_VALUE SM2_P256_h = {1,{1}};
324     const ECC_CURVE_DATA SM2_P256 = {&SM2_P256_p.b, &SM2_P256_a.b, &SM2_P256_b.b,
325                                      &SM2_P256_gX.b, &SM2_P256_gY.b, &SM2_P256_n.b,
```

```
326                                     &SM2_P256_h.b};
327 #endif // ECC_SM2_P256
328 #define comma
329 const ECC_CURVE   eccCurves[] = {
330 #if defined ECC_NIST_P192 && ECC_NIST_P192 == YES
331     comma
332     {TPM_ECC_NIST_P192,
333     192,
334     {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA256},
335     {TPM_ALG_NULL,TPM_ALG_NULL},
336     &NIST_P192}
337 #   undef comma
338 #   define comma ,
339 #endif // ECC_NIST_P192
340 #if defined ECC_NIST_P224 && ECC_NIST_P224 == YES
341     comma
342     {TPM_ECC_NIST_P224,
343     224,
344     {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA256},
345     {TPM_ALG_NULL,TPM_ALG_NULL},
346     &NIST_P224}
347 #   undef comma
348 #   define comma ,
349 #endif // ECC_NIST_P224
350 #if defined ECC_NIST_P256 && ECC_NIST_P256 == YES
351     comma
352     {TPM_ECC_NIST_P256,
353     256,
354     {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA256},
355     {TPM_ALG_NULL,TPM_ALG_NULL},
356     &NIST_P256}
357 #   undef comma
358 #   define comma ,
359 #endif // ECC_NIST_P256
360 #if defined ECC_NIST_P384 && ECC_NIST_P384 == YES
361     comma
362     {TPM_ECC_NIST_P384,
363     384,
364     {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA384},
365     {TPM_ALG_NULL,TPM_ALG_NULL},
366     &NIST_P384}
367 #   undef comma
368 #   define comma ,
369 #endif // ECC_NIST_P384
370 #if defined ECC_NIST_P521 && ECC_NIST_P521 == YES
371     comma
372     {TPM_ECC_NIST_P521,
373     521,
374     {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA512},
375     {TPM_ALG_NULL,TPM_ALG_NULL},
376     &NIST_P521}
377 #   undef comma
378 #   define comma ,
379 #endif // ECC_NIST_P521
380 #if defined ECC_BN_P256 && ECC_BN_P256 == YES
381     comma
382     {TPM_ECC_BN_P256,
383     256,
384     {TPM_ALG_NULL,TPM_ALG_NULL},
385     {TPM_ALG_NULL,TPM_ALG_NULL},
386     &BN_P256}
387 #   undef comma
388 #   define comma ,
389 #endif // ECC_BN_P256
390 #if defined ECC_BN_P638 && ECC_BN_P638 == YES
391     comma
```

```
392         {TPM_ECC_BN_P638,
393         638,
394         {TPM_ALG_NULL,TPM_ALG_NULL},
395         {TPM_ALG_NULL,TPM_ALG_NULL},
396         &BN_P638}
397     #   undef comma
398     #   define comma ,
399     #endif // ECC_BN_P638
400     #if defined ECC_SM2_P256 && ECC_SM2_P256 == YES
401         comma
402         {TPM_ECC_SM2_P256,
403         256,
404         {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SM3_256},
405         {TPM_ALG_NULL,TPM_ALG_NULL},
406         &SM2_P256}
407     #   undef comma
408     #   define comma ,
409     #endif // ECC_SM2_P256
410     };
411     const UINT16    ECC_CURVE_COUNT = sizeof(eccCurves) / sizeof(ECC_CURVE);
```

B.13.3. CpriECC.c

B.13.3.1. Includes and Defines

Need to include OsslCryptEngine.h to determine if ECC is defined for this Implementation

```
#include    "OsslCryptoEngine.h"
#ifdef  TPM_ALG_ECC
#include    "CpriDataEcc.h"
#include    "CpriDataEcc.c"
```

B.13.3.2. Functions

B.13.3.2.1. _cpri__EccStartup()

This function is called at TPM Startup to initialize the crypto units.

In this implementation, no initialization is performed at startup but a future version may initialize the self-test functions here.

```
LIB_EXPORT BOOL
_cpri__EccStartup(
    void
    )
{
    return TRUE;
}
```

B.13.3.2.2. _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.

```
LIB_EXPORT TPM_ECC_CURVE
_cpri__GetCurveIdByIndex(
    UINT16          i
    )
{
    if(i >= ECC_CURVE_COUNT)
        return TPM_ECC_NONE;
    return eccCurves[i].curveId;
}
LIB_EXPORT UINT32
_cpri__EccGetCurveCount(
    void
    )
{
    return ECC_CURVE_COUNT;
}
```

B.13.3.2.3. _cpri__EccGetParametersByCurveId()

This function returns a pointer to the curve data that is associated with the indicated *curveId*. If there is no curve with the indicated ID, the function returns NULL.

Return Value	Meaning
NULL	curve with the indicated TPM_ECC_CURVE value is not implemented
non-NULL	pointer to the curve data

```
28  LIB_EXPORT const ECC_CURVE *
29  _cpri__EccGetParametersByCurveId(
30      TPM_ECC_CURVE     curveId         // IN: the curveID
31      )
32  {
33      int           i;
34      for(i = 0; i < ECC_CURVE_COUNT; i++)
35      {
36          if(eccCurves[i].curveId == curveId)
37              return &eccCurves[i];
38      }
39      FAIL(FATAL_ERROR_INTERNAL);
40  }
41  static const ECC_CURVE_DATA *
42  GetCurveData(
43      TPM_ECC_CURVE     curveId         // IN: the curveID
44      )
45  {
46      const ECC_CURVE     *curve = _cpri__EccGetParametersByCurveId(curveId);
47      return curve->curveData;
48  }
```

B.13.3.2.4. Point2B()

This function makes a TPMS_ECC_POINT from a BIGNUM EC_POINT.

```
49  static BOOL
50  Point2B(
51      EC_GROUP        *group,         // IN: group for the point
52      TPMS_ECC_POINT  *p,             // OUT: receives the converted point
53      EC_POINT        *ecP,           // IN: the point to convert
54      INT16            size,          // IN: size of the coordinates
55      BN_CTX          *context        // IN: working context
56      )
57  {
58      BIGNUM          *bnX;
59      BIGNUM          *bnY;
60
61      BN_CTX_start(context);
62      bnX = BN_CTX_get(context);
63      bnY = BN_CTX_get(context);
64
65      if(     bnY == NULL
66
67          // Get the coordinate values
68         ||  EC_POINT_get_affine_coordinates_GFp(group, ecP, bnX, bnY, context) != 1
69
70         // Convert x
71         ||  (!BnTo2B(&p->x.b, bnX, size))
72
73         // Convert y
74         ||  (!BnTo2B(&p->y.b, bnY, size))
75        )
76              FAIL(FATAL_ERROR_INTERNAL);
77
78      BN_CTX_end(context);
79      return TRUE;
```

```
}
```

B.13.3.2.5. EccCurveInit()

This function initializes the OpenSSL() group definition structure

This function is only used within this file.

It is a fatal error if *groupContext* is not provided.

Return Value	Meaning
NULL	the TPM_ECC_CURVE is not valid
non-NULL	points to a structure in *groupContext* static EC_GROUP *

```
static EC_GROUP *
EccCurveInit(
    TPM_ECC_CURVE    curveId,       // IN: the ID of the curve
    BN_CTX          *groupContext   // IN: the context in which the group is to be
                                    //     created
    )
{
    const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
    EC_GROUP                *group = NULL;
    EC_POINT                *P = NULL;
    BN_CTX                  *context;
    BIGNUM                  *bnP;
    BIGNUM                  *bnA;
    BIGNUM                  *bnB;
    BIGNUM                  *bnX;
    BIGNUM                  *bnY;
    BIGNUM                  *bnN;
    BIGNUM                  *bnH;
    int                      ok = FALSE;

    // Context must be provided and curve selector must be valid
    pAssert(groupContext != NULL &&  curveData != NULL);

    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);

    BN_CTX_start(context);
    bnP = BN_CTX_get(context);
    bnA = BN_CTX_get(context);
    bnB = BN_CTX_get(context);
    bnX = BN_CTX_get(context);
    bnY = BN_CTX_get(context);
    bnN = BN_CTX_get(context);
    bnH = BN_CTX_get(context);

    if (bnH == NULL)
        goto Cleanup;

    // Convert the number formats

    BnFrom2B(bnP, curveData->p);
    BnFrom2B(bnA, curveData->a);
    BnFrom2B(bnB, curveData->b);
    BnFrom2B(bnX, curveData->x);
    BnFrom2B(bnY, curveData->y);
    BnFrom2B(bnN, curveData->n);
    BnFrom2B(bnH, curveData->h);

```

```
130         // initialize EC group, associate a generator point and initialize the point
131         // from the parameter data
132         ok = (   (group = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, groupContext)) != NULL
133               && (P = EC_POINT_new(group)) != NULL
134               && EC_POINT_set_affine_coordinates_GFp(group, P, bnX, bnY, groupContext)
135               && EC_GROUP_set_generator(group, P, bnN, bnH)
136              );
137     Cleanup:
138         if (!ok && group != NULL)
139         {
140             EC_GROUP_free(group);
141             group = NULL;
142         }
143         if(P != NULL)
144             EC_POINT_free(P);
145         BN_CTX_end(context);
146         BN_CTX_free(context);
147         return group;
148     }
```

B.13.3.2.6. PointFrom2B()

This function sets the coordinates of an existing BN Point from a TPMS_ECC_POINT.

```
149     static EC_POINT *
150     PointFrom2B(
151         EC_GROUP        *group,         // IN: the group for the point
152         EC_POINT        *ecP,           // IN: an existing BN point in the group
153         TPMS_ECC_POINT  *p,             // IN: the 2B coordinates of the point
154         BN_CTX          *context        // IN: the BIGNUM context
155         )
156     {
157         BIGNUM          *bnX;
158         BIGNUM          *bnY;
159
160         // If the point is not allocated then just return a NULL
161         if(ecP == NULL)
162             return NULL;
163
164         BN_CTX_start(context);
165         bnX = BN_CTX_get(context);
166         bnY = BN_CTX_get(context);
167         if( // Set the coordinates of the point
168               bnY == NULL
169            || BN_bin2bn(p->x.t.buffer, p->x.t.size, bnX) == NULL
170            || BN_bin2bn(p->y.t.buffer, p->y.t.size, bnY) == NULL
171            || !EC_POINT_set_affine_coordinates_GFp(group, ecP, bnX, bnY, context)
172            )
173            FAIL(FATAL_ERROR_INTERNAL);
174
175         BN_CTX_end(context);
176         return ecP;
177     }
```

B.13.3.2.7. EccInitPoint2B()

This function allocates a point in the provided group and initializes it with the values in a TPMS_ECC_POINT.

```
178     static EC_POINT *
179     EccInitPoint2B(
180         EC_GROUP        *group,         // IN: group for the point
181         TPMS_ECC_POINT  *p,             // IN: the coordinates for the point
```

```
182         BN_CTX          *context        // IN: the BIGNUM context
183         )
184     {
185         EC_POINT        *ecP;
186
187         BN_CTX_start(context);
188         ecP = EC_POINT_new(group);
189
190         if(PointFrom2B(group, ecP, p, context) == NULL)
191             FAIL(FATAL_ERROR_INTERNAL);
192
193         BN_CTX_end(context);
194         return ecP;
195     }
```

B.13.3.2.8. PointMul()

This function does a point multiply and checks for the result being the point at infinity. **Q** = ([A]G + [B]P)

Return Value	Meaning
CRYPT_NO_RESULT	point is at infinity
CRYPT_SUCCESS	point not at infinity

```
196     static CRYPT_RESULT
197     PointMul(
198         EC_GROUP        *group,         // IN: group curve
199         EC_POINT        *ecpQ,          // OUT: result
200         BIGNUM          *bnA,           // IN: scalar for [A]G
201         EC_POINT        *ecpP,          // IN: point for [B]P
202         BIGNUM          *bnB,           // IN: scalar for [B]P
203         BN_CTX          *context        // IN: working context
204         )
205     {
206            if(EC_POINT_mul(group, ecpQ, bnA, ecpP, bnB, context) != 1)
207                FAIL(FATAL_ERROR_INTERNAL);
208             if(EC_POINT_is_at_infinity(group, ecpQ))
209                 return CRYPT_NO_RESULT;
210             return CRYPT_SUCCESS;
211     }
```

B.13.3.2.9. 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 0 < *d* < *n*.

It is a fatal error if *dOut* or *pIn* is not provided or if the size of *pIn* is larger than MAX_ECC_KEY_BYTES (the largest buffer size of a TPM2B_ECC_PARAMETER)

```
212     static void
213     GetRandomPrivate(
214         TPM2B_ECC_PARAMETER     *dOut,          // OUT: the qualified random value
215         const TPM2B             *pIn            // IN: the maximum value for the key
216         )
217     {
218         int          i;
219         BYTE        *pb;
220
221         pAssert(pIn != NULL && dOut != NULL && pIn->size <= MAX_ECC_KEY_BYTES);
222
223         // Set the size of the output
224         dOut->t.size = pIn->size;
```

```
225         // Get some random bits
226         while(TRUE)
227         {
228             _cpri__GenerateRandom(dOut->t.size, dOut->t.buffer);
229             // See if the d < n
230             if(memcmp(dOut->t.buffer, pIn->buffer, pIn->size) < 0)
231             {
232                 // dOut < n so make sure that 0 < dOut
233                 for(pb = dOut->t.buffer, i = dOut->t.size; i > 0; i--)
234                 {
235                     if(*pb++ != 0)
236                         return;
237                 }
238             }
239         }
240     }
```

B.13.3.2.10. Mod2B()

Function does modular reduction of TPM2B values.

```
241     static CRYPT_RESULT
242     Mod2B(
243         TPM2B           *x,             // IN/OUT: value to reduce
244         const TPM2B     *n              // IN: mod
245         )
246     {
247         int         compare;
248         compare = _math__uComp(x->size, x->buffer, n->size, n->buffer);
249         if(compare < 0)
250             // if x < n, then mod is x
251             return CRYPT_SUCCESS;
252         if(compare == 0)
253         {
254             // if x == n then mod is 0
255             x->size = 0;
256             x->buffer[0] = 0;
257             return CRYPT_SUCCESS;
258         }
259        return _math__Div(x, n, NULL, x);
260     }
```

B.13.3.2.11. _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.

Return Value	Meaning
CRYPT_SUCCESS	point multiplication succeeded
CRYPT_POINT	the point *Qin* is not on the curve
CRYPT_NO_RESULT	the product point is at infinity

```
261  LIB_EXPORT CRYPT_RESULT
262  _cpri__EccPointMultiply(
263      TPMS_ECC_POINT          *Rout,          // OUT: the product point R
264      TPM_ECC_CURVE            curveId,       // IN: the curve to use
265      TPM2B_ECC_PARAMETER     *dIn,           // IN: value to multiply against the
266                                              //     curve generator
267      TPMS_ECC_POINT          *Qin,           // IN: point Q
268      TPM2B_ECC_PARAMETER     *uIn            // IN: scalar value for the multiplier
269                                              //     of Q
270      )
271  {
272      BN_CTX              *context;
273      BIGNUM              *bnD;
274      BIGNUM              *bnU;
275      EC_GROUP            *group;
276      EC_POINT            *R = NULL;
277      EC_POINT            *Q = NULL;
278      CRYPT_RESULT         retVal = CRYPT_SUCCESS;
279
280      // Validate that the required parameters are provided.
281      pAssert((dIn != NULL || uIn != NULL) && (Qin != NULL || dIn != NULL));
282
283      // If a point is provided for the multiply, make sure that it is on the curve
284      if(Qin != NULL && !_cpri__EccIsPointOnCurve(curveId, Qin))
285          return CRYPT_POINT;
286
287      context = BN_CTX_new();
288      if(context == NULL)
289          FAIL(FATAL_ERROR_ALLOCATION);
290
291      BN_CTX_start(context);
292      bnU = BN_CTX_get(context);
293      bnD = BN_CTX_get(context);
294      group = EccCurveInit(curveId, context);
295
296      // There should be no path for getting a bad curve ID into this function.
297      pAssert(group != NULL);
298
299      // check allocations should have worked and allocate R
300      if(   bnD == NULL
301         || (R = EC_POINT_new(group)) == NULL)
302          FAIL(FATAL_ERROR_ALLOCATION);
303
304      // If Qin is present, create the point
305      if(Qin != NULL)
306      {
307          // Assume the size variables do not overflow. This should not happen in
308          // the contexts in which this function will be called.
309          assert2Bsize(Qin->x.t);
310          assert2Bsize(Qin->x.t);
311          Q = EccInitPoint2B(group, Qin, context);
312
313      }
314      if(dIn != NULL)
315      {
316          // Assume the size variables do not overflow, which should not happen in
317          // the contexts that this function will be called.
318          assert2Bsize(dIn->t);
```

```
319             BnFrom2B(bnD, &dIn->b);
320         }
321         else
322             bnD = NULL;
323 
324         // If uIn is specified, initialize its BIGNUM
325         if(uIn != NULL)
326         {
327             // Assume the size variables do not overflow, which should not happen in
328             // the contexts that this function will be called.
329             assert2Bsize(uIn->t);
330             BnFrom2B(bnU, &uIn->b);
331         }
332         // If uIn is not specified but Q is, then we are going to
333         // do R = [d]Q
334         else if(Qin != NULL)
335         {
336             bnU = bnD;
337             bnD = NULL;
338         }
339         // If neither Q nor u is specified, then null this pointer
340         else
341             bnU = NULL;
342 
343         // Use the generator of the curve
344         if((retVal = PointMul(group, R, bnD, Q, bnU, context)) == CRYPT_SUCCESS)
345             Point2B(group, Rout, R, (INT16) BN_num_bytes(&group->field), context);
346 
347         if (Q)
348             EC_POINT_free(Q);
349         if(R)
350             EC_POINT_free(R);
351         if(group)
352             EC_GROUP_free(group);
353         BN_CTX_end(context);
354         BN_CTX_free(context);
355         return retVal;
356     }
```

B.13.3.2.12. ClearPoint2B()

Initialize the size values of a point

```
357     static void
358     ClearPoint2B(
359         TPMS_ECC_POINT  *p              // IN: the point
360         )
361     {
362         if(p != NULL) {
363             p->x.t.size = 0;
364             p->y.t.size = 0;
365         }
366     }
367     #if defined TPM_ALG_ECDAA || defined TPM_ALG_SM2 //%
```

B.13.3.2.13. _cpri__EccCommitCompute()

This function performs the point multiply operations required by TPM2_Commit().

If *B* or *M* is provided, they must be on the curve defined by *curveId*. This routine does not check that they are on the curve and results are unpredictable if they are not.

It is a fatal error if *r* or *d* is NULL. If *B* is not NULL, then it is a fatal error if *K* and *L* are both NULL. If *M* is not NULL, then it is a fatal error if *E* is NULL.

Return Value	Meaning
CRYPT_SUCCESS	computations completed normally
CRYPT_NO_RESULT	if *K*, *L* or *E* was computed to be the point at infinity
CRYPT_CANCEL	a cancel indication was asserted during this function

```
368 LIB_EXPORT CRYPT_RESULT
369 _cpri__EccCommitCompute(
370     TPMS_ECC_POINT          *K,             // OUT: [d]B or [r]Q
371     TPMS_ECC_POINT          *L,             // OUT: [r]B
372     TPMS_ECC_POINT          *E,             // OUT: [r]M
373     TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
374     TPMS_ECC_POINT          *M,             // IN: M (optional)
375     TPMS_ECC_POINT          *B,             // IN: B (optional)
376     TPM2B_ECC_PARAMETER     *d,             // IN: d (required)
377     TPM2B_ECC_PARAMETER     *r              // IN: the computed r value (required)
378     )
379 {
380     BN_CTX              *context;
381     BIGNUM              *bnX, *bnY, *bnR, *bnD;
382     EC_GROUP            *group;
383     EC_POINT            *pK = NULL, *pL = NULL, *pE = NULL, *pM = NULL, *pB = NULL;
384     UINT16               keySizeInBytes;
385     CRYPT_RESULT         retVal = CRYPT_SUCCESS;
386
387     // Validate that the required parameters are provided.
388     // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
389     // E := [r]Q if both M and B are NULL.
390     pAssert(   r != NULL && (K != NULL || B == NULL) && (L != NULL || B == NULL)
391             || (E != NULL || (M == NULL && B != NULL)));
392
393     context = BN_CTX_new();
394     if(context == NULL)
395         FAIL(FATAL_ERROR_ALLOCATION);
396     BN_CTX_start(context);
397     bnR = BN_CTX_get(context);
398     bnD = BN_CTX_get(context);
399     bnX = BN_CTX_get(context);
400     bnY = BN_CTX_get(context);
401     if(bnY == NULL)
402         FAIL(FATAL_ERROR_ALLOCATION);
403
404     // Initialize the output points in case they are not computed
405     ClearPoint2B(K);
406     ClearPoint2B(L);
407     ClearPoint2B(E);
408
409     if((group = EccCurveInit(curveId, context)) == NULL)
410     {
411         retVal = CRYPT_PARAMETER;
412         goto Cleanup2;
413     }
414     keySizeInBytes = (UINT16) BN_num_bytes(&group->field);
415
416     // Sizes of the r and d parameters may not be zero
417     pAssert(((int) r->t.size > 0) && ((int) d->t.size > 0));
418
419     // Convert scalars to BIGNUM
420     BnFrom2B(bnR, &r->b);
421     BnFrom2B(bnD, &d->b);
422
```

```
423         // If B is provided, compute K=[d]B and L=[r]B
424         if(B != NULL)
425         {
426             // Allocate the points to receive the value
427             if(     (pK = EC_POINT_new(group)) == NULL
428                 || (pL = EC_POINT_new(group)) == NULL)
429             FAIL(FATAL_ERROR_ALLOCATION);
430             // need to compute K = [d]B
431             // Allocate and initialize BIGNUM version of B
432             pB = EccInitPoint2B(group, B, context);
433 
434             // do the math for K = [d]B
435             if((retVal = PointMul(group, pK, NULL, pB, bnD, context)) != CRYPT_SUCCESS)
436                 goto Cleanup;
437 
438             // Convert BN K to TPM2B K
439             Point2B(group, K, pK, (INT16)keySizeInBytes, context);
440 
441             //  compute L= [r]B after checking for cancel
442             if(_plat__IsCanceled())
443             {
444                 retVal = CRYPT_CANCEL;
445                 goto Cleanup;
446             }
447             // compute L = [r]B
448             if((retVal = PointMul(group, pL, NULL, pB, bnR, context)) != CRYPT_SUCCESS)
449                 goto Cleanup;
450 
451             // Convert BN L to TPM2B L
452             Point2B(group, L, pL, (INT16)keySizeInBytes, context);
453         }
454         if(M != NULL || B == NULL)
455         {
456             // if this is the third point multiply, check for cancel first
457             if(B != NULL && _plat__IsCanceled())
458             {
459                 retVal = CRYPT_CANCEL;
460                 goto Cleanup;
461             }
462 
463             // Allocate E
464             if((pE = EC_POINT_new(group)) == NULL)
465                 FAIL(FATAL_ERROR_ALLOCATION);
466 
467             // Create BIGNUM version of M unless M is NULL
468             if(M != NULL)
469             {
470                 // M provided so initialize a BIGNUM M and compute E = [r]M
471                 pM = EccInitPoint2B(group, M, context);
472                 retVal = PointMul(group, pE, NULL, pM, bnR, context);
473             }
474             else
475                 // compute E = [r]G (this is only done if M and B are both NULL
476                 retVal = PointMul(group, pE, bnR, NULL, NULL, context);
477 
478             if(retVal == CRYPT_SUCCESS)
479                 // Convert E to 2B format
480                 Point2B(group, E, pE, (INT16)keySizeInBytes, context);
481         }
482     Cleanup:
483         EC_GROUP_free(group);
484         if(pK != NULL) EC_POINT_free(pK);
485         if(pL != NULL) EC_POINT_free(pL);
486         if(pE != NULL) EC_POINT_free(pE);
487         if(pM != NULL) EC_POINT_free(pM);
488         if(pB != NULL) EC_POINT_free(pB);
```

```
489     Cleanup2:
490         BN_CTX_end(context);
491         BN_CTX_free(context);
492         return retVal;
493     }
494     #endif  //%
```

B.13.3.2.14. _cpri__EccIsPointOnCurve()

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*x + b mod p

It is a fatal error if Q is not specified (is NULL).

Return Value	Meaning
TRUE	point is on curve
FALSE	point is not on curve or curve is not supported

```
495     LIB_EXPORT BOOL
496     _cpri__EccIsPointOnCurve(
497         TPM_ECC_CURVE    curveId,       // IN: the curve selector
498         TPMS_ECC_POINT  *Q              // IN: the point.
499         )
500     {
501         BN_CTX                  *context;
502         BIGNUM                  *bnX;
503         BIGNUM                  *bnY;
504         BIGNUM                  *bnA;
505         BIGNUM                  *bnB;
506         BIGNUM                  *bnP;
507         BIGNUM                  *bn3;
508         const  ECC_CURVE_DATA   *curveData = GetCurveData(curveId);
509         BOOL                     retVal;
510
511         pAssert(Q != NULL && curveData != NULL);
512
513         if((context = BN_CTX_new()) == NULL)
514             FAIL(FATAL_ERROR_ALLOCATION);
515         BN_CTX_start(context);
516         bnX = BN_CTX_get(context);
517         bnY = BN_CTX_get(context);
518         bnA = BN_CTX_get(context);
519         bnB = BN_CTX_get(context);
520         bn3 = BN_CTX_get(context);
521         bnP = BN_CTX_get(context);
522         if(bnP == NULL)
523             FAIL(FATAL_ERROR_ALLOCATION);
524
525         // Convert values
526         if (   !BN_bin2bn(Q->x.t.buffer, Q->x.t.size, bnX)
527             || !BN_bin2bn(Q->y.t.buffer, Q->y.t.size, bnY)
528             || !BN_bin2bn(curveData->p->buffer, curveData->p->size, bnP)
529             || !BN_bin2bn(curveData->a->buffer, curveData->a->size, bnA)
530             || !BN_set_word(bn3, 3)
531             || !BN_bin2bn(curveData->b->buffer, curveData->b->size, bnB)
532            )
533             FAIL(FATAL_ERROR_INTERNAL);
534
535         // The following sequence is probably not optimal but it seems to be correct.
536         // compute x^3 + a*x + b mod p
537                 // first, compute a*x mod p
538         if(   !BN_mod_mul(bnA, bnA, bnX, bnP, context)
```

```
539             // next, compute a*x + b mod p
540         || !BN_mod_add(bnA, bnA, bnB, bnP, context)
541             // next, compute X^3 mod p
542         || !BN_mod_exp(bnX, bnX, bn3, bnP, context)
543             // finally, compute x^3 + a*x + b mod p
544         || !BN_mod_add(bnX, bnX, bnA, bnP, context)
545             // then compute y^2
546         || !BN_mod_mul(bnY, bnY, bnY, bnP, context)
547        )
548           FAIL(FATAL_ERROR_INTERNAL);
549
550     retVal = BN_cmp(bnX, bnY) == 0;
551     BN_CTX_end(context);
552     BN_CTX_free(context);
553     return retVal;
554 }
```

B.13.3.2.15. _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 <= *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) <= *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 <= *d* < *n*

It is a fatal error if *Qout*, *dOut*, or *seed* is not provided (is NULL).

Return Value	Meaning
CRYPT_PARAMETER	the hash algorithm is not supported

```
555 LIB_EXPORT CRYPT_RESULT
556 _cpri__GenerateKeyEcc(
557     TPMS_ECC_POINT          *Qout,          // OUT: the public point
558     TPM2B_ECC_PARAMETER     *dOut,          // OUT: the private scalar
559     TPM_ECC_CURVE            curveId,       // IN: the curve identifier
560     TPM_ALG_ID               hashAlg,       // IN: hash algorithm to use in the key
561                                             //     generation process
562     TPM2B                   *seed,          // IN: the seed to use
563     const char              *label,         // IN: A label for the generation
564                                             //     process.
565     TPM2B                   *extra,         // IN: Party 1 data for the KDF
566     UINT32                  *counter        // IN/OUT: Counter value to allow KDF
567                                             //     iteration to be propagated across
568                                             //     multiple functions
569     )
570 {
571     const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
572     INT16                    keySizeInBytes;
573     UINT32                   count = 0;
574     CRYPT_RESULT             retVal;
575     UINT16                   hLen = _cpri__GetDigestSize(hashAlg);
576     BIGNUM                  *bnNm1;         // Order of the curve minus one
577     BIGNUM                  *bnD;           // the private scalar
578     BN_CTX                  *context;       // the context for the BIGNUM values
579     BYTE                     withExtra[MAX_ECC_KEY_BYTES + 8];  // trial key with
580                                                                 //extra bits
581     TPM2B_4_BYTE_VALUE       marshaledCounter = {4, {0}};
582     UINT32                   totalBits;
583
584     // Validate parameters (these are fatal)
```

```
585         pAssert(   seed != NULL && dOut != NULL && Qout != NULL && curveData != NULL);
586
587         // Non-fatal parameter checks.
588         if(hLen <= 0)
589             return CRYPT_PARAMETER;
590
591         // allocate the local BN values
592         context = BN_CTX_new();
593         if(context == NULL)
594             FAIL(FATAL_ERROR_ALLOCATION);
595         BN_CTX_start(context);
596         bnNm1 = BN_CTX_get(context);
597         bnD = BN_CTX_get(context);
598
599         // The size of the input scalars is limited by the size of the size of a
600         // TPM2B_ECC_PARAMETER. Make sure that it is not irrational.
601         pAssert((int) curveData->n->size <= MAX_ECC_KEY_BYTES);
602
603         if(   bnD == NULL
604            || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnNm1) == NULL
605            || (keySizeInBytes = (INT16) BN_num_bytes(bnNm1)) > MAX_ECC_KEY_BYTES)
606             FAIL(FATAL_ERROR_INTERNAL);
607
608         // get the total number of bits
609         totalBits = BN_num_bits(bnNm1) + 64;
610
611         // Reduce bnNm1 from 'n' to 'n' - 1
612         BN_sub_word(bnNm1, 1);
613
614         // Initialize the count value
615         if(counter != NULL)
616             count = *counter;
617         if(count == 0)
618             count = 1;
619
620         // Start search for key (should be quick)
621         for(; count != 0; count++)
622         {
623
624             UINT32_TO_BYTE_ARRAY(count, marshaledCounter.t.buffer);
625             _cpri__KDFa(hashAlg, seed, label, extra, &marshaledCounter.b,
626                         totalBits, withExtra, NULL, FALSE);
627
628             // Convert the result and modular reduce
629             // Assume the size variables do not overflow, which should not happen in
630             // the contexts that this function will be called.
631             pAssert(keySizeInBytes <= MAX_ECC_KEY_BYTES);
632             if (   BN_bin2bn(withExtra, keySizeInBytes+8, bnD) == NULL
633                 || BN_mod(bnD, bnD, bnNm1, context) != 1)
634                 FAIL(FATAL_ERROR_INTERNAL);
635
636             // Add one to get 0 < d < n
637             BN_add_word(bnD, 1);
638             if(BnTo2B(&dOut->b, bnD, keySizeInBytes) != 1)
639                     FAIL(FATAL_ERROR_INTERNAL);
640
641             // Do the point multiply to create the public portion of the key. If
642             // the multiply generates the point at infinity (unlikely), do another
643             // iteration.
644             if(    (retVal = _cpri__EccPointMultiply(Qout, curveId, dOut, NULL, NULL))
645                 != CRYPT_NO_RESULT)
646                 break;
647         }
648
649         if(count == 0) // if counter wrapped, then the TPM should go into failure mode
650             FAIL(FATAL_ERROR_INTERNAL);
```

```
651
652         // Free up allocated BN values
653         BN_CTX_end(context);
654         BN_CTX_free(context);
655         if(counter != NULL)
656             *counter = count;
657         return retVal;
658     }
```

B.13.3.2.16. _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

```
659     LIB_EXPORT CRYPT_RESULT
660     _cpri__GetEphemeralEcc(
661         TPMS_ECC_POINT          *Qout,          // OUT: the public point
662         TPM2B_ECC_PARAMETER     *dOut,          // OUT: the private scalar
663         TPM_ECC_CURVE            curveId        // IN: the curve for the key
664         )
665     {
666         CRYPT_RESULT             retVal;
667         const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
668
669         pAssert(curveData != NULL);
670
671         // Keep getting random values until one is found that doesn't create a point
672         // at infinity. This will never, ever, ever, ever, ever, happen but if it does
673         // we have to get a next random value.
674         while(TRUE)
675         {
676             GetRandomPrivate(dOut, curveData->p);
677
678             // _cpri__EccPointMultiply does not return CRYPT_ECC_POINT if no point is
679             // provided. CRYPT_PARAMTER should not be returned because the curve ID
680             // has to be supported. Thus the only possible error is CRYPT_NO_RESULT.
681             retVal =  _cpri__EccPointMultiply(Qout, curveId,  dOut, NULL, NULL);
682             if(retVal != CRYPT_NO_RESULT)
683                 return retVal; // Will return CRYPT_SUCCESS
684         }
685     }
686     #ifdef TPM_ALG_ECDSA  //%
```

B.13.3.2.17. 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.

```
687     LIB_EXPORT CRYPT_RESULT
688     SignEcdsa(
689         TPM2B_ECC_PARAMETER     *rOut,          // OUT: r component of the signature
690         TPM2B_ECC_PARAMETER     *sOut,          // OUT: s component of the signature
691         TPM_ECC_CURVE            curveId,       // IN: the curve used in the signature
692                                                 //     process
693         TPM2B_ECC_PARAMETER     *dIn,           // IN: the private key
694         TPM2B                   *digest         // IN: the value to sign
695         )
696     {
697         BIGNUM                  *bnK;
698         BIGNUM                  *bnIk;
699         BIGNUM                  *bnN;
700         BIGNUM                  *bnR;
```

```
701         BIGNUM                  *bnD;
702         BIGNUM                  *bnZ;
703         TPM2B_ECC_PARAMETER      k;
704         TPMS_ECC_POINT           R;
705         BN_CTX                  *context;
706         CRYPT_RESULT             retVal = CRYPT_SUCCESS;
707         const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
708
709         pAssert(rOut != NULL && sOut != NULL && dIn != NULL && digest != NULL);
710
711         context = BN_CTX_new();
712         if(context == NULL)
713             FAIL(FATAL_ERROR_ALLOCATION);
714         BN_CTX_start(context);
715         bnN = BN_CTX_get(context);
716         bnZ = BN_CTX_get(context);
717         bnR = BN_CTX_get(context);
718         bnD = BN_CTX_get(context);
719         bnIk = BN_CTX_get(context);
720         bnK = BN_CTX_get(context);
721         // Assume the size variables do not overflow, which should not happen in
722         // the contexts that this function will be called.
723         pAssert(curveData->n->size <= MAX_ECC_PARAMETER_BYTES);
724         if(   bnK == NULL
725            || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
726             FAIL(FATAL_ERROR_INTERNAL);
727
728     // The algorithm as described in "Suite B Implementer's Guide to FIPS 186-3(ECDSA)"
729     // 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a per-message
730     //    secret number and its inverse modulo n. Since n is prime, the
731     //    output will be invalid only if there is a failure in the RBG.
732     // 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
733     //    multiplication (see [Routines]), where G is the base point included in
734     //    the set of domain parameters.
735     // 3. Compute r = xR mod n. If r = 0, then return to Step 1. 1.
736     // 4. Use the selected hash function to compute H = Hash(M).
737     // 5. Convert the bit string H to an integer e as described in Appendix B.2.
738     // 6. Compute s = (k^-1 *  (e + d *  r)) mod n. If s = 0, return to Step 1.2.
739     // 7. Return (r, s).
740
741         // Generate a random value k in the range 1 <= k < n
742         // Want a K value that is the same size as the curve order
743         k.t.size = curveData->n->size;
744
745         while(TRUE) // This implements the loop at step 6. If s is zero, start over.
746         {
747             while(TRUE)
748             {
749                 // Step 1 and 2 -- generate an ephemeral key and the modular inverse
750                 // of the private key.
751                 while(TRUE)
752                 {
753                     GetRandomPrivate(&k, curveData->n);
754
755                     // Do the point multiply to generate a point and check to see if
756                     // the point it at infinity
757                     if(    _cpri__EccPointMultiply(&R, curveId, &k, NULL, NULL)
758                         != CRYPT_NO_RESULT)
759                         break;  // can only be CRYPT_SUCCESS
760                 }
761
762                 // x coordinate is mod p.  Make it mod n
763                 // Assume the size variables do not overflow, which should not happen
764                 // in the contexts that this function will be called.
765                 assert2Bsize(R.x.t);
766                 BN_bin2bn(R.x.t.buffer, R.x.t.size, bnR);
```

```
767             BN_mod(bnR, bnR, bnN, context);
768
769             // Make sure that it is not zero;
770             if(BN_is_zero(bnR))
771                 continue;
772
773             // Make sure that a modular inverse exists
774             // Assume the size variables do not overflow, which should not happen
775             // in the contexts that this function will be called.
776             assert2Bsize(k.t);
777             BN_bin2bn(k.t.buffer, k.t.size, bnK);
778             if( BN_mod_inverse(bnIk, bnK, bnN, context) != NULL)
779                 break;
780         }
781
782         // Set z = leftmost bits of the digest
783         // NOTE: This is implemented such that the key size needs to be
784         //       an even number of bytes in length.
785         if(digest->size > curveData->n->size)
786         {
787             // Assume the size variables do not overflow, which should not happen
788             // in the contexts that this function will be called.
789             pAssert(curveData->n->size <= MAX_ECC_KEY_BYTES);
790             // digest is larger than n so truncate
791             BN_bin2bn(digest->buffer, curveData->n->size, bnZ);
792         }
793         else
794         {
795             // Assume the size variables do not overflow, which should not happen
796             // in the contexts that this function will be called.
797             pAssert(digest->size <= MAX_DIGEST_SIZE);
798             // digest is same or smaller than n so use it all
799             BN_bin2bn(digest->buffer, digest->size, bnZ);
800         }
801
802         // Assume the size variables do not overflow, which should not happen in
803         // the contexts that this function will be called.
804         assert2Bsize(dIn->t);
805         if(   bnZ == NULL
806
807            // need the private scalar of the signing key
808            || BN_bin2bn(dIn->t.buffer, dIn->t.size, bnD) == NULL)
809             FAIL(FATAL_ERROR_INTERNAL);
810
811         // NOTE: When the result of an operation is going to be reduced mod x
812         // any modular multiplication is done so that the intermediate values
813         // don't get too large.
814         //
815         // now have inverse of K (bnIk), z (bnZ), r (bnR),  d (bnD) and n (bnN)
816         // Compute s = k^-1 (z + r*d)(mod n)
817             //  first do d = r*d mod n
818         if(  !BN_mod_mul(bnD, bnR, bnD, bnN, context)
819
820            // d = z + r * d
821            || !BN_add(bnD, bnZ, bnD)
822
823            // d = k^(-1)(z + r * d)(mod n)
824            || !BN_mod_mul(bnD, bnIk, bnD, bnN, context)
825
826            // convert to TPM2B format
827            || !BnTo2B(&sOut->b, bnD, curveData->n->size)
828
829            // and write the modular reduced version of r
830            // NOTE: this was deferred to reduce the number of
831            // error checks.
832            || !BnTo2B(&rOut->b, bnR, curveData->n->size))
```

```
833                 FAIL(FATAL_ERROR_INTERNAL);
834
835             if(!BN_is_zero(bnD))
836                 break;  // signature not zero so done
837
838             // if the signature value was zero, start over
839         }
840
841         // Free up allocated BN values
842         BN_CTX_end(context);
843         BN_CTX_free(context);
844         return retVal;
845     }
846     #endif  //%
847     #if defined TPM_ALG_ECDAA || defined TPM_ALG_ECSCHNORR  //%
```

B.13.3.2.18. EcDaa()

This function is used to perform a modified Schnorr signature for ECDAA.

This function performs *s* = *k* + *T* * *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 *tIn* is NULL then use *tOut* as *T*

Return Value	Meaning
CRYPT_SUCCESS	signature created

```
848     static CRYPT_RESULT
849     EcDaa(
850         TPM2B_ECC_PARAMETER     *tOut,          // OUT: T component of the signature
851         TPM2B_ECC_PARAMETER     *sOut,          // OUT: s component of the signature
852         TPM_ECC_CURVE            curveId,       // IN: the curve used in signing
853         TPM2B_ECC_PARAMETER     *dIn,           // IN: the private key
854         TPM2B                   *tIn,           // IN: the value to sign
855         TPM2B_ECC_PARAMETER     *kIn            // IN: a random value from commit
856         )
857     {
858         BIGNUM                  *bnN, *bnK, *bnT, *bnD;
859         BN_CTX                  *context;
860         const TPM2B             *n;
861         const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
862         BOOL                     OK = TRUE;
863
864         // Parameter checks
865          pAssert(   sOut != NULL && dIn != NULL && tOut != NULL
866                  && kIn != NULL && curveData != NULL);
867
868         // this just saves key strokes
869         n = curveData->n;
870
871         if(tIn != NULL)
872             Copy2B(&tOut->b, tIn);
873
874         // The size of dIn and kIn input scalars is limited by the size of the size
875         // of a TPM2B_ECC_PARAMETER and tIn can be no larger than a digest.
876         // Make sure they are within range.
877         pAssert(   (int) dIn->t.size <= MAX_ECC_KEY_BYTES
878                 && (int) kIn->t.size <= MAX_ECC_KEY_BYTES
```

```
879             && (int) tOut->t.size <= MAX_DIGEST_SIZE
880            );
881
882     context = BN_CTX_new();
883     if(context == NULL)
884         FAIL(FATAL_ERROR_ALLOCATION);
885     BN_CTX_start(context);
886     bnN = BN_CTX_get(context);
887     bnK = BN_CTX_get(context);
888     bnT = BN_CTX_get(context);
889     bnD = BN_CTX_get(context);
890
891     // Check for allocation problems
892     if(bnD == NULL)
893         FAIL(FATAL_ERROR_ALLOCATION);
894
895     // Convert values
896     if(   BN_bin2bn(n->buffer, n->size, bnN) == NULL
897        || BN_bin2bn(kIn->t.buffer, kIn->t.size, bnK) ==  NULL
898        || BN_bin2bn(dIn->t.buffer, dIn->t.size, bnD) == NULL
899        || BN_bin2bn(tOut->t.buffer, tOut->t.size, bnT) == NULL)
900
901         FAIL(FATAL_ERROR_INTERNAL);
902     // Compute T = T mod n
903     OK = OK && BN_mod(bnT, bnT, bnN, context);
904
905     // compute (s = k + T * d mod n)
906             //   d = T * d mod n
907     OK = OK && BN_mod_mul(bnD, bnT, bnD, bnN, context) == 1;
908             //   d = k + T * d mod n
909     OK = OK && BN_mod_add(bnD, bnK, bnD, bnN, context) == 1;
910             //   s = d
911     OK = OK && BnTo2B(&sOut->b, bnD, n->size);
912             //   r = T
913     OK = OK && BnTo2B(&tOut->b, bnT, n->size);
914     if(!OK)
915         FAIL(FATAL_ERROR_INTERNAL);
916
917     // Cleanup
918     BN_CTX_end(context);
919     BN_CTX_free(context);
920
921     return CRYPT_SUCCESS;
922 }
923 #endif  //%
924 #ifdef TPM_ALG_ECSCHNORR //%
```

B.13.3.2.19. SchnorrEcc()

This function is used to perform a modified Schnorr signature.

This function will generate a random value k and compute

a) *(xR, yR) = [k]G*

b) *r* = hash*(P || xR)*(mod n)

c) *s= k + r * ds*

d) return the tuple T, s

Return Value	Meaning
CRYPT_SUCCESS	signature created
CRYPT_SCHEME	*hashAlg* can't produce zero-length digest

```
925  static CRYPT_RESULT
926  SchnorrEcc(
927      TPM2B_ECC_PARAMETER     *rOut,          // OUT: r component of the signature
928      TPM2B_ECC_PARAMETER     *sOut,          // OUT: s component of the signature
929      TPM_ALG_ID               hashAlg,       // IN: hash algorithm used
930      TPM_ECC_CURVE            curveId,       // IN: the curve used in signing
931      TPM2B_ECC_PARAMETER     *dIn,           // IN: the private key
932      TPM2B                   *digest,        // IN: the digest to sign
933      TPM2B_ECC_PARAMETER     *kIn            // IN: for testing
934      )
935  {
936      TPM2B_ECC_PARAMETER      k;
937      BIGNUM                  *bnR, *bnN, *bnK, *bnT, *bnD;
938      BN_CTX                  *context;
939      const TPM2B             *n;
940      EC_POINT                *pR = NULL;
941      EC_GROUP                *group = NULL;
942      CPRI_HASH_STATE          hashState;
943      UINT16                   digestSize = _cpri__GetDigestSize(hashAlg);
944      const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
945      TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_PARAMETER_BYTES));
946      TPM2B_T                  T2b;
947      BOOL                     OK = TRUE;
948
949      // Parameter checks
950
951      // Must have a place for the 'r' and 's' parts of the signature, a private
952      // key ('d')
953      pAssert(   rOut != NULL && sOut != NULL && dIn != NULL
954              && digest != NULL && curveData != NULL);
955
956      // to save key strokes
957      n = curveData->n;
958
959      // If the digest does not produce a hash, then null the signature and return
960      // a failure.
961      if(digestSize == 0)
962      {
963          rOut->t.size = 0;
964          sOut->t.size = 0;
965          return CRYPT_SCHEME;
966      }
967
968      // Allocate big number values
969      context = BN_CTX_new();
970      if(context == NULL)
971          FAIL(FATAL_ERROR_ALLOCATION);
972      BN_CTX_start(context);
973      bnR = BN_CTX_get(context);
974      bnN = BN_CTX_get(context);
975      bnK = BN_CTX_get(context);
976      bnT = BN_CTX_get(context);
977      bnD = BN_CTX_get(context);
978      if(   bnD == NULL
979              // initialize the group parameters
980         || (group = EccCurveInit(curveId, context)) == NULL
981             // allocate a local point
982         || (pR = EC_POINT_new(group)) == NULL
983        )
```

```
984             FAIL(FATAL_ERROR_ALLOCATION);
985
986         if(BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
987             FAIL(FATAL_ERROR_INTERNAL);
988
989         while(OK)
990         {
991     // a) set k to a random value such that 1  k  n-1
992             if(kIn != NULL)
993             {
994                 Copy2B(&k.b, &kIn->b);  // copy input k if testing
995                 OK = FALSE;             // not OK to loop
996             }
997             else
998             // If get a random value in the correct range
999                 GetRandomPrivate(&k, n);
1000
1001            // Convert 'k' and generate pR = ['k']G
1002            BnFrom2B(bnK, &k.b);
1003
1004    // b) compute E  (xE, yE)  [k]G
1005            if(PointMul(group, pR, bnK, NULL, NULL, context) == CRYPT_NO_RESULT)
1006    // c) if E is the point at infinity, go to a)
1007                continue;
1008
1009    // d) compute e  xE (mod n)
1010            // Get the x coordinate of the point
1011            EC_POINT_get_affine_coordinates_GFp(group, pR, bnR, NULL, context);
1012
1013            // make (mod n)
1014            BN_mod(bnR, bnR, bnN, context);
1015
1016    // e) if e is zero, go to a)
1017            if(BN_is_zero(bnR))
1018                continue;
1019
1020            // Convert xR to a string (use T as a temp)
1021            BnTo2B(&T2b.b, bnR, (UINT16)(BN_num_bits(bnR)+7)/8);
1022
1023    // f) compute r  HschemeHash(P || e) (mod n)
1024            _cpri__StartHash(hashAlg, FALSE, &hashState);
1025            _cpri__UpdateHash(&hashState, digest->size, digest->buffer);
1026            _cpri__UpdateHash(&hashState, T2b.t.size, T2b.t.buffer);
1027            if(_cpri__CompleteHash(&hashState, digestSize, T2b.b.buffer) != digestSize)
1028                FAIL(FATAL_ERROR_INTERNAL);
1029            T2b.t.size = digestSize;
1030            BnFrom2B(bnT, &T2b.b);
1031            BN_div(NULL, bnT, bnT, bnN, context);
1032            BnTo2B(&rOut->b, bnT, (UINT16)BN_num_bytes(bnT));
1033
1034            // We have a value and we are going to exit the loop successfully
1035            OK = TRUE;
1036            break;
1037        }
1038        // Cleanup
1039        EC_POINT_free(pR);
1040        EC_GROUP_free(group);
1041        BN_CTX_end(context);
1042        BN_CTX_free(context);
1043
1044        // If we have a value, finish the signature
1045        if(OK)
1046            return EcDaa(rOut, sOut, curveId, dIn, NULL, &k);
1047        else
1048            return CRYPT_NO_RESULT;
1049    }
```



```
1050    #endif  //%
1051    #ifdef TPM_ALG_SM2 //%
1052    #ifdef  _SM2_SIGN_DEBUG //%
1053    static int
1054    cmp_bn2hex(
1055        BIGNUM          *bn,            // IN: big number value
1056        const char      *c              // IN: character string number
1057        )
1058    {
1059        int         result;
1060        BIGNUM      *bnC = BN_new();
1061        pAssert(bnC != NULL);
1062
1063        BN_hex2bn(&bnC, c);
1064        result = BN_ucmp(bn, bnC);
1065        BN_free(bnC);
1066        return result;
1067    }
1068    static int
1069    cmp_2B2hex(
1070        TPM2B           *a,             // IN: TPM2B number to compare
1071        const char      *c              // IN: character string
1072        )
1073    {
1074        int         result;
1075        int         sl = strlen(c);
1076        BIGNUM      *bnA;
1077
1078        result = (a->size * 2) - sl;
1079        if(result != 0)
1080            return result;
1081        pAssert((bnA = BN_bin2bn(a->buffer, a->size, NULL)) != NULL);
1082        result = cmp_bn2hex(bnA, c);
1083        BN_free(bnA);
1084        return result;
1085    }
1086    static void
1087    cpy_hexTo2B(
1088        TPM2B           *b,             // OUT: receives value
1089        const char      *c              // IN: source string
1090        )
1091    {
1092        BIGNUM      *bnB = BN_new();
1093        pAssert((strlen(c) & 1) == 0);  // must have an even number of digits
1094        b->size = strlen(c) / 2;
1095        BN_hex2bn(&bnB, c);
1096        pAssert(bnB != NULL);
1097        BnTo2B(b, bnB, b->size);
1098        BN_free(bnB);
1099
1100    }
1101    #endif //% _SM2_SIGN_DEBUG
```

B.13.3.2.20. SignSM2()

This function signs a digest using the method defined in SM2 Part 2. The method in the standard will add a header to the message to be signed that is a hash of the values that define the key. This then hashed with the message to produce a digest (e) that is signed. This function signs e.

Return Value	Meaning
CRYPT_SUCCESS	sign worked

```
1102    static CRYPT_RESULT
1103    SignSM2(
1104        TPM2B_ECC_PARAMETER     *rOut,          // OUT: r component of the signature
1105        TPM2B_ECC_PARAMETER     *sOut,          // OUT: s component of the signature
1106        TPM_ECC_CURVE            curveId,       // IN: the curve used in signing
1107        TPM2B_ECC_PARAMETER     *dIn,           // IN: the private key
1108        TPM2B                   *digest         // IN: the digest to sign
1109        )
1110    {
1111        BIGNUM                  *bnR;
1112        BIGNUM                  *bnS;
1113        BIGNUM                  *bnN;
1114        BIGNUM                  *bnK;
1115        BIGNUM                  *bnX1;
1116        BIGNUM                  *bnD;
1117        BIGNUM                  *bnT;   // temp
1118        BIGNUM                  *bnE;
1119
1120        BN_CTX                  *context;
1121        TPM2B_TYPE(DIGEST, MAX_DIGEST_SIZE);
1122        TPM2B_ECC_PARAMETER      k;
1123        TPMS_ECC_POINT           p2Br;
1124        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1125
1126        pAssert(curveData != NULL);
1127        context = BN_CTX_new();
1128        BN_CTX_start(context);
1129        bnK = BN_CTX_get(context);
1130        bnR = BN_CTX_get(context);
1131        bnS = BN_CTX_get(context);
1132        bnX1 = BN_CTX_get(context);
1133        bnN = BN_CTX_get(context);
1134        bnD = BN_CTX_get(context);
1135        bnT = BN_CTX_get(context);
1136        bnE = BN_CTX_get(context);
1137        if(bnE == NULL)
1138            FAIL(FATAL_ERROR_ALLOCATION);
1139
1140        BnFrom2B(bnE, digest);
1141        BnFrom2B(bnN, curveData->n);
1142        BnFrom2B(bnD, &dIn->b);
1143
1144    #ifdef _SM2_SIGN_DEBUG
1145    BN_hex2bn(&bnE, "B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76");
1146    BN_hex2bn(&bnD, "128B2FA8BD433C6C068C8D803DFF79792A519A55171B1B650C23661D15897263");
1147    #endif
1148    // A3: Use random number generator to generate random number 1 <= k <= n-1;
1149    // NOTE: Ax: numbers are from the SM2 standard
1150        k.t.size = curveData->n->size;
1151    loop:
1152        {
1153            // Get a random number
1154            _cpri__GenerateRandom(k.t.size, k.t.buffer);
1155
1156    #ifdef _SM2_SIGN_DEBUG
1157    BN_hex2bn(&bnK, "6CB28D99385C175C94F94E934817663FC176D925DD72B727260DBAAE1FB2F96F");
1158    BnTo2B(&k.b,bnK, 32);
1159    k.t.size = 32;
1160    #endif
1161            //make sure that the number is 0 < k < n
1162            BnFrom2B(bnK, &k.b);
```

```
1163            if(    BN_ucmp(bnK, bnN) >= 0
1164                || BN_is_zero(bnK))
1165                goto loop;
1166
1167    // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
1168    // to details specified in 4.2.7 in Part 1 of this document, transform the
1169    // data type of x1 into an integer;
1170            if(   _cpri__EccPointMultiply(&p2Br, curveId, &k, NULL, NULL)
1171               == CRYPT_NO_RESULT)
1172                goto loop;
1173
1174            BnFrom2B(bnX1, &p2Br.x.b);
1175
1176     // A5: Figure out r = (e + x1) mod n,
1177            if(!BN_mod_add(bnR, bnE, bnX1, bnN, context))
1178                FAIL(FATAL_ERROR_INTERNAL);
1179    #ifdef _SM2_SIGN_DEBUG
1180    pAssert(cmp_bn2hex(bnR,
1181                       "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1")
1182            == 0);
1183    #endif
1184
1185               // if r=0 or r+k=n, return to A3;
1186             if(!BN_add(bnT, bnK, bnR))
1187                FAIL(FATAL_ERROR_INTERNAL);
1188
1189            if(BN_is_zero(bnR) || BN_ucmp(bnT, bnN) == 0)
1190                goto loop;
1191
1192    // A6: Figure out s = ((1 + dA)^-1  (k - r  dA)) mod n, if s=0, return to A3;
1193            // compute t = (1+d)-1
1194            BN_copy(bnT, bnD);
1195            if(   !BN_add_word(bnT, 1)
1196               || !BN_mod_inverse(bnT, bnT, bnN, context) // (1 + dA)^-1 mod n
1197               )
1198                FAIL(FATAL_ERROR_INTERNAL);
1199    #ifdef _SM2_SIGN_DEBUG
1200    pAssert(cmp_bn2hex(bnT,
1201                       "79BFCF3052C80DA7B939E0C6914A18CBB2D96D8555256E83122743A7D4F5F956")
1202            == 0);
1203    #endif
1204            // compute s = t * (k - r * dA) mod n
1205            if(   !BN_mod_mul(bnS, bnD, bnR, bnN, context) // (r * dA) mod n
1206               || !BN_mod_sub(bnS, bnK, bnS, bnN, context) // (k - (r * dA) mod n
1207               || !BN_mod_mul(bnS, bnT, bnS, bnN, context))// t * (k - (r * dA) mod n
1208               FAIL(FATAL_ERROR_INTERNAL);
1209    #ifdef _SM2_SIGN_DEBUG
1210    pAssert(cmp_bn2hex(bnS,
1211                       "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7")
1212            == 0);
1213    #endif
1214
1215            if(BN_is_zero(bnS))
1216                goto loop;
1217        }
1218
1219    // A7: According to details specified in 4.2.1 in Part 1 of this document, transform
1220    // the data type of r, s into bit strings, signature of message M is (r, s).
1221
1222        BnTo2B(&rOut->b, bnR, curveData->n->size);
1223        BnTo2B(&sOut->b, bnS, curveData->n->size);
1224    #ifdef _SM2_SIGN_DEBUG
1225    pAssert(cmp_2B2hex(&rOut->b,
1226                       "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1")
1227            == 0);
1228    pAssert(cmp_2B2hex(&sOut->b,
```

```
1229                 "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7")
1230         == 0);
1231 #endif
1232     BN_CTX_end(context);
1233     BN_CTX_free(context);
1234     return CRYPT_SUCCESS;
1235 }
1236 #endif  //% TPM_ALG_SM2
```

B.13.3.2.21. _cpri__SignEcc()

This function is the dispatch function for the various ECC-based signing schemes.

Return Value	Meaning
CRYPT_SCHEME	*scheme* is not supported

```
1237 LIB_EXPORT CRYPT_RESULT
1238 _cpri__SignEcc(
1239     TPM2B_ECC_PARAMETER     *rOut,          // OUT: r component of the signature
1240     TPM2B_ECC_PARAMETER     *sOut,          // OUT: s component of the signature
1241     TPM_ALG_ID               scheme,        // IN: the scheme selector
1242     TPM_ALG_ID               hashAlg,       // IN: the hash algorithm if need
1243     TPM_ECC_CURVE            curveId,       // IN: the curve used in the signature
1244                                             //     process
1245     TPM2B_ECC_PARAMETER     *dIn,           // IN: the private key
1246     TPM2B                   *digest,        // IN: the digest to sign
1247     TPM2B_ECC_PARAMETER     *kIn            // IN: k for input
1248     )
1249 {
1250     switch  (scheme)
1251     {
1252         case TPM_ALG_ECDSA:
1253             // SignEcdsa always works
1254             return SignEcdsa(rOut, sOut, curveId, dIn, digest);
1255             break;
1256 #ifdef TPM_ALG_ECDAA
1257         case TPM_ALG_ECDAA:
1258             if(rOut != NULL)
1259                 rOut->b.size = 0;
1260             return EcDaa(rOut, sOut, curveId, dIn, digest, kIn);
1261             break;
1262 #endif
1263 #ifdef TPM_ALG_ECSCHNORR
1264         case TPM_ALG_ECSCHNORR:
1265             return SchnorrEcc(rOut, sOut, hashAlg, curveId, dIn, digest, kIn);
1266             break;
1267 #endif
1268 #ifdef TPM_ALG_SM2
1269         case TPM_ALG_SM2:
1270             return SignSM2(rOut, sOut, curveId, dIn, digest);
1271             break;
1272 #endif
1273         default:
1274             return CRYPT_SCHEME;
1275     }
1276 }
1277 #ifdef TPM_ALG_ECDSA //%
```

B.13.3.2.22. ValidateSignatureEcdsa()

This function validates an ECDSA signature. *rIn* and *sIn* shoudl have been checked to make sure that they are not zero.

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid

```
1278    static CRYPT_RESULT
1279    ValidateSignatureEcdsa(
1280        TPM2B_ECC_PARAMETER     *rIn,           // IN: r component of the signature
1281        TPM2B_ECC_PARAMETER     *sIn,           // IN: s component of the signature
1282        TPM_ECC_CURVE            curveId,       // IN: the curve used in the signature
1283                                                //     process
1284        TPMS_ECC_POINT          *Qin,           // IN: the public point of the key
1285        TPM2B                   *digest         // IN: the digest that was signed
1286        )
1287    {
1288        TPM2B_ECC_PARAMETER      U1;
1289        TPM2B_ECC_PARAMETER      U2;
1290        TPMS_ECC_POINT           R;
1291        const TPM2B             *n;
1292        BN_CTX                  *context;
1293        EC_POINT                *pQ = NULL;
1294        EC_GROUP                *group = NULL;
1295        BIGNUM                  *bnU1;
1296        BIGNUM                  *bnU2;
1297        BIGNUM                  *bnR;
1298        BIGNUM                  *bnS;
1299        BIGNUM                  *bnW;
1300        BIGNUM                  *bnV;
1301        BIGNUM                  *bnN;
1302        BIGNUM                  *bnE;
1303        BIGNUM                  *bnGx;
1304        BIGNUM                  *bnGy;
1305        BIGNUM                  *bnQx;
1306        BIGNUM                  *bnQy;
1307        CRYPT_RESULT             retVal = CRYPT_FAIL;
1308        int                      t;
1309
1310        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1311
1312        // The curve selector should have been filtered by the unmarshaling process
1313        pAssert (curveData != NULL);
1314        n = curveData->n;
1315
1316    // 1. If r and s are not both integers in the interval [1, n - 1], output
1317    //    INVALID.
1318    //  rIn and sIn are known to be greater than zero (was checked by the caller).
1319        if(     _math__uComp(rIn->t.size, rIn->t.buffer, n->size, n->buffer) >= 0
1320            ||  _math__uComp(sIn->t.size, sIn->t.buffer, n->size, n->buffer) >= 0
1321          )
1322           return CRYPT_FAIL;
1323
1324        context = BN_CTX_new();
1325        if(context == NULL)
1326            FAIL(FATAL_ERROR_ALLOCATION);
1327        BN_CTX_start(context);
1328        bnR = BN_CTX_get(context);
1329        bnS = BN_CTX_get(context);
1330        bnN = BN_CTX_get(context);
1331        bnE = BN_CTX_get(context);
1332        bnV = BN_CTX_get(context);
1333        bnW = BN_CTX_get(context);
1334        bnGx = BN_CTX_get(context);
1335        bnGy = BN_CTX_get(context);
1336        bnQx = BN_CTX_get(context);
```

```
1337        bnQy = BN_CTX_get(context);
1338        bnU1 = BN_CTX_get(context);
1339        bnU2 = BN_CTX_get(context);
1340
1341        // Assume the size variables do not overflow, which should not happen in
1342        // the contexts that this function will be called.
1343        assert2Bsize(Qin->x.t);
1344        assert2Bsize(rIn->t);
1345        assert2Bsize(sIn->t);
1346
1347        // BN_CTX_get() is sticky so only need to check the last value to know that
1348        // all worked.
1349        if(   bnU2 == NULL
1350
1351            // initialize the group parameters
1352           || (group = EccCurveInit(curveId, context)) == NULL
1353
1354           // allocate a local point
1355           || (pQ = EC_POINT_new(group)) == NULL
1356
1357           // use the public key values (QxIn and QyIn) to initialize Q
1358           || BN_bin2bn(Qin->x.t.buffer, Qin->x.t.size, bnQx) == NULL
1359           || BN_bin2bn(Qin->x.t.buffer, Qin->x.t.size, bnQy) == NULL
1360           || !EC_POINT_set_affine_coordinates_GFp(group, pQ, bnQx, bnQy, context)
1361
1362           // convert the signature values
1363           || BN_bin2bn(rIn->t.buffer, rIn->t.size, bnR) == NULL
1364           || BN_bin2bn(sIn->t.buffer, sIn->t.size, bnS) == NULL
1365
1366           // convert the curve order
1367           || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
1368            FAIL(FATAL_ERROR_INTERNAL);
1369
1370    // 2. Use the selected hash function to compute H0 = Hash(M0).
1371        // This is an input parameter
1372
1373    // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
1374        t = (digest->size > rIn->t.size) ? rIn->t.size : digest->size;
1375        if(BN_bin2bn(digest->buffer, t, bnE) == NULL)
1376            FAIL(FATAL_ERROR_INTERNAL);
1377
1378    // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
1379        if (BN_mod_inverse(bnW, bnS, bnN, context) == NULL)
1380            FAIL(FATAL_ERROR_INTERNAL);
1381
1382    // 5. Compute u1 = (e' *   w) mod n, and compute u2 = (r' *  w) mod n.
1383        if(   !BN_mod_mul(bnU1, bnE, bnW, bnN, context)
1384           || !BN_mod_mul(bnU2, bnR, bnW, bnN, context))
1385            FAIL(FATAL_ERROR_INTERNAL);
1386
1387        BnTo2B(&U1.b, bnU1, (INT16) BN_num_bytes(bnU1));
1388        BnTo2B(&U2.b, bnU2, (INT16) BN_num_bytes(bnU2));
1389
1390    // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
1391    //    scalar multiplication and EC addition (see [Routines]). If R is equal to
1392    //    the point at infinity O, output INVALID.
1393        if(_cpri__EccPointMultiply(&R, curveId, &U1, Qin, &U2) == CRYPT_SUCCESS)
1394        {
1395            // 7. Compute v = Rx mod n.
1396            if(   BN_bin2bn(R.x.t.buffer, R.x.t.size, bnV) == NULL
1397               || !BN_mod(bnV, bnV, bnN, context))
1398                FAIL(FATAL_ERROR_INTERNAL);
1399
1400        // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
1401            if(BN_cmp(bnV, bnR) == 0)
1402                retVal = CRYPT_SUCCESS;
```

```
1403        }
1404
1405        if(pQ != NULL) EC_POINT_free(pQ);
1406        if(group != NULL) EC_GROUP_free(group);
1407        BN_CTX_end(context);
1408        BN_CTX_free(context);
1409
1410        return retVal;
1411    }
1412    #endif      //% TPM_ALG_ECDSA
1413    #ifdef TPM_ALG_ECSCHNORR //%
```

B.13.3.2.23. ValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature. *rIn* and *sIn* are required to be greater than zero. This is checked in _cpri__ValidateSignatureEcc().

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid
CRYPT_SCHEME	*hashAlg* is not supported

```
1414    static CRYPT_RESULT
1415    ValidateSignatureEcSchnorr(
1416        TPM2B_ECC_PARAMETER     *rIn,           // IN: r component of the signature
1417        TPM2B_ECC_PARAMETER     *sIn,           // IN: s component of the signature
1418        TPM_ALG_ID               hashAlg,       // IN: hash algorithm of the signature
1419        TPM_ECC_CURVE            curveId,       // IN: the curve used in the signature
1420                                                //     process
1421        TPMS_ECC_POINT          *Qin,           // IN: the public point of the key
1422        TPM2B                   *digest         // IN: the digest that was signed
1423        )
1424    {
1425        TPMS_ECC_POINT           pE;
1426        const TPM2B             *n;
1427        CPRI_HASH_STATE          hashState;
1428        TPM2B_DIGEST             rPrime;
1429        TPM2B_ECC_PARAMETER      minusR;
1430        UINT16                   digestSize = _cpri__GetDigestSize(hashAlg);
1431        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1432
1433        // The curve parameter should have been filtered by unmarshaling code
1434        pAssert(curveData != NULL);
1435
1436        if(digestSize == 0)
1437            return CRYPT_SCHEME;
1438
1439        // Input parameter validation
1440        pAssert(rIn != NULL && sIn != NULL && Qin != NULL && digest != NULL);
1441
1442        n = curveData->n;
1443
1444        // if sIn or rIn are not between 1 and N-1, signature check fails
1445        // sIn and rIn were verified to be non-zero by the caller
1446        if(   _math__uComp(sIn->b.size, sIn->b.buffer, n->size, n->buffer) >= 0
1447           || _math__uComp(rIn->b.size, rIn->b.buffer, n->size, n->buffer) >= 0
1448          )
1449            return CRYPT_FAIL;
1450
1451        //E = [s]InG - [r]InQ
1452        _math__sub(n->size, n->buffer,
1453                   rIn->t.size, rIn->t.buffer,
```

```
1454                    &minusR.t.size, minusR.t.buffer);
1455        if(_cpri__EccPointMultiply(&pE, curveId, sIn, Qin, &minusR) != CRYPT_SUCCESS)
1456            return CRYPT_FAIL;
1457
1458        // Ex = Ex mod N
1459        if(Mod2B(&pE.x.b, n) != CRYPT_SUCCESS)
1460            FAIL(FATAL_ERROR_INTERNAL);
1461
1462        _math__Normalize2B(&pE.x.b);
1463
1464        // rPrime = h(digest || pE.x) mod n;
1465        _cpri__StartHash(hashAlg, FALSE, &hashState);
1466        _cpri__UpdateHash(&hashState, digest->size, digest->buffer);
1467        _cpri__UpdateHash(&hashState, pE.x.t.size, pE.x.t.buffer);
1468        if(_cpri__CompleteHash(&hashState, digestSize, rPrime.t.buffer) != digestSize)
1469            FAIL(FATAL_ERROR_INTERNAL);
1470
1471        rPrime.t.size = digestSize;
1472
1473        // rPrime = rPrime (mod n)
1474        if(Mod2B(&rPrime.b, n) != CRYPT_SUCCESS)
1475            FAIL(FATAL_ERROR_INTERNAL);
1476
1477        // if the values don't match, then the signature is bad
1478        if(_math__uComp(rIn->t.size, rIn->t.buffer,
1479                        rPrime.t.size, rPrime.t.buffer) !=  0)
1480            return CRYPT_FAIL;
1481        else
1482            return CRYPT_SUCCESS;
1483    }
1484    #endif  //% TPM_ALG_ECSCHNORR
1485    #ifdef TPM_ALG_SM2  //%
```

B.13.3.2.24. ValidateSignatueSM2Dsa()

This function is used to validate an SM2 signature.

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid

```
1486    static CRYPT_RESULT
1487    ValidateSignatureSM2Dsa(
1488        TPM2B_ECC_PARAMETER     *rIn,           // IN: r component of the signature
1489        TPM2B_ECC_PARAMETER     *sIn,           // IN: s component of the signature
1490        TPM_ECC_CURVE            curveId,       // IN: the curve used in the signature
1491                                                //     process
1492        TPMS_ECC_POINT          *Qin,           // IN: the public point of the key
1493        TPM2B                   *digest         // IN: the digest that was signed
1494        )
1495    {
1496        BIGNUM                  *bnR;
1497        BIGNUM                  *bnRp;
1498        BIGNUM                  *bnT;
1499        BIGNUM                  *bnS;
1500        BIGNUM                  *bnE;
1501        EC_POINT                *pQ;
1502        BN_CTX                  *context;
1503        EC_GROUP                *group = NULL;
1504        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1505        BOOL                     fail = FALSE;
1506
```

```
1507        if((context = BN_CTX_new()) == NULL || curveData == NULL)
1508            FAIL(FATAL_ERROR_INTERNAL);
1509        bnR = BN_CTX_get(context);
1510        bnRp= BN_CTX_get(context);
1511        bnE = BN_CTX_get(context);
1512        bnT = BN_CTX_get(context);
1513        bnS = BN_CTX_get(context);
1514        if(   bnS == NULL
1515           || (group = EccCurveInit(curveId, context)) == NULL)
1516            FAIL(FATAL_ERROR_INTERNAL);
1517
1518    #ifdef  _SM2_SIGN_DEBUG
1519        cpy_hexTo2B(&Qin->x.b,
1520                "0AE4C7798AA0F119471BEE11825BE46202BB79E2A5844495E97C04FF4DF2548A");
1521        cpy_hexTo2B(&Qin->y.b,
1522                "7C0240F88F1CD4E16352A73C17B7F16F07353E53A176D684A9FE0C6BB798E857");
1523        cpy_hexTo2B(digest,
1524                "B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76");
1525    #endif
1526        pQ = EccInitPoint2B(group, Qin, context);
1527
1528    #ifdef _SM2_SIGN_DEBUG
1529        pAssert(EC_POINT_get_affine_coordinates_GFp(group, pQ, bnT, bnS, context));
1530        pAssert(cmp_bn2hex(bnT,
1531                    "0AE4C7798AA0F119471BEE11825BE46202BB79E2A5844495E97C04FF4DF2548A")
1532                == 0);
1533        pAssert(cmp_bn2hex(bnS,
1534                    "7C0240F88F1CD4E16352A73C17B7F16F07353E53A176D684A9FE0C6BB798E857")
1535                == 0);
1536    #endif
1537
1538        BnFrom2B(bnR, &rIn->b);
1539        BnFrom2B(bnS, &sIn->b);
1540        BnFrom2B(bnE, digest);
1541
1542    #ifdef _SM2_SIGN_DEBUG
1543    // Make sure that the input signature is the test signature
1544    pAssert(cmp_2B2hex(&rIn->b,
1545            "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1") == 0);
1546    pAssert(cmp_2B2hex(&sIn->b,
1547            "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7") == 0);
1548    #endif
1549
1550    // a) verify that r and s are in the inclusive interval 1 to (n  1)
1551        fail = (BN_ucmp(bnR, &group->order) >= 0);
1552
1553        fail = (BN_ucmp(bnS, &group->order) >= 0) || fail;
1554        if(fail)
1555        // There is no reason to continue. Since r and s are inputs from the caller,
1556        // they can know that the values are not in the proper range. So, exiting here
1557        // does not disclose any information.
1558            goto Cleanup;
1559
1560    // b) compute t  := (r + s) mod n
1561        if(!BN_mod_add(bnT, bnR, bnS, &group->order, context))
1562            FAIL(FATAL_ERROR_INTERNAL);
1563    #ifdef _SM2_SIGN_DEBUG
1564        pAssert(cmp_bn2hex(bnT,
1565                    "2B75F07ED7ECE7CCC1C8986B991F441AD324D6D619FE06DD63ED32E0C997C801")
1566                == 0);
1567    #endif
1568
1569    // c) verify that t > 0
1570        if(BN_is_zero(bnT)) {
1571            fail = TRUE;
1572            // set to a value that should allow rest of the computations to run without
```

```
1573            // trouble
1574            BN_copy(bnT, bnS);
1575        }
1576    // d) compute (x, y) := [s]G + [t]Q
1577        if(!EC_POINT_mul(group, pQ, bnS, pQ, bnT, context))
1578            FAIL(FATAL_ERROR_INTERNAL);
1579        // Get the x coordinate of the point
1580        if(!EC_POINT_get_affine_coordinates_GFp(group, pQ, bnT, NULL, context))
1581            FAIL(FATAL_ERROR_INTERNAL);
1582
1583    #ifdef  _SM2_SIGN_DEBUG
1584        pAssert(cmp_bn2hex(bnT,
1585                    "110FCDA57615705D5E7B9324AC4B856D23E6D9188B2AE47759514657CE25D112")
1586                    == 0);
1587    #endif
1588
1589    // e) compute r' := (e + x) mod n (the x coordinate is in bnT)
1590        if(!BN_mod_add(bnRp, bnE, bnT, &group->order, context))
1591            FAIL(FATAL_ERROR_INTERNAL);
1592
1593    // f) verify that r' = r
1594        fail = BN_ucmp(bnR, bnRp) != 0 || fail;
1595
1596    Cleanup:
1597        if(pQ) EC_POINT_free(pQ);
1598        if(group) EC_GROUP_free(group);
1599        BN_CTX_end(context);
1600        BN_CTX_free(context);
1601
1602        if(fail)
1603            return CRYPT_FAIL;
1604        else
1605            return CRYPT_SUCCESS;
1606    }
1607    #endif //% TPM_ALG_SM2
```

B.13.3.2.25. _cpri__ValidateSignatureEcc()

This function validates

Return Value	Meaning
CRYPT_SUCCESS	signature is valid
CRYPT_FAIL	not a valid signature
CRYPT_SCHEME	unsupported scheme

```
1608    LIB_EXPORT CRYPT_RESULT
1609    _cpri__ValidateSignatureEcc(
1610        TPM2B_ECC_PARAMETER     *rIn,           // IN: r component of the signature
1611        TPM2B_ECC_PARAMETER     *sIn,           // IN: s component of the signature
1612        TPM_ALG_ID               scheme,        // IN: the scheme selector
1613        TPM_ALG_ID               hashAlg,       // IN: the hash algorithm used (not used
1614                                                //     in all schemes)
1615        TPM_ECC_CURVE            curveId,       // IN: the curve used in the signature
1616                                                //     process
1617        TPMS_ECC_POINT          *Qin,           // IN: the public point of the key
1618        TPM2B                   *digest         // IN: the digest that was signed
1619        )
1620    {
1621        CRYPT_RESULT            retVal;
1622
1623        // return failure if either part of the signature is zero
1624        if(_math__Normalize2B(&rIn->b) == 0 || _math__Normalize2B(&sIn->b) == 0)
```

```
1625            return CRYPT_FAIL;
1626
1627        switch (scheme)
1628        {
1629            case TPM_ALG_ECDSA:
1630                retVal = ValidateSignatureEcdsa(rIn, sIn, curveId, Qin, digest);
1631                break;
1632
1633    #ifdef  TPM_ALG_ECSCHNORR
1634            case TPM_ALG_ECSCHNORR:
1635                retVal = ValidateSignatureEcSchnorr(rIn, sIn, hashAlg, curveId, Qin,
1636                                                    digest);
1637                break;
1638    #endif
1639
1640    #ifdef TPM_ALG_SM2
1641            case TPM_ALG_SM2:
1642                retVal = ValidateSignatureSM2Dsa(rIn, sIn, curveId, Qin, digest);
1643    #endif
1644            default:
1645                retVal = CRYPT_SCHEME;
1646                break;
1647        }
1648        return retVal;
1649    }
1650    #if CC_ZGen_2Phase == YES //%
1651    #ifdef TPM_ALG_ECMQV
```

B.13.3.2.26. avf1()

This function does the associated value computation required by MQV key exchange. Process:

a) Convert *xQ* to an integer xqi using the convention specified in Appendix C.3.

b) Calculate xqm = xqi mod 2^ceil(f/2) (where f = ceil(log2(n)).

c) Calculate the associate value function avf(Q) = xqm + 2ceil(f / 2)

```
1652    static BOOL
1653    avf1(
1654        BIGNUM          *bnX,           // IN/OUT: the reduced value
1655        BIGNUM          *bnN            // IN: the order of the curve
1656        )
1657    {
1658    // compute f = 2^(ceil(ceil(log2(n)) / 2))
1659        int                         f = (BN_num_bits(bnN) + 1) / 2;
1660    // x' = 2^f + (x mod 2^f)
1661        BN_mask_bits(bnX, f);   // This is mod 2*2^f but it doesn't matter because
1662                                // the next operation will SET the extra bit anyway
1663        BN_set_bit(bnX, f);
1664        return TRUE;
1665    }
```

B.13.3.2.27. C_2_2_MQV()

This function performs the key exchange defined in SP800-56A 6.1.1.4 Full MQV, C(2, 2, ECC MQV).

CAUTION: Implementation of this function may require use of essential claims in patents not owned by TCG members.

Points QsB() and QeB() are required to be on the curve of *inQsA*. The function will fail, possibly catastrophically, if this is not the case.

Return Value	Meaning
CRYPT_SUCCESS	results is valid
CRYPT_NO_RESULT	the value for *dsA* does not give a valid point on the curve

```
1666    static CRYPT_RESULT
1667    C_2_2_MQV(
1668        TPMS_ECC_POINT          *outZ,          // OUT: the computed point
1669        TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
1670        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1671        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1672        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1673        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1674        )
1675    {
1676        BN_CTX                  *context;
1677        EC_POINT                *pQeA = NULL;
1678        EC_POINT                *pQeB = NULL;
1679        EC_POINT                *pQsB = NULL;
1680        EC_GROUP                *group = NULL;
1681        BIGNUM                  *bnTa;
1682        BIGNUM                  *bnDeA;
1683        BIGNUM                  *bnDsA;
1684        BIGNUM                  *bnXeA;         // x coordinate of ephemeral party A key
1685        BIGNUM                  *bnH;
1686        BIGNUM                  *bnN;
1687        BIGNUM                  *bnXeB;
1688        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1689        CRYPT_RESULT            retVal;
1690
1691        pAssert(    curveData != NULL && outZ != NULL && dsA != NULL
1692                &&        deA != NULL &&  QsB != NULL && QeB != NULL);
1693
1694        context = BN_CTX_new();
1695        if(context == NULL || curveData == NULL)
1696            FAIL(FATAL_ERROR_ALLOCATION);
1697        BN_CTX_start(context);
1698        bnTa = BN_CTX_get(context);
1699        bnDeA = BN_CTX_get(context);
1700        bnDsA = BN_CTX_get(context);
1701        bnXeA = BN_CTX_get(context);
1702        bnH = BN_CTX_get(context);
1703        bnN = BN_CTX_get(context);
1704        bnXeB = BN_CTX_get(context);
1705        if(bnXeB == NULL)
1706            FAIL(FATAL_ERROR_ALLOCATION);
1707
1708    // Process:
1709    //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
1710    //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
1711    //  3. If P = O, output an error indicator.
1712    //  4. Z=xP, where xP is the x-coordinate of P.
1713
1714        // Initialize group parameters and local values of input
1715        if((group = EccCurveInit(curveId, context)) == NULL)
1716            FAIL(FATAL_ERROR_INTERNAL);
1717
1718        if((pQeA = EC_POINT_new(group)) == NULL)
1719            FAIL(FATAL_ERROR_ALLOCATION);
1720
1721        BnFrom2B(bnDeA, &deA->b);
1722        BnFrom2B(bnDsA, &dsA->b);
1723        BnFrom2B(bnH, curveData->h);
1724        BnFrom2B(bnN, curveData->n);
```

```
1725        BnFrom2B(bnXeB, &QeB->x.b);
1726        pQeB = EccInitPoint2B(group, QeB, context);
1727        pQsB = EccInitPoint2B(group, QsB, context);
1728
1729        // Compute the public ephemeral key pQeA = [de,A]G
1730        if(    (retVal = PointMul(group, pQeA, bnDeA, NULL, NULL, context))
1731           !=  CRYPT_SUCCESS)
1732            goto Cleanup;
1733
1734        if(EC_POINT_get_affine_coordinates_GFp(group, pQeA, bnXeA, NULL, context) != 1)
1735                FAIL(FATAL_ERROR_INTERNAL);
1736
1737    //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
1738    // tA := (ds,A + de,A  avf(Xe,A)) mod n (3)
1739    // Compute 'tA' = ('deA' +  'dsA'  avf('XeA')) mod n
1740        // Ta = avf(XeA);
1741        BN_copy(bnTa, bnXeA);
1742        avf1(bnTa, bnN);
1743        if(// do Ta = ds,A * Ta mod n = dsA * avf(XeA) mod n
1744              !BN_mod_mul(bnTa, bnDsA, bnTa, bnN, context)
1745
1746           // now Ta = deA + Ta mod n =  deA + dsA * avf(XeA) mod n
1747           || !BN_mod_add(bnTa, bnDeA, bnTa, bnN, context)
1748          )
1749                FAIL(FATAL_ERROR_INTERNAL);
1750
1751    //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
1752    // Put this in because almost every case of h is == 1 so skip the call when
1753        // not necessary.
1754        if(!BN_is_one(bnH))
1755        {
1756            // Cofactor is not 1 so compute Ta := Ta * h mod n
1757            if(!BN_mul(bnTa, bnTa, bnH, context))
1758                FAIL(FATAL_ERROR_INTERNAL);
1759        }
1760
1761        // Now that 'tA' is (h * 'tA' mod n)
1762        // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
1763
1764        // first, compute XeB = avf(XeB)
1765        avf1(bnXeB, bnN);
1766
1767        // QsB := [XeB]QsB
1768        if(     !EC_POINT_mul(group, pQsB, NULL, pQsB, bnXeB, context)
1769
1770            // QeB := QsB + QeB
1771            ||  !EC_POINT_add(group, pQeB, pQeB, pQsB, context)
1772           )
1773            FAIL(FATAL_ERROR_INTERNAL);
1774
1775        // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
1776        if(PointMul(group, pQeB, NULL, pQeB, bnTa, context) == CRYPT_SUCCESS)
1777            // Convert BIGNUM E to TPM2B E
1778            Point2B(group, outZ, pQeB, (INT16)BN_num_bytes(bnN), context);
1779
1780    Cleanup:
1781        if(pQeA != NULL) EC_POINT_free(pQeA);
1782        if(pQeB != NULL) EC_POINT_free(pQeB);
1783        if(pQsB != NULL) EC_POINT_free(pQsB);
1784        if(group != NULL) EC_GROUP_free(group);
1785        BN_CTX_end(context);
1786        BN_CTX_free(context);
1787
1788        return retVal;
1789
1790    }
```

```
#endif // TPM_ALG_ECMQV
#ifdef TPM_ALG_SM2 //%
```

B.13.3.2.28. 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.

```
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 wiht avf1, this is too big by a factor of 2 but
                            // it doesn't matter becasue 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* avf(Xe,A)) mod n Then, compute the Z value from *outZ* = (*h tA* mod *n*) (*QsA* + [avf*(QeB*().x)]*(QeB*())). The function will compute the ephemeral public key from the ephemeral private key. All points are required to be on the curve of *inQsA*. The function will fail catastrophically if this is not the case

Return Value	Meaning
CRYPT_SUCCESS	results is valid
CRYPT_NO_RESULT	the value for *dsA* does not give a valid point on the curve

```
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;
```

```
1831        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1832        CRYPT_RESULT            retVal;
1833
1834        pAssert(    curveData != NULL && outZ != NULL && dsA != NULL
1835                &&        deA != NULL &&  QsB != NULL && QeB != NULL);
1836
1837        context = BN_CTX_new();
1838        if(context == NULL || curveData == NULL)
1839            FAIL(FATAL_ERROR_ALLOCATION);
1840        BN_CTX_start(context);
1841        bnTa = BN_CTX_get(context);
1842        bnDeA = BN_CTX_get(context);
1843        bnDsA = BN_CTX_get(context);
1844        bnXeA = BN_CTX_get(context);
1845        bnH = BN_CTX_get(context);
1846        bnN = BN_CTX_get(context);
1847        bnXeB = BN_CTX_get(context);
1848        if(bnXeB == NULL)
1849            FAIL(FATAL_ERROR_ALLOCATION);
1850
1851        // Initialize group parameters and local values of input
1852        if((group = EccCurveInit(curveId, context)) == NULL)
1853            FAIL(FATAL_ERROR_INTERNAL);
1854
1855        if((pQeA = EC_POINT_new(group)) == NULL)
1856            FAIL(FATAL_ERROR_ALLOCATION);
1857
1858        BnFrom2B(bnDeA, &deA->b);
1859        BnFrom2B(bnDsA, &dsA->b);
1860        BnFrom2B(bnH, curveData->h);
1861        BnFrom2B(bnN, curveData->n);
1862        BnFrom2B(bnXeB, &QeB->x.b);
1863        pQeB = EccInitPoint2B(group, QeB, context);
1864        pQsB = EccInitPoint2B(group, QsB, context);
1865
1866        // Compute the public ephemeral key pQeA = [de,A]G
1867        if(     (retVal = PointMul(group, pQeA, bnDeA, NULL, NULL, context))
1868           !=  CRYPT_SUCCESS)
1869            goto Cleanup;
1870
1871        if(EC_POINT_get_affine_coordinates_GFp(group, pQeA, bnXeA, NULL, context) != 1)
1872                FAIL(FATAL_ERROR_INTERNAL);
1873
1874    // tA := (ds,A + de,A  avf(Xe,A)) mod n (3)
1875    // Compute 'tA' = ('dsA' +  'deA'  avf('XeA')) mod n
1876        // Ta = avf(XeA);
1877        BN_copy(bnTa, bnXeA);
1878        avfSm2(bnTa, bnN);
1879        if(// do Ta = de,A * Ta mod n = deA * avf(XeA) mod n
1880              !BN_mod_mul(bnTa, bnDeA, bnTa, bnN, context)
1881
1882           // now Ta = dsA + Ta mod n =  dsA + deA * avf(XeA) mod n
1883           || !BN_mod_add(bnTa, bnDsA, bnTa, bnN, context)
1884          )
1885                FAIL(FATAL_ERROR_INTERNAL);
1886
1887    // outZ ? [h  tA mod n] (Qs,B + [avf(Xe,B)](Qe,B)) (4)
1888        // Put this in because almost every case of h is == 1 so skip the call when
1889        // not necessary.
1890        if(!BN_is_one(bnH))
1891        {
1892            // Cofactor is not 1 so compute Ta := Ta * h mod n
1893            if(!BN_mul(bnTa, bnTa, bnH, context))
1894                FAIL(FATAL_ERROR_INTERNAL);
1895        }
1896
```

```
1897        // Now that 'tA' is (h * 'tA' mod n)
1898        // 'outZ' = ['tA'](QsB + [avf(QeB.x)](QeB)).
1899
1900        // first, compute XeB = avf(XeB)
1901        avfSm2(bnXeB, bnN);
1902
1903        // QeB := [XeB]QeB
1904        if(     !EC_POINT_mul(group, pQeB, NULL, pQeB, bnXeB, context)
1905
1906            // QeB := QsB + QeB
1907            ||  !EC_POINT_add(group, pQeB, pQeB, pQsB, context)
1908           )
1909            FAIL(FATAL_ERROR_INTERNAL);
1910
1911        // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
1912        if(PointMul(group, pQeB, NULL, pQeB, bnTa, context) == CRYPT_SUCCESS)
1913            // Convert BIGNUM E to TPM2B E
1914            Point2B(group, outZ, pQeB, (INT16)BN_num_bytes(bnN), context);
1915
1916    Cleanup:
1917        if(pQeA != NULL) EC_POINT_free(pQeA);
1918        if(pQeB != NULL) EC_POINT_free(pQeB);
1919        if(pQsB != NULL) EC_POINT_free(pQsB);
1920        if(group != NULL) EC_GROUP_free(group);
1921        BN_CTX_end(context);
1922        BN_CTX_free(context);
1923
1924        return retVal;
1925
1926    }
1927    #endif   //% TPM_ALG_SM2
```

B.13.3.2.29. C_2_2_ECDH()

This function performs the two phase key exchange defined in SP800-56A, 6.1.1.2 Full Unified Model, C(2, 2, ECC CDH).

```
1928    static CRYPT_RESULT
1929    C_2_2_ECDH(
1930        TPMS_ECC_POINT          *outZ1,         // OUT: Zs
1931        TPMS_ECC_POINT          *outZ2,         // OUT: Ze
1932        TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
1933        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1934        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1935        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1936        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1937        )
1938    {
1939        BN_CTX                  *context;
1940        EC_POINT                *pQ = NULL;
1941        EC_GROUP                *group = NULL;
1942        BIGNUM                  *bnD;
1943        INT16                    size;
1944        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1945
1946        context = BN_CTX_new();
1947        if(context == NULL || curveData == NULL)
1948            FAIL(FATAL_ERROR_ALLOCATION);
1949        BN_CTX_start(context);
1950        if((bnD = BN_CTX_get(context)) == NULL)
1951            FAIL(FATAL_ERROR_INTERNAL);
1952
1953        // Initialize group parameters and local values of input
1954        if((group = EccCurveInit(curveId, context)) == NULL)
```

```
1955            FAIL(FATAL_ERROR_INTERNAL);
1956        size = (INT16)BN_num_bytes(&group->order);
1957
1958        // Get the static private key of A
1959        BnFrom2B(bnD, &dsA->b);
1960
1961        // Initialize the static public point from B
1962        pQ = EccInitPoint2B(group, QsB, context);
1963
1964        // Do the point multiply for the Zs value
1965        if(PointMul(group, pQ, NULL, pQ, bnD, context) != CRYPT_NO_RESULT)
1966            // Convert the Zs value
1967            Point2B(group, outZ1, pQ, size, context);
1968
1969        // Get the ephemeral private key of A
1970        BnFrom2B(bnD, &deA->b);
1971
1972        // Initalize the ephemeral public point from B
1973        PointFrom2B(group, pQ, QeB, context);
1974
1975        // Do the point multiply for the Ze value
1976        if(PointMul(group, pQ, NULL, pQ, bnD, context) != CRYPT_NO_RESULT)
1977            // Convert the Ze value.
1978            Point2B(group, outZ2, pQ, size, context);
1979
1980        if(pQ != NULL) EC_POINT_free(pQ);
1981        if(group != NULL) EC_GROUP_free(group);
1982        BN_CTX_end(context);
1983        BN_CTX_free(context);
1984        return CRYPT_SUCCESS;
1985    }
```

B.13.3.2.30. _cpri__C_2_2_KeyExchange()

This function is the dispatch routine for the EC key exchange function that use two ephemeral and two static keys.

Return Value	Meaning
CRYPT_SCHEME	scheme is not defined

```
1986    LIB_EXPORT CRYPT_RESULT
1987    _cpri__C_2_2_KeyExchange(
1988        TPMS_ECC_POINT          *outZ1,         // OUT: a computed point
1989        TPMS_ECC_POINT          *outZ2,         // OUT: and optional second point
1990        TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
1991        TPM_ALG_ID               scheme,        // IN: the key exchange scheme
1992        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1993        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1994        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1995        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1996        )
1997    {
1998        pAssert(   outZ1 != NULL
1999                && dsA != NULL && deA != NULL
2000                && QsB != NULL && QeB != NULL);
2001
2002        // Initalize the output points so that they are empty until one of the
2003        // functions decides otherwise
2004        outZ1->x.b.size = 0;
2005        outZ1->y.b.size = 0;
2006        if(outZ2 != NULL)
2007        {
2008            outZ2->x.b.size = 0;
```

```
2009            outZ2->y.b.size = 0;
2010        }
2011
2012        switch (scheme)
2013        {
2014            case TPM_ALG_ECDH:
2015                return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
2016                break;
2017    #ifdef  TPM_ALG_ECMQV
2018            case TPM_ALG_ECMQV:
2019                return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
2020                break;
2021    #endif
2022    #ifdef  TPM_ALG_SM2
2023            case TPM_ALG_SM2:
2024                return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
2025                break;
2026    #endif
2027            default:
2028                return CRYPT_SCHEME;
2029        }
2030    }
2031    #else       //%
```

Stub used when the 2-phase key exchange is not defined so that the linker has something to associate with the value in the .def file.

```
2032    LIB_EXPORT CRYPT_RESULT
2033    _cpri__C_2_2_KeyExchange(
2034        void
2035        )
2036    {
2037        return CRYPT_FAIL;
2038    }
2039    #endif //% CC_ZGen_2Phase
2040    #endif // TPM_ALG_ECC
```

Annex C
(informative)
Simulation Environment

C.1 Introduction

These files are used to simulate some of the implementation-dependent hardware of a TPM. These files are provided to allow creation of a simulation environment for the TPM. These files are not expected to be part of a hardware TPM implementation.

C.2 Cancel.c

C.2.1. Introduction

This module simulates the cancel pins on the TPM.

C.2.2. Includes, Typedefs, Structures, and Defines

```
1   #include "PlatformData.h"
```

C.2.3. Functions

C.2.3.1. _plat__IsCanceled()

Check if the cancel flag is set

Return Value	Meaning
TRUE	if cancel flag is set
FALSE	if cancel flag is not set

```
2   LIB_EXPORT BOOL
3   _plat__IsCanceled(
4       void
5       )
6   {
7       // return cancel flag
8       return s_isCanceled;
9   }
```

C.2.3.2. _plat__SetCancel()

Set cancel flag.

```
10  LIB_EXPORT void
11  _plat__SetCancel(
12      void
13      )
14  {
15      s_isCanceled = TRUE;
16      return;
17  }
```

C.2.3.3. _plat__ClearCancel()

Clear cancel flag

```
18  LIB_EXPORT void
19  _plat__ClearCancel(
20      void
21      )
22  {
23      s_isCanceled = FALSE;
24      return;
25  }
```

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

```
#include <time.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

```
LIB_EXPORT 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.

```
unsigned long long
_plat__ClockTimeFromStart(
    void
    )
{
    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

```
LIB_EXPORT 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_MULTIPLER 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

```
LIB_EXPORT void
_plat__ClockAdjustRate(
    int              adjust         // IN: the adjust number.  It could be positive
                                    //     or negative
    )
{
    // We expect the caller should only use a fixed set of constant values to
    // adjust the rate
    switch(adjust)
    {
        case CLOCK_ADJUST_COARSE:
            s_adjustRate += CLOCK_ADJUST_COARSE;
            break;
        case -CLOCK_ADJUST_COARSE:
            s_adjustRate -= CLOCK_ADJUST_COARSE;
            break;
        case CLOCK_ADJUST_MEDIUM:
            s_adjustRate += CLOCK_ADJUST_MEDIUM;
            break;
        case -CLOCK_ADJUST_MEDIUM:
            s_adjustRate -= CLOCK_ADJUST_MEDIUM;
            break;
        case CLOCK_ADJUST_FINE:
            s_adjustRate += CLOCK_ADJUST_FINE;
            break;
        case -CLOCK_ADJUST_FINE:
            s_adjustRate -= CLOCK_ADJUST_FINE;
            break;
        default:
            // ignore any other values;
            break;
    }

    if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
        s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
    if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
        s_adjustRate = CLOCK_NOMINAL-CLOCK_ADJUST_LIMIT;

    return;
}
```

C.4 Entropy.c

C.4.1. Includes

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdint.h>
#include <memory.h>
#include "TpmBuildSwitches.h"
```

C.4.2. Local values

This is the last 32-bits of hardware entropy produced. We have to check to see that two consecutive 32-bit values are not the same because (according to FIPS 140-2, annex C

> "If each call to a RNG produces blocks of n bits (where n > 15), the first n-bit block generated after power-up, initialization, or reset shall not be used, but shall be saved for comparison with the next n-bit block to be generated. Each subsequent generation of an n-bit block shall be compared with the previously generated block. The test shall fail if any two compared n-bit blocks are equal."

```
extern uint32_t         lastEntropy;
extern int              firstValue;
```

C.4.3. _plat__GetEntropy()

This function is used to get available hardware entropy. In a hardware implementation of this function, there would be no call to the system to get entropy. If the caller does not ask for any entropy, then this is a startup indication and *firstValue* should be reset.

Return Value	Meaning
< 0	hardware failure of the entropy generator, this is sticky
>= 0	the returned amount of entropy (bytes)

```
LIB_EXPORT int32_t
_plat__GetEntropy(
    unsigned char       *entropy,           // output buffer
    uint32_t             amount             // amount requested
)
{
    uint32_t            rndNum;
    int                 OK = 1;

    if(amount == 0)
    {
        firstValue = 1;
        return 0;
    }

    // Only provide entropy 32 bits at a time to test the ability
    // of the caller to deal with partial results.
    OK = rand_s(&rndNum) == 0;
    if(OK)
    {
        if(firstValue)
            firstValue = 0;
        else
            OK = (rndNum != lastEntropy);
    }
```

```
33          if(OK)
34          {
35              lastEntropy = rndNum;
36              if(amount > sizeof(rndNum))
37                  amount = sizeof(rndNum);
38              memcpy(entropy, &rndNum, amount);
39          }
40          return (OK) ? (int32_t)amount : -1;
41      }
```

C.5 LocalityPlat.c

C.5.1. Includes

```
#include "PlatformData.h"
#include "TpmError.h"
```

C.5.2. Functions

C.5.2.1. _plat__LocalityGet()

Get the most recent command locality in locality value form. This is an integer value for locality and not a locality structure The locality can be 0-4 or 32-255. 5-31 is not allowed.

```
LIB_EXPORT unsigned char
_plat__LocalityGet(
    void
    )
{
    return s_locality;
}
```

C.5.2.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
LIB_EXPORT void
_plat__LocalitySet(
    unsigned char    locality
    )
{
    if(locality > 4 && locality < 32)
        locality = 0;
    s_locality = locality;
    return;
}
```

C.5.2.3. _plat__IsRsaKeyCacheEnabled()

This function is used to check if the RSA key cache is enabled or not.

```
LIB_EXPORT int
_plat__IsRsaKeyCacheEnabled(
    void
    )
{
    return s_RsaKeyCacheEnabled;
}
```

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 <memory.h>
#include <string.h>
#include "PlatformData.h"
#include "TpmError.h"
#include "assert.h"
```

C.6.3. Functions

C.6.3.1. _plat__NvErrors()

This function is used by the simulator to set the error flags in the NV subsystem to simulate an error in the NV loading process

```
LIB_EXPORT void
_plat__NvErrors(
    BOOL             recoverable,
    BOOL             unrecoverable
    )
{
    s_NV_unrecoverable = unrecoverable;
    s_NV_recoverable = recoverable;
}
```

C.6.3.2. _plat__NVEnable()

Enable NV memory.

This version just pulls in data from a file. In a real TPM, with NV on chip, this function would verify the integrity of the saved context. If the NV memory was not on chip but was in something like RPMB, the NV state would be read in, decrypted and integrity checked.

The recovery from an integrity failure depends on where the error occurred. It it was in the state that is discarded by TPM Reset, then the error is recoverable if the TPM is reset. Otherwise, the TPM must go into failure mode.

Return Value	Meaning
0	if success
> 0	if receive recoverable error
<0	if unrecoverable error

```
LIB_EXPORT int
_plat__NVEnable(
    void            *platParameter  // IN: platform specific parameter
    )
{
    (platParameter);                    // to keep compiler quiet
    // Start assuming everything is OK
```

```
22      s_NV_unrecoverable = FALSE;
23      s_NV_recoverable = FALSE;
24
25  #ifdef FILE_BACKED_NV
26
27      if(s_NVFile != NULL) return 0;
28
29      // Try to open an exist NVChip file for read/write
30      if(0 != fopen_s(&s_NVFile, "NVChip", "r+b"))
31          s_NVFile = NULL;
32
33      if(NULL != s_NVFile)
34      {
35          // See if the NVChip file is empty
36          fseek(s_NVFile, 0, SEEK_END);
37          if(0 == ftell(s_NVFile))
38              s_NVFile = NULL;
39      }
40
41      if(s_NVFile == NULL)
42      {
43          // Initialize all the byte in the new file to 0
44          memset(s_NV, 0, NV_MEMORY_SIZE);
45
46          // If NVChip file does not exist, try to create it for read/write
47          fopen_s(&s_NVFile, "NVChip", "w+b");
48          // Start initialize at the end of new file
49          fseek(s_NVFile, 0, SEEK_END);
50          // Write 0s to NVChip file
51          fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);
52      }
53      else
54      {
55          // If NVChip file exist, assume the size is correct
56          fseek(s_NVFile, 0, SEEK_END);
57          assert(ftell(s_NVFile) == NV_MEMORY_SIZE);
58          // read NV file data to memory
59          fseek(s_NVFile, 0, SEEK_SET);
60          fread(s_NV, NV_MEMORY_SIZE, 1, s_NVFile);
61      }
62  #endif
63      // NV contents have been read and the error checks have been performed. For
64      // simulation purposes, use the signaling interface to indicate if an error is
65      // to be simulated and the type of the error.
66      if(s_NV_unrecoverable)
67          return -1;
68      return s_NV_recoverable;
69  }
```

C.6.3.3. _plat__NVDisable()

Disable NV memory

```
70  LIB_EXPORT void
71  _plat__NVDisable(
72      void
73      )
74  {
75  #ifdef  FILE_BACKED_NV
76
77      assert(s_NVFile != NULL);
78      // Close NV file
79      fclose(s_NVFile);
80      // Set file handle to NULL
```

```
81      s_NVFile = NULL;
82
83  #endif
84
85      return;
86  }
```

C.6.3.4. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
87  LIB_EXPORT int
88  _plat__IsNvAvailable(
89      void
90      )
91  {
92      // NV is not available if the TPM is in failure mode
93      if(!s_NvIsAvailable)
94          return 1;
95
96  #ifdef FILE_BACKED_NV
97      if(s_NVFile == NULL)
98          return 1;
99  #endif
100
101     return 0;
102
103 }
```

C.6.3.5. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```
104 LIB_EXPORT void
105 _plat__NvMemoryRead(
106     unsigned int     startOffset,   // IN: read start
107     unsigned int     size,          // IN: size of bytes to read
108     void            *data           // OUT: data buffer
109     )
110 {
111     assert(startOffset + size <= NV_MEMORY_SIZE);
112
113     // Copy data from RAM
114     memcpy(data, &s_NV[startOffset], size);
115     return;
116 }
```

C.6.3.6. _plat__NvIsDifferent()

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE	the NV location is different from the test value
FALSE	the NV location is the same as the test value

```
117  LIB_EXPORT BOOL
118  _plat__NvIsDifferent(
119      unsigned int     startOffset,   // IN: read start
120      unsigned int     size,          // IN: size of bytes to read
121      void            *data           // IN: data buffer
122      )
123  {
124      return (memcmp(&s_NV[startOffset], data, size) != 0);
125  }
```

C.6.3.7. _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.

```
126  LIB_EXPORT void
127  _plat__NvMemoryWrite(
128      unsigned int     startOffset,   // IN: write start
129      unsigned int     size,          // IN: size of bytes to write
130      void            *data           // OUT: data buffer
131      )
132  {
133      assert(startOffset + size <= NV_MEMORY_SIZE);
134
135      // Copy the data to the NV image
136      memcpy(&s_NV[startOffset], data, size);
137  }
```

C.6.3.8. _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

```
138  LIB_EXPORT void
139  _plat__NvMemoryMove(
140      unsigned int     sourceOffset,  // IN: source offset
141      unsigned int     destOffset,    // IN: destination offset
142      unsigned int     size           // IN: size of data being moved
143      )
144  {
145      assert(sourceOffset + size <= NV_MEMORY_SIZE);
146      assert(destOffset + size <= NV_MEMORY_SIZE);
147
148      // Move data in RAM
149      memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);
150
151      return;
152  }
```

C.6.3.9. _plat__NvCommit()

Update NV chip

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
153  LIB_EXPORT int
154  _plat__NvCommit(
155      void
156      )
157  {
158  #ifdef FILE_BACKED_NV
159      // If NV file is not available, return failure
160      if(s_NVFile == NULL)
161          return 1;
162
163      // Write RAM data to NV
164      fseek(s_NVFile, 0, SEEK_SET);
165      fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);
166      return 0;
167  #else
168      return 0;
169  #endif
170
171  }
```

C.6.3.10. _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

```
172  LIB_EXPORT void
173  _plat__SetNvAvail(
174      void
175      )
176  {
177      s_NvIsAvailable = TRUE;
178      return;
179  }
```

C.6.3.11. _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

```
180  LIB_EXPORT void
181  _plat__ClearNvAvail(
182      void
183      )
184  {
185      s_NvIsAvailable = FALSE;
186      return;
187  }
```

C.7 PowerPlat.c

C.7.1. Includes and Function Prototypes

```
#include    "PlatformData.h"
#include    "Platform.h"
```

C.7.2. Functions

C.7.2.1. _plat__Signal_PowerOn()

Signal platform power on

```
LIB_EXPORT int
_plat__Signal_PowerOn(
    void
    )
{
    // Start clock
    _plat__ClockReset();

    // Initialize locality
    s_locality = 0;

    // Command cancel
     s_isCanceled = FALSE;

    // Need to indicate that we lost power
    s_powerLost = TRUE;

    return 0;
}
```

C.7.2.2. _plat__WasPowerLost()

Test whether power was lost before a _TPM_Init()

```
LIB_EXPORT BOOL
_plat__WasPowerLost(
    BOOL             clear
    )
{
    BOOL        retVal = s_powerLost;
    if(clear)
        s_powerLost = FALSE;
    return retVal;
}
```

C.7.2.3. _plat_Signal_Reset()

This a TPM reset without a power loss.

```
LIB_EXPORT int
_plat__Signal_Reset(
    void
    )
{
    // Need to reset the clock
    _plat__ClockReset();
```

```
39
40      // if we are doing reset but did not have a power failure, then we should
41      // not need to reload NV ...
42      return 0;
43  }
```

C.7.2.4. _plat__Signal_PowerOff()

Signal platform power off

```
44  LIB_EXPORT void
45  _plat__Signal_PowerOff(
46      void
47      )
48  {
49      // Prepare NV memory for power off
50      _plat__NVDisable();
51
52      return;
53  }
```

C.8 Platform.h

```
#ifndef     PLATFORM_H
#define     PLATFORM_H
```

C.8.1. Includes and Defines

```
#include "bool.h"
#include "stdint.h"
#include "TpmError.h"
#include "TpmBuildSwitches.h"
#define UNREFERENCED(a) ((void)(a))
```

C.8.2. Power Functions

C.8.2.1. _plat__Signal_PowerOn

Signal power on This signal is simulate by a RPC call

```
LIB_EXPORT int
_plat__Signal_PowerOn(void);
```

C.8.2.2. _plat__Signal_Reset

Signal reset This signal is simulate by a RPC call

```
LIB_EXPORT int
_plat__Signal_Reset(void);
```

C.8.2.3. _plat__WasPowerLost()

Indicates if the power was lost before a _TPM__Init().

```
LIB_EXPORT BOOL
_plat__WasPowerLost(BOOL clear);
```

C.8.2.4. _plat__Signal_PowerOff()

Signal power off This signal is simulate by a RPC call

```
LIB_EXPORT void
_plat__Signal_PowerOff(void);
```

C.8.3. Physical Presence Functions

C.8.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

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

```
16  LIB_EXPORT BOOL
17  _plat__PhysicalPresenceAsserted(void);
```

C.8.3.2. _plat__Signal_PhysicalPresenceOn

Signal physical presence on This signal is simulate by a RPC call

```
18  LIB_EXPORT void
19  _plat__Signal_PhysicalPresenceOn(void);
```

C.8.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off This signal is simulate by a RPC call

```
20  LIB_EXPORT void
21  _plat__Signal_PhysicalPresenceOff(void);
```

C.8.4. Command Canceling Functions

C.8.4.1. _plat__IsCanceled()

Check if the cancel flag is set

Return Value	Meaning
TRUE	if cancel flag is set
FALSE	if cancel flag is not set

```
22  LIB_EXPORT BOOL
23  _plat__IsCanceled(void);
```

C.8.4.2. _plat__SetCancel()

Set cancel flag.

```
24  LIB_EXPORT void
25  _plat__SetCancel(void);
```

C.8.4.3. _plat__ClearCancel()

Clear cancel flag

```
26  LIB_EXPORT void
27  _plat__ClearCancel( void);
```

C.8.5. NV memory functions

C.8.5.1. _plat__NvErrors()

This function is used by the simulator to set the error flags in the NV subsystem to simulate an error in the NV loading process

```
28  LIB_EXPORT void
29  _plat__NvErrors(
30      BOOL        recoverable,
31      BOOL        unrecoverable
32      );
```

C.8.5.2. _plat__NVEnable()

Enable platform NV memory NV memory is automatically enabled at power on event. This function is mostly for TPM_Manufacture() to access NV memory without a power on event

Return Value	Meaning
0	if success
non-0	if fail

```
33  LIB_EXPORT int
34  _plat__NVEnable(
35      void    *platParameter              // IN: platform specific parameters
36  );
```

C.8.5.3. _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

```
37  LIB_EXPORT void
38  _plat__NVDisable(void);
```

C.8.5.4. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
39  LIB_EXPORT int
40  _plat__IsNvAvailable(void);
```

C.8.5.5. _plat__NvCommit()

Update NV chip

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
LIB_EXPORT int
_plat__NvCommit(void);
```

C.8.5.6. _plat__NvMemoryRead()

Read a chunk of NV memory

```
LIB_EXPORT void
_plat__NvMemoryRead(
    unsigned int        startOffset,        // IN: read start
    unsigned int        size,               // IN: size of bytes to read
    void                *data               // OUT: data buffer
);
```

C.8.5.7. _plat__NvIsDifferent()

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE	the NV location is different from the test value
FALSE	the NV location is the same as the test value

```
LIB_EXPORT BOOL
_plat__NvIsDifferent(
    unsigned int         startOffset,         // IN: read start
    unsigned int         size,                // IN: size of bytes to compare
    void                *data                 // IN: data buffer
    );
```

C.8.5.8. _plat__NvMemoryWrite()

Write a chunk of NV memory

```
LIB_EXPORT void
_plat__NvMemoryWrite(
    unsigned int        startOffset,        // IN: read start
    unsigned int        size,               // IN: size of bytes to read
    void                *data               // OUT: data buffer
);
```

C.8.5.9. _plat__NvMemoryMove()

Move a chunk of NV memory from source to destination This function should ensure that if there overlap, the original data is copied before it is written

```
LIB_EXPORT void
_plat__NvMemoryMove(
    unsigned int        sourceOffset,        // IN: source offset
    unsigned int        destOffset,          // IN: destination offset
    unsigned int        size                 // IN: size of data being moved
```

```
66  );
```

C.8.5.10. _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

```
67  LIB_EXPORT void
68  _plat__SetNvAvail(void);
```

C.8.5.11. _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

```
69  LIB_EXPORT void
70  _plat__ClearNvAvail(void);
```

C.8.6. Locality Functions

C.8.6.1. _plat__LocalityGet()

Get the most recent command locality in locality value form

```
71  LIB_EXPORT unsigned char
72  _plat__LocalityGet(void);
```

C.8.6.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
73  LIB_EXPORT void
74  _plat__LocalitySet(
75      unsigned char   locality
76  );
```

C.8.6.3. _plat__IsRsaKeyCacheEnabled()

This function is used to check if the RSA key cache is enabled or not.

```
77  LIB_EXPORT int
78  _plat__IsRsaKeyCacheEnabled(
79      void
80      );
```

C.8.7. Clock Constants and Functions

Assume that the nominal divisor is 30000

```
81  #define     CLOCK_NOMINAL           30000
```

A 1% change in rate is 300 counts

```
82  #define     CLOCK_ADJUST_COARSE     300
```

A .1 change in rate is 30 counts

```
#define     CLOCK_ADJUST_MEDIUM     30
```

A minimum change in rate is 1 count

```
#define     CLOCK_ADJUST_FINE       1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

```
#define     CLOCK_ADJUST_LIMIT      5000
```

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

```
LIB_EXPORT 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.

```
LIB_EXPORT 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

```
LIB_EXPORT unsigned long long
_plat__ClockTimeElapsed(void);
```

C.8.7.4. _plat__ClockAdjustRate()

Adjust the clock rate

```
LIB_EXPORT void
_plat__ClockAdjustRate(
    int         adjust              // IN: the adjust number.  It could be
                                    // positive or negative
    );
```

C.8.8. Single Function Files

C.8.8.1. _plat__GetEntropy()

This function is used to get available hardware entropy. In a hardware implementation of this function, there would be no call to the system to get entropy. If the caller does not ask for any entropy, then this is a startup indication and *firstValue* should be reset.

Return Value	Meaning
< 0	hardware failure of the entropy generator, this is sticky
>= 0	the returned amount of entropy (bytes)

```
 99  LIB_EXPORT int32_t
100  _plat__GetEntropy(
101      unsigned char       *entropy,           // output buffer
102      uint32_t             amount             // amount requested
103  );
104  #endif
```

C.9 PlatformData.h

This file contains the instance data for the Platform module. It is collected in this file so that the state of the module is easier to manage.

```
#ifndef _PLATFORM_DATA_H_
#define _PLATFORM_DATA_H_
#include    "TpmBuildSwitches.h"
#include    "Implementation.h"
#include    "bool.h"
```

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

```
extern BOOL     s_isCanceled;
```

From Clock.c This variable records the time when _plat__ClockReset() is called. This mechanism allow us to subtract the time when TPM is power off from the total time reported by clock() function

```
extern unsigned long long   s_initClock;
extern unsigned int         s_adjustRate;
```

From LocalityPlat.c Locality of current command

```
extern unsigned char s_locality;
```

From NVMem.c Choose if the NV memory should be backed by RAM or by file. If this macro is defined, then a file is used as NV. If it is not defined, then RAM is used to back NV memory. Comment out to use RAM.

```
#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;
extern BOOL             s_NV_unrecoverable;
extern BOOL             s_NV_recoverable;
```

From PPPlat.c Physical presence. It is initialized to FALSE

```
extern BOOL     s_physicalPresence;
```

From Power

```
extern BOOL        s_powerLost;
```

From Entropy.c

```
extern uint32_t        lastEntropy;
extern int             firstValue;
#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;
```

From Clock.c

```
5   unsigned long long  s_initClock;
6   unsigned int        s_adjustRate;
```

From LocalityPlat.c

```
7   unsigned char       s_locality;
```

From Power.c

```
8   BOOL                s_powerLost;
```

From Entropy.c

```
9   uint32_t            lastEntropy;
10  int                 firstValue;
```

From NVMem.c

```
11  #ifdef  VTPM
12  #   undef FILE_BACKED_NV
13  #endif
14  #ifdef FILE_BACKED_NV
15  FILE               *s_NVFile = NULL;
16  #endif
17  unsigned char       s_NV[NV_MEMORY_SIZE];
18  BOOL                s_NvIsAvailable;
19  BOOL                s_NV_unrecoverable;
20  BOOL                s_NV_recoverable;
```

From PPPlat.c

```
21  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

```
#include "PlatformData.h"
```

C.11.3. Functions

C.11.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

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

```
LIB_EXPORT 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

```
LIB_EXPORT void
_plat__Signal_PhysicalPresenceOn(
    void
    )
{
    s_physicalPresence = TRUE;
    return;
}
```

C.11.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
LIB_EXPORT void
_plat__Signal_PhysicalPresenceOff(
    void
    )
{
    s_physicalPresence = FALSE;
    return;
}
```

C.12 Unique.c

C.12.1. Introduction

In some implementations of the TPM, the hardware can provide a secret value to the TPM. This secret value is statistically unique to the instance of the TPM. Typical uses of this value are to provide personalization to the random number generation and as a shared secret between the TPM and the manufacturer.

C.12.2. Includes

```
#include "stdint.h"
#include "TpmBuildSwitches.h"
const char notReallyUnique[] =
        "This is not really a unique value. A real unique value should"
        " be generated by the platform.";
```

C.12.3. _plat__GetUnique()

This function is used to access the platform-specific unique value. This function places the unique value in the provided buffer *(b*) and returns the number of bytes transferred. The function will not copy more data than *bSize*.

NOTE: If a platform unique value has unequal distribution of uniqueness and *bSize* is smaller than the size of the unique value, the *bSize* portion with the most uniqueness should be returned.

```
LIB_EXPORT uint32_t
_plat__GetUnique(
    uint32_t             which,         // authorities (0) or details
    uint32_t             bSize,         // size of the buffer
    unsigned char       *b              // output buffer
)
{
    const char          *from = notReallyUnique;
    uint32_t             retVal = 0;

    if(which == 0) // the authorities value
    {
        for(retVal = 0;
            *from != 0 &&  retVal < bSize;
            retVal++)
        {
            *b++ = *from++;
        }
    }
    else
    {
#define uSize  sizeof(notReallyUnique)
        b = &b[((bSize < uSize) ? bSize : uSize) - 1];
        for(retVal = 0;
            *from != 0 && retVal < bSize;
            retVal++)
        {
            *b-- = *from++;
        }
    }
    return retVal;
}
```


Annex D
(informative)
Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in TPM 2.0 Part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.

D.2 TpmTcpProtocol.h

D.2.1. Introduction

TPM commands are communicated as BYTE streams on a TCP connection. The TPM command protocol is enveloped with the interface protocol described in this file. The command is indicated by a UINT32 with one of the values below. Most commands take no parameters return no TPM errors. In these cases the TPM interface protocol acknowledges that command processing is complete by returning a UINT32=0. The command TPM_SIGNAL_HASH_DATA takes a UINT32-prepended variable length BYTE array and the interface protocol acknowledges command completion with a UINT32=0. Most TPM commands are enveloped using the TPM_SEND_COMMAND interface command. The parameters are as indicated below. The interface layer also appends a UIN32=0 to the TPM response for regularity.

D.2.2. Typedefs and Defines

```
#ifndef     TCP_TPM_PROTOCOL_H
#define     TCP_TPM_PROTOCOL_H
```

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} ->
        //     {UINT32 OutBufferSize, BYTE[OutBufferSize] OutBuffer}
#define TPM_SIGNAL_CANCEL_ON        9
#define TPM_SIGNAL_CANCEL_OFF       10
#define TPM_SIGNAL_NV_ON            11
#define TPM_SIGNAL_NV_OFF           12
#define TPM_SIGNAL_KEY_CACHE_ON     13
#define TPM_SIGNAL_KEY_CACHE_OFF    14
#define TPM_REMOTE_HANDSHAKE        15
#define TPM_SET_ALTERNATIVE_RESULT  16
#define TPM_SIGNAL_RESET            17
#define TPM_SESSION_END             20
#define TPM_STOP                    21
#define TPM_GET_COMMAND_RESPONSE_SIZES  25
#define TPM_TEST_FAILURE_MODE      30
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;
```

```
42
43  typedef unsigned char *_OUTPUT_BUFFER;
44
45  typedef struct
46  {
47      uint32_t          BufferSize;
48      _OUTPUT_BUFFER    Buffer;
49  } _OUT_BUFFER;
50
51  //** TPM Command Function Prototypes
52  void _rpc__Signal_PowerOn(BOOL isReset);
53  void _rpc__Signal_PowerOff();
54  void _rpc__ForceFailureMode();
55  void _rpc__Signal_PhysicalPresenceOn();
56  void _rpc__Signal_PhysicalPresenceOff();
57  void _rpc__Signal_Hash_Start();
58  void _rpc__Signal_Hash_Data(
59      _IN_BUFFER input
60  );
61  void _rpc__Signal_HashEnd();
62  void _rpc__Send_Command(
63      unsigned char   locality,
64      _IN_BUFFER      request,
65      _OUT_BUFFER     *response
66  );
67  void _rpc__Signal_CancelOn();
68  void _rpc__Signal_CancelOff();
69  void _rpc__Signal_NvOn();
70  void _rpc__Signal_NvOff();
71  BOOL _rpc__InjectEPS(
72      const char* seed,
73      int seedSize
74  );
```

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.

```
75  BOOL TpmServer(SOCKET ServerSocket);
76  #endif
```


D.3 TcpServer.c

D.3.1. Description

This file contains the socket interface to a TPM simulator.

D.3.2. Includes, Locals, Defines and Function Prototypes

```
#include <stdio.h>
#include <windows.h>
#include <winsock.h>
#include "string.h"
#include <stdlib.h>
#include <stdint.h>
#include "TpmTcpProtocol.h"
BOOL ReadBytes(SOCKET s, char* buffer, int NumBytes);
BOOL ReadVarBytes(SOCKET s, char* buffer, UINT32* BytesReceived, int MaxLen);
BOOL WriteVarBytes(SOCKET s, char *buffer, int BytesToSend);
BOOL WriteBytes(SOCKET s, char* buffer, int NumBytes);
BOOL WriteUINT32(SOCKET s, UINT32 val);
#ifndef __IGNORE_STATE__
static UINT32 ServerVersion = 1;
#define MAX_BUFFER 1048576
char InputBuffer[MAX_BUFFER];       //The input data buffer for the simulator.
char OutputBuffer[MAX_BUFFER];      //The output data buffer for the simulator.
struct {
    UINT32      largestCommandSize;
    UINT32      largestCommand;
    UINT32      largestResponseSize;
    UINT32      largestResponse;
} CommandResponseSizes = {0};
#endif // __IGNORE_STATE___
```

D.3.3. Functions

D.3.3.1. CreateSocket()

This function creates a socket listening on *PortNumber*.

```
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);
```

```
46      if(INVALID_SOCKET == *listenSocket)
47      {
48          printf("Cannot create server listen socket.  Error is 0x%x\n",
49                  WSAGetLastError());
50          return -1;
51      }
52
53      // bind the listening socket to the specified port
54      ZeroMemory(&MyAddress, sizeof(MyAddress));
55      MyAddress.sin_port=htons((short) PortNumber);
56      MyAddress.sin_family=AF_INET;
57
58      res= bind(*listenSocket,(struct sockaddr*) &MyAddress,sizeof(MyAddress));
59      if(res==SOCKET_ERROR)
60      {
61          printf("Bind error.  Error is 0x%x\n", WSAGetLastError());
62          return -1;
63      };
64
65      // listen/wait for server connections
66      res= listen(*listenSocket,3);
67      if(res==SOCKET_ERROR)
68      {
69          printf("Listen error.  Error is 0x%x\n", WSAGetLastError());
70          return -1;
71      };
72
73      return 0;
74  }
```

D.3.3.2. PlatformServer()

This function processes incoming platform requests.

```
75  BOOL
76  PlatformServer(
77      SOCKET           s
78      )
79  {
80      BOOL                ok = TRUE;
81      UINT32              length = 0;
82      UINT32              Command;
83
84      for(;;)
85      {
86          ok = ReadBytes(s, (char*) &Command, 4);
87          // client disconnected (or other error).  We stop processing this client
88          // and return to our caller who can stop the server or listen for another
89          // connection.
90          if(!ok) return TRUE;
91          Command = ntohl(Command);
92          switch(Command)
93          {
94              case TPM_SIGNAL_POWER_ON:
95                  _rpc__Signal_PowerOn(FALSE);
96                  break;
97
98              case TPM_SIGNAL_POWER_OFF:
99                  _rpc__Signal_PowerOff();
100                 break;
101
102             case TPM_SIGNAL_RESET:
103                 _rpc__Signal_PowerOn(TRUE);
104                 break;
```

```
105
106             case TPM_SIGNAL_PHYS_PRES_ON:
107                 _rpc__Signal_PhysicalPresenceOn();
108                 break;
109
110             case TPM_SIGNAL_PHYS_PRES_OFF:
111                 _rpc__Signal_PhysicalPresenceOff();
112                 break;
113
114             case TPM_SIGNAL_CANCEL_ON:
115                 _rpc__Signal_CancelOn();
116                 break;
117
118             case TPM_SIGNAL_CANCEL_OFF:
119                 _rpc__Signal_CancelOff();
120                 break;
121
122             case TPM_SIGNAL_NV_ON:
123                 _rpc__Signal_NvOn();
124                 break;
125
126             case TPM_SIGNAL_NV_OFF:
127                 _rpc__Signal_NvOff();
128                 break;
129
130             case TPM_SESSION_END:
131                 // Client signaled end-of-session
132                 return TRUE;
133
134             case TPM_STOP:
135                 // Client requested the simulator to exit
136                 return FALSE;
137
138             case TPM_TEST_FAILURE_MODE:
139                 _rpc__ForceFailureMode();
140                 break;
141
142             case TPM_GET_COMMAND_RESPONSE_SIZES:
143                 ok = WriteVarBytes(s, (char *)&CommandResponseSizes,
144                                    sizeof(CommandResponseSizes));
145                 memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
146                 if(!ok)
147                     return TRUE;
148                 break;
149
150             default:
151                 printf("Unrecognized platform interface command %d\n", Command);
152                 WriteUINT32(s, 1);
153                 return TRUE;
154         }
155         WriteUINT32(s,0);
156     }
157     return FALSE;
158 }
```

D.3.3.3. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

```
159 DWORD WINAPI
160 PlatformSvcRoutine(
161     LPVOID          port
162     )
163 {
```

```
164     int                 PortNumber = (int)(INT_PTR) port;
165     SOCKET              listenSocket, serverSocket;
166     struct              sockaddr_in HerAddress;
167     int                 res;
168     int                 length;
169     BOOL                continueServing;
170
171     res = CreateSocket(PortNumber, &listenSocket);
172     if(res != 0)
173     {
174         printf("Create platform service socket fail\n");
175         return res;
176     }
177
178     // Loop accepting connections one-by-one until we are killed or asked to stop
179     // Note the platform service is single-threaded so we don't listen for a new
180     // connection until the prior connection drops.
181     do
182     {
183         printf("Platform server listening on port %d\n", PortNumber);
184
185         // blocking accept
186         length = sizeof(HerAddress);
187         serverSocket = accept(listenSocket,
188                               (struct sockaddr*) &HerAddress,
189                               &length);
190         if(serverSocket == SOCKET_ERROR)
191         {
192             printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
193             return -1;
194         };
195         printf("Client accepted\n");
196
197         // normal behavior on client disconnection is to wait for a new client
198         // to connect
199         continueServing = PlatformServer(serverSocket);
200         closesocket(serverSocket);
201     }
202     while(continueServing);
203
204     return 0;
205 }
```

D.3.3.4. PlatformSignalService()

This function starts a new thread waiting for platform signals. Platform signals are processed one at a time in the order in which they are received.

```
206 int
207 PlatformSignalService(
208     int             PortNumber
209     )
210 {
211     HANDLE              hPlatformSvc;
212     int                 ThreadId;
213     int                 port = PortNumber;
214
215     // Create service thread for platform signals
216     hPlatformSvc = CreateThread(NULL, 0,
217                                 (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
218                                 (LPVOID) (INT_PTR) port, 0, (LPDWORD)&ThreadId);
219     if(hPlatformSvc == NULL)
220     {
221         printf("Thread Creation failed\n");
```

```
222             return -1;
223         }
224 
225         return 0;
226     }
```

D.3.3.5. RegularCommandService()

This funciton services regular commands.

```
227     int
228     RegularCommandService(
229         int             PortNumber
230         )
231     {
232         SOCKET              listenSocket;
233         SOCKET              serverSocket;
234         struct              sockaddr_in HerAddress;
235 
236         int res, length;
237         BOOL continueServing;
238 
239         res = CreateSocket(PortNumber, &listenSocket);
240         if(res != 0)
241         {
242             printf("Create platform service socket fail\n");
243             return res;
244         }
245 
246         // Loop accepting connections one-by-one until we are killed or asked to stop
247         // Note the TPM command service is single-threaded so we don't listen for
248         // a new connection until the prior connection drops.
249         do
250         {
251             printf("TPM command server listening on port %d\n", PortNumber);
252 
253             // blocking accept
254             length = sizeof(HerAddress);
255             serverSocket = accept(listenSocket,
256                                   (struct sockaddr*) &HerAddress,
257                                   &length);
258             if(serverSocket ==SOCKET_ERROR)
259             {
260                 printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
261                 return -1;
262             };
263             printf("Client accepted\n");
264 
265             // normal behavior on client disconnection is to wait for a new client
266             // to connect
267             continueServing = TpmServer(serverSocket);
268             closesocket(serverSocket);
269         }
270         while(continueServing);
271 
272         return 0;
273     }
```

D.3.3.6. StartTcpServer()

Main entry-point to the TCP server. The server listens on port specified. Note that there is no way to specify the network interface in this implementation.

```
274 int
275 StartTcpServer(
276     int             PortNumber
277     )
278 {
279     int                 res;
280
281     // Start Platform Signal Processing Service
282     res = PlatformSignalService(PortNumber+1);
283     if (res != 0)
284     {
285         printf("PlatformSignalService failed\n");
286         return res;
287     }
288
289     // Start Regular/DRTM TPM command service
290     res = RegularCommandService(PortNumber);
291     if (res != 0)
292     {
293         printf("RegularCommandService failed\n");
294         return res;
295     }
296
297     return 0;
298 }
```

D.3.3.7. ReadBytes()

This function reads the indicated number of bytes *(NumBytes*) into buffer from the indicated socket.

```
299 BOOL
300 ReadBytes(
301     SOCKET           s,
302     char            *buffer,
303     int              NumBytes
304     )
305 {
306     int                 res;
307     int                 numGot = 0;
308
309     while(numGot<NumBytes)
310     {
311         res = recv(s, buffer+numGot, NumBytes-numGot, 0);
312         if(res == -1)
313         {
314             printf("Receive error.  Error is 0x%x\n", WSAGetLastError());
315             return FALSE;
316         }
317         if(res==0)
318         {
319             return FALSE;
320         }
321         numGot+=res;
322     }
323     return TRUE;
324 }
```

D.3.3.8. WriteBytes()

This function will send the indicated number of bytes *(NumBytes*) to the indicated socket

```
325 BOOL
326 WriteBytes(
```

```
327         SOCKET          s,
328         char            *buffer,
329         int             NumBytes
330         )
331     {
332         int                 res;
333         int                 numSent = 0;
334         while(numSent<NumBytes)
335         {
336             res = send(s, buffer+numSent, NumBytes-numSent, 0);
337             if(res == -1)
338             {
339                 if(WSAGetLastError() == 0x2745)
340                 {
341                     printf("Client disconnected\n");
342                 }
343                 else
344                 {
345                     printf("Send error.  Error is 0x%x\n", WSAGetLastError());
346                 }
347                 return FALSE;
348             }
349             numSent+=res;
350         }
351         return TRUE;
352     }
```

D.3.3.9. WriteUINT32()

Send 4 bytes containing hton(1)

```
353     BOOL
354     WriteUINT32(
355         SOCKET          s,
356         UINT32          val
357         )
358     {
359         UINT32 netVal = htonl(val);
360         return WriteBytes(s, (char*) &netVal, 4);
361     }
```

D.3.3.10. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
362     BOOL
363     ReadVarBytes(
364         SOCKET          s,
365         char            *buffer,
366         UINT32          *BytesReceived,
367         int             MaxLen
368         )
369     {
370         int                 length;
371         BOOL                res;
372
373         res = ReadBytes(s, (char*) &length, 4);
374         if(!res) return res;
375         length = ntohl(length);
376         *BytesReceived = length;
377         if(length>MaxLen)
378         {
```

```
379             printf("Buffer too big.  Client says %d\n", length);
380             return FALSE;
381         }
382         if(length==0) return TRUE;
383         res = ReadBytes(s, buffer, length);
384         if(!res) return res;
385         return TRUE;
386     }
```

D.3.3.11. WriteVarBytes()

Send a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
387     BOOL
388     WriteVarBytes(
389         SOCKET           s,
390         char            *buffer,
391         int              BytesToSend
392         )
393     {
394         UINT32               netLength = htonl(BytesToSend);
395         BOOL res;
396
397         res = WriteBytes(s, (char*) &netLength, 4);
398         if(!res) return res;
399         res = WriteBytes(s, buffer, BytesToSend);
400         if(!res) return res;
401         return TRUE;
402     }
```

D.3.3.12. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```
403     BOOL
404     TpmServer(
405         SOCKET           s
406         )
407     {
408         UINT32               length;
409         UINT32               Command;
410         BYTE                 locality;
411         BOOL                 ok;
412         int                  result;
413         int                  clientVersion;
414         _IN_BUFFER           InBuffer;
415         _OUT_BUFFER          OutBuffer;
416
417         for(;;)
418         {
419             ok = ReadBytes(s, (char*) &Command, 4);
420             // client disconnected (or other error).  We stop processing this client
421             // and return to our caller who can stop the server or listen for another
422             // connection.
423             if(!ok)
424                 return TRUE;
425             Command = ntohl(Command);
426             switch(Command)
427             {
428                 case TPM_SIGNAL_HASH_START:
429                     _rpc__Signal_Hash_Start();
430                     break;
```

```
431
432             case TPM_SIGNAL_HASH_END:
433                 _rpc__Signal_HashEnd();
434                 break;
435
436             case TPM_SIGNAL_HASH_DATA:
437                 ok = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
438                 if(!ok) return TRUE;
439                 InBuffer.Buffer = (BYTE*) InputBuffer;
440                 InBuffer.BufferSize = length;
441                 _rpc__Signal_Hash_Data(InBuffer);
442                 break;
443
444             case TPM_SEND_COMMAND:
445                 ok = ReadBytes(s, (char*) &locality, 1);
446                 if(!ok)
447                     return TRUE;
448
449                 ok = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
450                 if(!ok)
451                     return TRUE;
452                 InBuffer.Buffer = (BYTE*) InputBuffer;
453                 InBuffer.BufferSize = length;
454                 OutBuffer.BufferSize = MAX_BUFFER;
455                 OutBuffer.Buffer = (_OUTPUT_BUFFER) OutputBuffer;
456                 // record the number of bytes in the command if it is the largest
457                 // we have seen so far.
458                 if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
459                 {
460                     CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
461                     memcpy(&CommandResponseSizes.largestCommand,
462                            &InputBuffer[6], sizeof(UINT32));
463                 }
464
465                 _rpc__Send_Command(locality, InBuffer, &OutBuffer);
466                 // record the number of bytes in the response if it is the largest
467                 // we have seen so far.
468                 if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
469                 {
470                     CommandResponseSizes.largestResponseSize
471                         = OutBuffer.BufferSize;
472                     memcpy(&CommandResponseSizes.largestResponse,
473                            &OutputBuffer[6], sizeof(UINT32));
474                 }
475                 ok = WriteVarBytes(s,
476                                    (char*) OutBuffer.Buffer,
477                                    OutBuffer.BufferSize);
478                 if(!ok)
479                     return TRUE;
480                 break;
481
482             case TPM_REMOTE_HANDSHAKE:
483                 ok = ReadBytes(s, (char*)&clientVersion, 4);
484                 if(!ok)
485                     return TRUE;
486                 if( clientVersion == 0 )
487                 {
488                     printf("Unsupported client version (0).\n");
489                     return TRUE;
490                 }
491                 ok &= WriteUINT32(s, ServerVersion);
492                 ok &= WriteUINT32(s,
493                                   tpmInRawMode | tpmPlatformAvailable | tpmSupportsPP);
494                 break;
495
496             case TPM_SET_ALTERNATIVE_RESULT:
```

```
497                 ok = ReadBytes(s, (char*)&result, 4);
498                 if(!ok)
499                     return TRUE;
500                 // Alternative result is not applicable to the simulator.
501                 break;
502 
503             case TPM_SESSION_END:
504                 // Client signaled end-of-session
505                 return TRUE;
506 
507             case TPM_STOP:
508                 // Client requested the simulator to exit
509                 return FALSE;
510             default:
511                 printf("Unrecognized TPM interface command %d\n", Command);
512                 return TRUE;
513         }
514         ok = WriteUINT32(s,0);
515         if(!ok)
516             return TRUE;
517     }
518     return FALSE;
519 }
```


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

```
#define _SWAP_H         // Preclude inclusion of unnecessary simulator header
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <setjmp.h>
#include "bool.h"
#include "Platform.h"
#include "ExecCommand_fp.h"
#include "Manufacture_fp.h"
#include "DRTM_fp.h"
#include "_TPM_Init_fp.h"
#include "TpmFail_fp.h"
#include <windows.h>
#include "TpmTcpProtocol.h"
static BOOL     s_isPowerOn = FALSE;
```

D.4.3. Functions

D.4.3.1. Signal_PowerOn()

This function processes a power-on indicataion. Amoung other things, it calls the _TPM_Init() hangler.

```
void
_rpc__Signal_PowerOn(
    BOOL        isReset
    )
{
    // if power is on and this is not a call to do TPM reset then return
    if(s_isPowerOn && !isReset)
        return;

    // If this is a reset but power is not on, then return
    if(isReset && !s_isPowerOn)
        return;

    // Pass power on signal to platform
    if(isReset)
        _plat__Signal_Reset();
    else
        _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()

This function processes the power off indication. Its primary funtion is to set a flag indicating that the next power on indication should cause _TPM_Init() to be called.

```
41  void
42  _rpc__Signal_PowerOff(
43      void
44      )
45  {
46      if(!s_isPowerOn) return;
47
48      // Pass power off signal to platform
49      _plat__Signal_PowerOff();
50
51      s_isPowerOn = FALSE;
52
53      return;
54  }
```

D.4.3.3. _rpc__ForceFailureMode()

This function is used to debug the Failure Mode logic of the TPM. It will set a flag in the TPM code such that the next call to TPM2_SelfTest() will result in a failure, putting the TPM into Failure Mode.

```
55  void
56  _rpc__ForceFailureMode(
57      void
58      )
59  {
60      SetForceFailureMode();
61  }
```

D.4.3.4. _rpc__Signal_PhysicalPresenceOn()

This function is called to simulate activation of the physical presence **pin**.

```
62  void
63  _rpc__Signal_PhysicalPresenceOn(
64      void
65      )
66  {
67      // If TPM is power off, reject this signal
68      if(!s_isPowerOn) return;
69
70      // Pass physical presence on to platform
71      _plat__Signal_PhysicalPresenceOn();
72
73      return;
74  }
```

D.4.3.5. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence **pin**.

```
75  void
76  _rpc__Signal_PhysicalPresenceOff(
77      void
78      )
79  {
```

```
80      // If TPM is power off, reject this signal
81      if(!s_isPowerOn) return;
82
83      // Pass physical presence off to platform
84      _plat__Signal_PhysicalPresenceOff();
85
86      return;
87  }
```

D.4.3.6. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start() event. It will call

```
88  void
89  _rpc__Signal_Hash_Start(
90      void
91      )
92  {
93      // If TPM is power off, reject this signal
94      if(!s_isPowerOn) return;
95
96      // Pass _TPM_Hash_Start signal to TPM
97      Signal_Hash_Start();
98      return;
99  }
```

D.4.3.7. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data() event.

```
100 void
101 _rpc__Signal_Hash_Data(
102     _IN_BUFFER       input
103     )
104 {
105     // If TPM is power off, reject this signal
106     if(!s_isPowerOn) return;
107
108     // Pass _TPM_Hash_Data signal to TPM
109     Signal_Hash_Data(input.BufferSize, input.Buffer);
110     return;
111 }
```

D.4.3.8. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End() event.

```
112 void
113 _rpc__Signal_HashEnd(
114     void
115     )
116 {
117     // If TPM is power off, reject this signal
118     if(!s_isPowerOn) return;
119
120     // Pass _TPM_HashEnd signal to TPM
121     Signal_Hash_End();
122     return;
123 }
```

Command interface Entry of a RPC call

```
124  void
125  _rpc__Send_Command(
126      unsigned char    locality,
127      _IN_BUFFER       request,
128      _OUT_BUFFER     *response
129      )
130  {
131      // If TPM is power off, reject any commands.
132      if(!s_isPowerOn) {
133          response->BufferSize = 0;
134          return;
135      }
136      // Set the locality of the command so that it doesn't change during the command
137      _plat__LocalitySet(locality);
138      // Do implementation-specific command dispatch
139      ExecuteCommand(request.BufferSize, request.Buffer,
140                              &response->BufferSize, &response->Buffer);
141      return;
142
143  }
```

D.4.3.9. _rpc__Signal_CancelOn()

This function is used to turn on the indication to cancel a command in process. An executing command is not interrupted. The command code may perodically check this indication to see if it should abort the current command processing and returned TPM_RC_CANCELLED.

```
144  void
145  _rpc__Signal_CancelOn(
146      void
147      )
148  {
149      // If TPM is power off, reject this signal
150      if(!s_isPowerOn) return;
151
152      // Set the platform canceling flag.
153      _plat__SetCancel();
154
155      return;
156  }
```

D.4.3.10. _rpc__Signal_CancelOff()

This function is used to turn off the indication to cancel a command in process.

```
157  void
158  _rpc__Signal_CancelOff(
159      void
160      )
161  {
162      // If TPM is power off, reject this signal
163      if(!s_isPowerOn) return;
164
165      // Set the platform canceling flag.
166      _plat__ClearCancel();
167
168      return;
169  }
```


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.

```
170 void
171 _rpc__Signal_NvOn(
172     void
173     )
174 {
175     // If TPM is power off, reject this signal
176     if(!s_isPowerOn) return;
177 
178     _plat__SetNvAvail();
179     return;
180 }
```

D.4.3.12. _rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

```
181 void
182 _rpc__Signal_NvOff(
183     void
184     )
185 {
186     // If TPM is power off, reject this signal
187     if(!s_isPowerOn) return;
188 
189     _plat__ClearNvAvail();
190     return;
191 }
```

D.4.3.13. _rpc__Shutdown()

This function is used to stop the TPM simulator.

```
192 void
193 _rpc__Shutdown(
194     void
195     )
196 {
197     RPC_STATUS status;
198 
199     // Stop TPM
200     TPM_TearDown();
201 
202     status = RpcMgmtStopServerListening(NULL);
203     if (status != RPC_S_OK)
204     {
205         printf_s("RpcMgmtStopServerListening returned: 0x%x\n", status);
206         exit(status);
207     }
208 
209     status = RpcServerUnregisterIf(NULL, NULL, FALSE);
210     if (status != RPC_S_OK)
211     {
212         printf_s("RpcServerUnregisterIf returned 0x%x\n", status);
213         exit(status);
214     }
215 }
```

D.5 TPMCmds.c

D.5.1. Description

This file contains the entry point for the simulator.

D.5.2. Includes, Defines, Data Definitions, and Function Prototypes

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <ctype.h>
#include <windows.h>
#include <strsafe.h>
#include "string.h"
#include "TpmTcpProtocol.h"
#include "..\tpm\include\TpmBuildSwitches.h"
#include "..\tpm\include\prototypes\Manufacture_fp.h"
#define PURPOSE \
"TPM Reference Simulator.\nCopyright Microsoft 2010, 2011.\n"
#define DEFAULT_TPM_PORT 2321
void* MainPointer;
int _plat__NVEnable(void* platParameters);
void _plat__NVDisable();
int StartTcpServer(int PortNumber);
```

D.5.3. Functions

D.5.3.1. Usage()

This function prints the proper calling sequence for the simulator.

```
void
Usage(
    char                *pszProgramName
    )
{
    fprintf_s(stderr, "%s", PURPOSE);
    fprintf_s(stderr, "Usage:\n");
    fprintf_s(stderr, "%s         - Starts the TPM server listening on port %d\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()

This is the main entry point for the simulator.

main: register the interface, start listening for clients

```
void __cdecl
main(
    int              argc,
    char            *argv[]
    )
```

```
38  {
39      int portNum = DEFAULT_TPM_PORT;
40      if(argc>2)
41      {
42          Usage(argv[0]);
43      }
44
45      if(argc==2)
46      {
47          if(strcmp(argv[1], "?") ==0)
48          {
49              Usage(argv[0]);
50          }
51          portNum = atoi(argv[1]);
52          if(portNum <=0 || portNum>65535)
53          {
54              Usage(argv[0]);
55          }
56      }
57      _plat__NVEnable(NULL);
58      if(TPM_Manufacture(1) != 0)
59      {
60          exit(1);
61      }
62      // Coverage test - repeated manufacturing attempt
63      if(TPM_Manufacture(0) != 1)
64      {
65          exit(2);
66      }
67      // Coverage test - re-manufacturing
68      TPM_TearDown();
69      if(TPM_Manufacture(1) != 0)
70      {
71          exit(3);
72      }
73      // Disable NV memory
74      _plat__NVDisable();
75
76      StartTcpServer(portNum);
77      return;
78  }
```