Trusted Platform Module Library

Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 00.99

October 31, 2013

Contact: admin@trustedcomputinggroup.org

Published

Copyright © TCG 2006-2013

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

 Copyright © TCG 2006-2013

CONTENTS

DRAFT

DRAFT

DRAFT

DRAFT

DRAFT

 Copyright © TCG 2006-2013

DRAFT

 Copyright © TCG 2006-2013

 Copyright © TCG 2006-2013

DRAFT

 Copyright © TCG 2006-2013

DRAFT

 Copyright © TCG 2006-2013

DRAFT

DRAFT

DRAFT

Trusted Platform Module Library
Part 4: Supporting Routines

1 Scope

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

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

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

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

2 Terms and definitions

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

3 Symbols and abbreviated terms

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

4 Automation

Part 2 and 3 are constructed so that they can be processed by an automated parser. For example, part 2 can be processed to generate header file contents such as structures, typedefs, and enums. 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 part 2 Annexes are constructed so that they can be processed by a program. The program that processes these tables in the part 2 Annexes is called "The part 2 Configuration Parser."

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

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

 Copyright © TCG 2006-2013

4.2 Structure Parser

4.2.1 Introduction

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

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

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

The unmarshaling code produced by the 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 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 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.

 Copyright © TCG 2006-2013

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 4.2.2.2 but the data movement is from **source** to **buffer**.

4.2.3.3 Arrays

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

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

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

4.3 Command Parser

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

The part 3 Command Parser takes as input a part 3 of the TPM specification and some configuration files produced by the part 2 Configuration Parser. This parser uses the contents of the command and response tables in part 3 to produce unmarshaling code for the command and the marshaling code for the response. Additionally, this parser produces support routines that are used to check that the proper number of authorization values of the proper type have been provided. These support routines are called by the functions in this Part 4.

 Copyright © TCG 2006-2013

4.4 Portability

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

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

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

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

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

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

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

 Copyright © TCG 2006-2013

5 Header Files

5.1 Introduction

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

5.2 bool.h

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

DRAFT

5.3 Capabilities.h

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

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

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

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

5.4 TPMB.h

This file contains extra TPM2B structures

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

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

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

Macro to instance and initialize a TPM2B value

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

A 2B structure for a seed

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

A 2B hash block

```
TPM2B_TYPE(HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
TPM2B_TYPE(RSA_PRIME, MAX_RSA_KEY_BYTES/2);
TPM2B_TYPE(1_BYTE_VALUE, 1);
TPM2B_TYPE(2_BYTE_VALUE, 2);
TPM2B_TYPE(4_BYTE_VALUE, 4);
TPM2B_TYPE(20_BYTE_VALUE, 20);
TPM2B_TYPE(32_BYTE_VALUE, 32);
TPM2B_TYPE(48_BYTE_VALUE, 48);
TPM2B_TYPE(64_BYTE_VALUE, 64);
TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
#endif
```

DRAFT

5.5 TpmError.h

```
#ifndef _TPM_ERROR_H
#define _TPM_ERROR_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_DRBG_SELF_TEST          (6)
#define     FATAL_ERROR_CRYPTO                  (7)
```

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

```
int _plat__TpmFail(const char *function, int line, int code);
#ifdef EMPTY_ASSERT
    #define pAssert(a)
#else
    #define pAssert(a) (!!(a) || _plat__TpmFail(__FUNCTION__, \
                                                __LINE__,     \
                                                FATAL_ERROR_PARAMETER))
#endif
#define FAIL(a) (_plat__TpmFail(__FUNCTION__, __LINE__, a))
#endif
```

5.6 Global.h

5.6.1 Description

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

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

5.6.2 Includes

```
1   #ifndef         GLOBAL_H
2   #define         GLOBAL_H
3   #include        "TpmBuildSwitches.h"
4   #include        "Tpm.h"
5   #include        "TPMB.h"
6   #include        "CryptoEngine.h"
```

5.6.3 Defines

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

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

5.6.4 Hash State Structures

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

```
11  typedef struct
12  {
13      CPRI_HASH_STATE     state;              // hash state
14      HASH_STATE_TYPE     type;               // type of the context
15  } HASH_STATE;
```

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

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

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

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

A NAME is a BYTE array that can contain a TPMU_NAME

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

 Copyright © TCG 2006-2013

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.

```
19  typedef struct
20  {
21      HASH_STATE          hashState;          // the hash state
22      TPM2B_HASH_BLOCK    hmacKey;            // the HMAC key
23  } HMAC_STATE;
```

5.6.5 Loaded Object Structures

5.6.5.1 Description

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

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

5.6.5.2 OBJECT_ATTRIBUTES

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

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

5.6.5.3 OBJECT Structure

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

```
60  typedef struct
61  {
62      // The attributes field is required to be first followed by the publicArea.
63      // This allows the overlay of the object structure and a sequence structure
64      OBJECT_ATTRIBUTES   attributes;         // object attributes
65      TPMT_PUBLIC         publicArea;         // public area of an object
66      TPMT_SENSITIVE      sensitive;          // sensitive area of an object
67
68  #ifdef  TPM_ALG_RSA
69      TPM2B_PUBLIC_KEY_RSA privateExponent;   // Additional field for the private
70                                              // exponent of an RSA key.
71  #endif
72      TPM2B_NAME          qualifiedName;      // object qualified name
73      TPMI_DH_OBJECT      evictHandle;        // if the object is an evict object,
74                                              // the original handle is kept here.
75                                              // The 'working' handle will be the
76                                              // handle of an object slot.
77
78      TPM2B_NAME          name;               // Name of the object name. Kept here
79                                              // to avoid repeatedly computing it.
80  } OBJECT;
```

5.6.5.4 HASH_OBJECT Structure

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

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

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

5.6.5.5 ANY_OBJECT

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

```
96  typedef union
97  {
```

```
    OBJECT              entity;
    HASH_OBJECT         hash;
} ANY_OBJECT;
```

5.6.6 AUTH_DUP Types

These values are used in the authorization processing.

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

5.6.7 Active Session Context

5.6.7.1 Description

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

5.6.7.2 SESSION_ATTRIBUTES

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

```
typedef struct
{
    unsigned            isPolicy : 1;       //1) SET if the session may only
                                            //   be used for policy
    unsigned            isAudit : 1;        //2) SET if the session is used
                                            //   for audit
    unsigned            isBound : 1;        //3) SET if the session is bound to
                                            //   with an entity.
                                            //   This attribute will be CLEAR if
                                            //   either isPolicy or isAudit is SET.
    unsigned            iscpHashDefined : 1;//4) SET if the cpHash has been defined
                                            //   This attribute is not SET unless
                                            //   'isPolicy' is SET.
    unsigned            isAuthValueNeeded : 1;
                                            //5) SET if the authValue is required
                                            //   for computing the session HMAC.
                                            //   This attribute is not SET unless
                                            //   isPolicy is SET.
    unsigned            isPasswordNeeded : 1;
                                            //6) SET if a password authValue is
                                            //   required for authorization
                                            //   This attribute is not SET unless
                                            //   isPolicy is SET.
    unsigned            isPPRequired : 1;   //7) SET if physical presence is
                                            //   required to be asserted when the
                                            //   authorization is checked.
                                            //   This attribute is not SET unless
                                            //   isPolicy is SET.
    unsigned            isTrialPolicy : 1;  //8) SET if the policy session is
                                            //   created for trial of the policy's
                                            //   policyHash generation.
                                            //   This attribute is not SET unless
                                            //   isPolicy is SET.
    unsigned            isDaBound : 1;      //9) SET if the bind entity had noDA
                                            //   CLEAR. If this is SET, then an
```

```
                                            //   auth failure using this session
                                            //   will count against lockout even
                                            //   if the object being authorized is
                                            //   exempt from DA.
    unsigned            isLockoutBound : 1; //10)SET if the session is bound to
                                            //   lockoutAuth.
    unsigned            requestWasBound : 1;//11) SET if the session is being used
                                            //    with the bind entiry. 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;
```

5.6.7.3 SESSION Structure

The SESION 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
                                            // session)

    UINT64              timeOut;            // timeout relative to
                                            // TPMS_CLOCK_INFO.clock
                                            // There is no timeout if this value
                                            // is 0.
    union
    {
        TPM2B_NAME      boundEntity;         // value used to track the entity to
                                             // which the session is bound

        TPM2B_DIGEST    cpHash;              // the required cpHash value for the
                                             // command being authorized

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

    union
```

```
198         {
199             TPM2B_DIGEST    auditDigest;        // audit session digest
200             TPM2B_DIGEST    policyDigest;         // policyHash
201
202         } u2;                                   // audit log and policyHash may
203                                                 // share space to save memory
204     } SESSION;
```

5.6.8 PCR

5.6.8.1 PCR_SAVE Structure

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

```
205     typedef struct
206     {
207     #ifdef TPM_ALG_SHA1
208         BYTE                sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
209     #endif
210     #ifdef TPM_ALG_SHA256
211         BYTE                sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
212     #endif
213     #ifdef TPM_ALG_SHA384
214         BYTE                sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
215     #endif
216     #ifdef TPM_ALG_SHA512
217         BYTE                sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
218     #endif
219     #ifdef TPM_ALG_SM3_256
220         BYTE                sm3_256[NUM_STATIC_PCR][SM3_256_DIGEST_SIZE];
221     #endif
222
223         // This counter increments whenever the PCR are updated.
224         // NOTE: A platform-specific specification may designate
225         //       certain PCR changes as not causing this counter
226         //       to increment.
227         UINT32              pcrCounter;
228
229     } PCR_SAVE;
```

5.6.8.2 PCR_POLICY

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

```
230     typedef struct
231     {
232         TPMI_ALG_HASH       hashAlg[NUM_POLICY_PCR_GROUP];
233         TPM2B_DIGEST        a;
234         TPM2B_DIGEST        policy[NUM_POLICY_PCR_GROUP];
235     } PCR_POLICY;
```

5.6.8.3 PCR_AUTHVALUE

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

```
236     typedef struct
237     {
```

```
238         TPM2B_DIGEST        auth[NUM_AUTHVALUE_PCR_GROUP];
239     } PCR_AUTHVALUE;
```

5.6.9 Startup

5.6.9.1 SHUTDOWN_NONE

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

NOTE: This is a reserved value.

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

5.6.9.2 STARTUP_TYPE

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

```
241     typedef enum
242     {
243         SU_RESET,
244         SU_RESTART,
245         SU_RESUME
246     } STARTUP_TYPE;
```

5.6.10 NV

5.6.10.1 NV_RESERVE

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

```
247     typedef enum
248     {
249     // Entries below mirror the PERSISTENT_DATA structure. These values are written
250     // to NV as individual items.
251         // hierarchy
252         NV_DISABLE_CLEAR,
253         NV_OWNER_ALG,
254         NV_ENDORSEMENT_ALG,
255         NV_OWNER_POLICY,
256         NV_ENDORSEMENT_POLICY,
257         NV_OWNER_AUTH,
258         NV_ENDORSEMENT_AUTH,
259         NV_LOCKOUT_AUTH,
260
261         NV_EP_SEED,
262         NV_SP_SEED,
263         NV_PP_SEED,
264
265         NV_PH_PROOF,
266         NV_SH_PROOF,
267         NV_EH_PROOF,
268
269         // Time
270         NV_TOTAL_RESET_COUNT,
```

```
271         NV_RESET_COUNT,
272
273         // PCR
274         NV_PCR_POLICIES,
275         NV_PCR_ALLOCATED,
276
277         // Physical Presence
278         NV_PP_LIST,
279
280         // Dictionary Attack
281         NV_FAILED_TRIES,
282         NV_MAX_TRIES,
283         NV_RECOVERY_TIME,
284         NV_LOCKOUT_RECOVERY,
285         NV_LOCKOUT_AUTH_ENABLED,
286
287         // Orderly State flag
288         NV_ORDERLY,
289
290         // Command Audit
291         NV_AUDIT_COMMANDS,
292         NV_AUDIT_HASH_ALG,
293         NV_AUDIT_COUNTER,
294
295         // Algorithm Set
296         NV_ALGORITHM_SET,
297
298         NV_FIRMWARE_V1,
299         NV_FIRMWARE_V2,
300
301     // The entries above are in PERSISTENT_DATA. The entries below represent
302     // structures that are read and written as a unit.
303
304     // ORDERLY_DATA data structure written on each orderly shutdown
305         NV_ORDERLY_DATA,
306
307     // STATE_CLEAR_DATA structure written on each Shutdown(STATE)
308         NV_STATE_CLEAR,
309
310     // STATE_RESET_DATA structure written on each Shutdown(STATE)
311         NV_STATE_RESET,
312
313         NV_RESERVE_LAST             // end of NV reserved data list
314     } NV_RESERVE;
```

5.6.10.2 NV_INDEX

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

```
315     typedef struct
316     {
317         TPMS_NV_PUBLIC      publicArea;
318         TPM2B_AUTH          authValue;
319     } NV_INDEX;
```

5.6.11 COMMIT_INDEX_MASK

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

```
320     #ifdef TPM_ALG_ECC
```

```
321 #define COMMIT_INDEX_MASK ((UINT16)((sizeof(gr.commitArray)*8)-1))
322 #endif
```

5.6.12 RAM Global Values

5.6.12.1 Description

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

5.6.12.2 g_rcIndex

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

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

5.6.12.3 g_exclusiveAuditSession

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

```
324 extern TPM_HANDLE        g_exclusiveAuditSession;
```

5.6.12.4 g_time

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

```
325 extern  UINT64            g_time;
```

5.6.12.5 g_phEnable

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

```
326 extern BOOL              g_phEnable;
```

5.6.12.6 g_pceReConfig

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

```
327 extern BOOL              g_pcrReConfig;
```

5.6.12.7 g_DRTMHandle

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

```
328 extern TPMI_DH_OBJECT    g_DRTMHandle;
```

 Copyright © TCG 2006-2013

5.6.12.8 g_DrtmPreStartup

This value indicates that an H-CRTM occured after _TPM_Init() but before TPM2_Startup()

```
329    extern  BOOL            g_DrtmPreStartup;
```

5.6.12.9 g_updateNV

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

```
330    extern BOOL             g_updateNV;
```

5.6.12.10 g_clearOrderly

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

```
331    extern BOOL             g_clearOrderly;
```

5.6.12.11 g_prevOrderlyState

This location indicates how the TPM was shut down before the most recent TPM2_Startup(). This value, along with the startup type, determines if the TPM should do a TPM Reset, TPM Restart, or TPM Resume.

```
332    extern TPM_SU           g_prevOrderlyState;
```

5.6.13 Persistent Global Values

5.6.13.1 Description

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

5.6.13.2 PERSISTENT_DATA

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

```
333    typedef struct
334    {
335    //*****************************************************************************
336    //          Hierarchy
337    //*****************************************************************************
338    // The values in this section are related to the hierarchies.
339
340        BOOL                disableClear;       // TRUE if TPM2_Clear() using
341                                                // lockoutAuth is disabled
342
```

 Copyright © TCG 2006-2013

```
343         // Hierarchy authPolicies
344         TPMI_ALG_HASH       ownerAlg;
345         TPMI_ALG_HASH       endorsementAlg;
346         TPM2B_DIGEST        ownerPolicy;
347         TPM2B_DIGEST        endorsementPolicy;
348
349         // Hierarchy authValues
350         TPM2B_AUTH          ownerAuth;
351         TPM2B_AUTH          endorsementAuth;
352         TPM2B_AUTH          lockoutAuth;
353
354         // Primary Seeds
355         TPM2B_SEED          EPSeed;
356         TPM2B_SEED          SPSeed;
357         TPM2B_SEED          PPSeed;
358         // Note there is a nullSeed in the state_reset memory.
359
360         // Hierarchy proofs
361         TPM2B_AUTH          phProof;
362         TPM2B_AUTH          shProof;
363         TPM2B_AUTH          ehProof;
364         // Note there is a nullProof in the state_reset memory.
365
366     //*****************************************************************************
367     //          Reset Events
368     //*****************************************************************************
369     // A count that increments at each TPM reset and never get reset during the life
370     // time of TPM.  The value of this counter is initialized to 1 during TPM
371     // manufacture process.
372         UINT64              totalResetCount;
373
374     // This counter increments on each TPM Reset. The counter is reset by
375     // TPM2_Clear().
376         UINT32              resetCount;
377
378
379     //*****************************************************************************
380     //          PCR
381     //*****************************************************************************
382     // This structure hold the policies for those PCR that have an update policy.
383     // This implementation only supports a single group of PCR controlled by
384     // policy. If more are required, then this structure would be changed to
385     // an array.
386         PCR_POLICY          pcrPolicies;
387
388     // This structure indicates the allocation of PCR. The structure contains a
389     // list of PCR allocations for each implemented algorithm. If no PCR are
390     // allocated for an algorithm, a list entry still exists but the bit map
391     // will contain no SET bits.
392         TPML_PCR_SELECTION  pcrAllocated;
393
394     //*****************************************************************************
395     //          Physical Presence
396     //*****************************************************************************
397     // The PP_LIST type contains a bit map of the commands that require physical
398     // to be asserted when the authorization is evaluated. Physical presence will be
399     // checked if the corresponding bit in the array is SET and if the authorization
400     // handle is TPM_RH_PLATFORM.
401     //
402     // These bits may be changed with TPM2_PP_Commands().
403         BYTE                ppList[((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7)/8];
404
405     //*****************************************************************************
406     //          Dictionary attack values
407     //*****************************************************************************
408     // These values are used for dictionary attack tracking and control.
```

```
409         UINT32              failedTries;        // the current count of unexpired
410                                                 // authorization failures
411
412         UINT32              maxTries;           // number of unexpired authorization
413                                                 // failures before the TPM is in
414                                                 // lockout
415
416         UINT32              recoveryTime;       // time between authorization failures
417                                                 // before failedTries is decremented
418
419         UINT32              lockoutRecovery;    // time that must expire between
420                                                 // authorization failures associated
421                                                 // with lockoutAuth
422
423         BOOL                lockOutAuthEnabled; // TRUE if use of lockoutAuth is
424                                                 // allowed
425
426     //**********************************************************************
427     //            Orderly State
428     //**********************************************************************
429     // The orderly state for current cycle
430         TPM_SU              orderlyState;
431
432     //**********************************************************************
433     //           Command audit values.
434     //**********************************************************************
435         BYTE                auditComands[((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8];
436         TPMI_ALG_HASH       auditHashAlg;
437         UINT64              auditCounter;
438
439     //**********************************************************************
440     //           Algorithm selection
441     //**********************************************************************
442     //
443     // The 'algorithmSet' value indicates the collection of algorithms that are
444     // currently in used on the TPM.  The interpretation of value is vendor dependent.
445         UINT32              algorithmSet;
446
447     //**********************************************************************
448     //           Firmware version
449     //**********************************************************************
450     // The firmwareV1 and firmwareV2 values are instanced in TimeStamp.c. This is
451     // a scheme used in development to allow determination of the linker build time
452     // of the TPM. An actual implementation would implement these values in a way that
453     // is consistent with vendor needs. The values are maintained in RAM for simplified
454     // access with a master version in NV.  These values are modified in a
455     // vendor-specific way.
456
457     // g_firmwareV1 contains the more significant 32-bits of the vendor version number.
458     // In the reference implementation, if this value is printed as a hex
459     // value, it will have the format of yyyymmdd
460         UINT32              firmwareV1;
461
462     // g_firmwareV1 contains the less significant 32-bits of the vendor version number.
463     // In the reference implementation, if this value is printed as a hex
464     // value, it will have the format of 00 hh mm ss
465         UINT32              firmwareV2;
466
467     } PERSISTENT_DATA;
468     extern PERSISTENT_DATA  gp;
```

5.6.13.3 ORDERLY_DATA

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

```
469  typedef struct orderly_data
470  {
471
472  //*****************************************************************************
473  //           TIME
474  //*****************************************************************************
475
476  // Clock has two parts. One is the state save part and one is the NV part. The
477  // state save version is updated on each command. When the clock rolls over, the
478  // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
479  // orderly way, then the sClock value is used to initialize the clock. If the
480  // TPM shutdown was not orderly, then the persistent value is used and the safe
481  // attribute is clear.
482
483      UINT64              clock;              // The orderly version of clock
484      TPMI_YES_NO         clockSafe;          // Indicates if the clock value is
485                                              // safe.
486  //*****************************************************************************
487  //          DRBG
488  //*****************************************************************************
489  #ifdef _DRBG_STATE_SAVE
490      // This is DRBG state data. This is saved each time the value of clock is
491      // updated.
492      DRBG_STATE          drbgState;
493  #endif
494
495  } ORDERLY_DATA;
496  extern ORDERLY_DATA     go;
```

5.6.13.4 STATE_CLEAR_DATA

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

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

```
497  typedef struct state_clear_data
498  {
499  //*****************************************************************************
500  //           Hierarchy Control
501  //*****************************************************************************
502      BOOL                shEnable;           // default reset is SET
503      BOOL                ehEnable;           // default reset is SET
504      BOOL                phEnableNV;         // default reset is SET
505      TPMI_ALG_HASH       platformAlg;        // default reset is TPM_ALG_NULL
506      TPM2B_DIGEST        platformPolicy;     // default reset is an Empty Buffer
507      TPM2B_AUTH          platformAuth;       // default reset is an Empty Buffer
508
509  //*****************************************************************************
510  //           PCR
511  //*****************************************************************************
512  // The set of PCR to be saved on Shutdown(STATE)
513      PCR_SAVE            pcrSave;            // default reset is 0...0
514
515  // This structure hold the authorization values for those PCR that have an
516  // update authorization.
517  // This implementation only supports a single group of PCR controlled by
518  // authorization. If more are required, then this structure would be changed to
519  // an array.
520      PCR_AUTHVALUE       pcrAuthValues;
521
522  } STATE_CLEAR_DATA;
```

```
523  extern STATE_CLEAR_DATA gc;
```

5.6.13.5 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.

```
524  typedef struct state_reset_data
525  {
526  //*****************************************************************************
527  //          Hierarchy Control
528  //*****************************************************************************
529      TPM2B_AUTH          nullProof;          // The proof value associated with
530                                              // the TPM_RH_NULL hierarchy. The
531                                              // default reset value is from the RNG.
532
533      TPM2B_SEED          nullSeed;           // The seed value for the TPM_RN_NULL
534                                              // hierarchy. The default reset value
535                                              // is from the RNG.
536
537  //*****************************************************************************
538  //           Context
539  //*****************************************************************************
540  // The 'clearCount' counter is incremented each time the TPM successfully executes
541  // a TPM Resume. The counter is included in each saved context that has 'stClear'
542  // SET (including descendants of keys that have 'stClear' SET). This prevents these
543  // objects from being loaded after a TPM Resume.
544  // If 'clearCount' at its maximum value when the TPM receives a Shutdown(STATE),
545  // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
546      UINT32              clearCount;         // The default reset value is 0.
547
548      UINT64              objectContextID;    // This is the context ID for a saved
549                                              //  object context. The default reset
550                                              //  value is 0.
551
552      CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];
553                                              // This is the value from which the
554                                              // 'contextID' is derived. The
555                                              // default reset value is {0}.
556
557
558      CONTEXT_COUNTER     contextCounter;     // This array contains contains the
559                                              // values used to track the version
560                                              // numbers of saved contexts (see
561                                              // Session.c in for details). The
562                                              // default reset value is 0.
563
564  //*****************************************************************************
565  //           Command Audit
566  //*****************************************************************************
567  // When an audited command completes, ExecuteCommand() checks the return
568  // value.  If it is TPM_RC_SUCCESS, and the command is an audited command, the
569  // TPM will extend the cpHash and rpHash for the command to this value. If this
570  // digest was the Zero Digest before the cpHash was extended, the audit counter
571  // is incremented.
572
573      TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest
574                                              // by TPM2_GetCommandAuditDigest() or a
575                                              // TPM Reset.
576
577  //*****************************************************************************
578  //           Boot counter
579  //*****************************************************************************
```

```
580
581     UINT32              restartCount;       // This counter counts TPM Restarts.
582                                             // The default reset value is 0.
583
584 //*****************************************************************************
585 //            PCR
586 //*****************************************************************************
587 // This counter increments whenever the PCR are updated. This counter is preserved
588 // across TPM Resume even though the PCR are not preserved. This is because
589 // sessions remain active across TPM Restart and the count value in the session
590 // is compared to this counter so this counter must have values that are unique
591 // as long as the sessions are active.
592 // NOTE: A platform-specific specification may designate that certain PCR changes
593 //       do not increment this counter to increment.
594     UINT32              pcrCounter;         // The default reset value is 0.
595
596 #ifdef TPM_ALG_ECC
597
598 //*****************************************************************************
599 //         ECDAA
600 //*****************************************************************************
601     UINT64              commitCounter;      // This counter increments each time
602                                             // TPM2_Commit() returns
603                                             // TPM_RC_SUCCESS. The default reset
604                                             // value is 0.
605
606
607     TPM2B_NONCE         commitNonce;        // This random value is used to compute
608                                             // the commit values. The default reset
609                                             // value is from the RNG.
610
611 // This implementation relies on the number of bits in g_commitArray being a
612 // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
613     BYTE                 commitArray[16];   // The default reset value is {0}.
614
615 #endif //TPM_ALG_ECC
616
617 } STATE_RESET_DATA;
618 extern STATE_RESET_DATA gr;
```

5.6.14 Global Macro Definitions

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

```
619 #define RcSafeAddToResult(r, v) \
620     ((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 paramter might not otherwise be referenced. This macro **uses** the parameter from the perspective of the compiler so it doesn't complain.

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

5.6.15 Private data

```
622 #if defined SESSION_PROCESS_C || defined GLOBAL_C
```

From SessionProcess.c

 Copyright © TCG 2006-2013

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

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

Array of authorization session attributes

```
624   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

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

Array of nonces provided by the caller for the corresponding sessions

```
626   TPM2B_NONCE      s_nonceCaller[MAX_SESSION_NUM];
```

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

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

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter

```
629   extern UINT32           s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
630   extern UINT32           s_decryptSessionIndex;
```

Index of a session used for audit

```
631   extern UINT32           s_auditSessionIndex;
```

The *cpHash* for an audit session

```
632   extern TPM2B_DIGEST     s_cpHashForAudit;
```

The *cpHash* for command audit

```
633   #ifdef  TPM_CC_GetCommandAuditDigest
634   extern TPM2B_DIGEST    s_cpHashForCommandAudit;
635   #endif
```

Number of authorization sessions present in the command

```
636   extern UINT32           s_sessionNum;
```

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

```
637   extern BOOL             s_DAPendingOnNV;
```

```
638  #endif // SESSION_PROCESS_C
639  #if defined DA_C || defined GLOBAL_C
```

From DA.c

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

```
640  extern UINT64        s_selfHealTimer;
```

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

```
641  UINT64        s_lockoutTimer;
642  #endif // DA_C
643  #if defined NV_C || defined GLOBAL_C
```

From NV.c

List of pre-defined address of reserved data

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

List of pre-defined reserved data size in byte

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

Size of data in RAM index buffer

```
646  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

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

Address of size of RAM index space in NV

```
648  extern UINT32    s_ramIndexSizeAddr;
```

Address of NV copy of RAM index space

```
649  extern UINT32    s_ramIndexAddr;
```

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

```
650  extern UINT32    s_maxCountAddr;
```

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

```
651  extern UINT32    s_evictNvStart;
```

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

```
652  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

 Copyright © TCG 2006-2013

DRAFT

```
653  extern TPM_RC   s_NvIsAvailable;
654  #endif
655  #if defined OBJECT_C || defined GLOBAL_C
```

From Object.c

This type is the container for an object.

```
656  typedef struct
657  {
658      BOOL        occupied;
659      ANY_OBJECT      object;
660  } OBJECT_SLOT;
```

This is the memory that holds the loaded objects.

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

From PCR.c

```
664  typedef struct
665  {
666  #ifdef TPM_ALG_SHA1
667      // SHA1 PCR
668      BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
669  #endif
670  #ifdef TPM_ALG_SHA256
671      // SHA256 PCR
672      BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
673  #endif
674  #ifdef TPM_ALG_SHA384
675      // SHA384 PCR
676      BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
677  #endif
678  #ifdef TPM_ALG_SHA512
679      // SHA512 PCR
680      BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
681  #endif
682  #ifdef TPM_ALG_SM3_256
683      // SHA256 PCR
684      BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
685  #endif
686  } PCR;
687  typedef struct
688  {
689      unsigned int    stateSave : 1;              // if the PCR value should be
690                                                  // saved in state save
691      unsigned int    resetLocality : 5;          // The locality that the PCR
692                                                  // can be reset
693      unsigned int    extendLocality : 5;         // The locality that the PCR
694                                                  // can be extend
695  } PCR_Attributes;
696  extern PCR          s_pcrs[IMPLEMENTATION_PCR];
697  #endif // PCR_C
698  #if defined SESSION_C || defined GLOBAL_C
```

From Session.c

Container for HMAC or policy session tracking information

```
699  typedef struct
700  {
```

```
701         BOOL                occupied;
702         SESSION             session;        // session structure
703     } SESSION_SLOT;
704     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.

```
705     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)

```
706     extern int              s_freeSessionSlots;
707     #endif // SESSION_C
708     #if defined MANUFACTURE_C || defined GLOBAL_C
```

From Manufacture.c

```
709     extern BOOL             s_manufactured;
710     #endif // MANUFACTURE_C
711     #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.

```
712     extern BOOL             s_initialized;
713     #endif // POWER_C
714     #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.

```
715     extern UINT32   s_actionInputBuffer[1024];          // action input buffer
716     extern UINT32   s_actionOutputBuffer[1024];         // action output buffer
717     extern BYTE     s_responseBuffer[MAX_RESPONSE_SIZE];// response buffer
718     #endif // MEMORY_LIB_C
719     #endif // GLOBAL_H
```

5.7 swap.h

```
  1     #ifndef _SWAP_H
  2     #define _SWAP_H
  3     #include <Implementation.h>
  4     #if    NO_AUTO_ALIGN == YES || LITTLE_ENDIAN_TPM == YES
```

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

```
  5     #define BYTE_ARRAY_TO_UINT8(b)    (UINT8)((b)[0])
  6     #define BYTE_ARRAY_TO_UINT16(b)   (UINT16)(  ((b)[0] <<  8) \
  7                                              +  (b)[1])
  8     #define BYTE_ARRAY_TO_UINT32(b)   (UINT32)(  ((b)[0] << 24) \
  9                                              + ((b)[1] << 16) \
 10                                              + ((b)[2] << 8 ) \
```

```
11                                          +  (b)[3])
12  #define BYTE_ARRAY_TO_UINT64(b)  (UINT64)(  ((UINT64)(b)[0] << 56) \
13                                          + ((UINT64)(b)[1] << 48) \
14                                          + ((UINT64)(b)[2] << 40) \
15                                          + ((UINT64)(b)[3] << 32) \
16                                          + ((UINT64)(b)[4] << 24) \
17                                          + ((UINT64)(b)[5] << 16) \
18                                          + ((UINT64)(b)[6] <<  8) \
19                                          +  (UINT64)(b)[7])
```

Disaggregate a UINT into a byte array

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

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

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

Disaggregate a UINT into a byte array

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

5.8 InternalRoutines.h

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

Error Reporting

```
3   #include "TpmError.h"
```

NULL definition

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

UNUSED_PARAMETER

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

Internal data definition

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

DRTM functions

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

Internal subsystem functions

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

Internal support functions

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

Internal crypto functions

DRAFT

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

5.9 TpmBuildSwitches.h

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

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

This switch enables the RNG state save and restore

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

Set the alignment size for the crypto. It woudl be nice to set this according to macros automatically defined by the build environment, but that doesn't seem possible becaysas 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.

```
6   #ifndef CRYPTO_ALIGNMENT
7   #   define CRYPTO_ALIGNMENT    4
8   #endif
```

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

```
9   #define CRYPTO_ALIGNED  __declspec(align(CRYPTO_ALIGNMENT))
```

For ISO 9899:2011

```
10  //#define CRYPTO_ALIGNED    _Alignas(uint32_t)
```

For other environments where the alignement doesn't need to be forced, just null the macro.

```
11  //#define CRYPTO_ALIGNMENT sizeof(uint32_t)
12  //#define CRYPTO_ALIGNED
```

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

```
13  #ifdef SIMULATION
14  #   define RSA_KEY_CACHE
15  #   define TPM_RNG_FOR_DEBUG
16  #else
17  #   undef RSA_KEY_CACHE
18  #   undef TPM_RNG_FOR_DEBUG
19  #endif  // SIMULATION
20  #endif // _TPM_BUILD_SWITCHES_H
```

 Copyright © TCG 2006-2013

5.10 VendorString.h

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6 Main

6.1 CommandDispatcher()

In the reference implementation, a program that uses 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      command_code,       // IN: Command code
7        INT32       *parm_buffer_size,  // IN: size of parameter buffer
8        BYTE        *parm_buffer_start, // IN: pointer to start of parameter buffer
9        TPM_HANDLE  handles[],          // IN: handle array
10       UINT32      *res_handle_size,   // OUT: size of handle buffer in response
11       UINT32      *res_parm_size      // OUT: size of parameter buffer in response
12       );
13   #endif
```

ExecCommand.c

6.1.1 Introduction

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

6.1.2 Includes

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

6.1.3 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

 Copyright © TCG 2006-2013

DRAFT

3) when necessary, decrypt a parameter.

e) Calls CommandDispatcher() to:

1) unmarshal the command parameters from the command buffer;

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

3) marshal the responses into the response buffer.

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

g) Calls BuildResponseSession() to:

1) when necessary, encrypt a parameter

2) build the response authorization sessions

3) update the audit sessions and nonces

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

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

```
51                                          // for marshaling or unmarshaling
52
53      UINT32              i;                  // local temp
54
55      // Assume that everything is going to work.
56      result = TPM_RC_SUCCESS;
57
58      // Set flags for NV access state. This should happen before any other
59      // operation that may require a NV write.
60      g_updateNV = FALSE;
61      g_clearOrderly = FALSE;
62
63      // Query platform to get the NV state.  The result state is saved internally
64      // and will be reported by NvIsAvailable(). The reference code requires that
65      // accessibility of NV does not change during the execution of a command.
66      // Specifically, if NV is available when the command execution starts and then
67      // is not available later when it is necessary to write to NV, then the TPM
68      // will go into failure mode.
69      NvCheckState();
70
71      // Due to the limitations of the simulation, TPM clock must be explicitly
72      // synchronized with the system clock whenever a command is received.
73      // This function call is not necessary in a hardware TPM. However, taking
74      // a snapshot of the hardware timer at the beginning of the command allows
75      // the time value to be consistent for the duration of the command execution.
76      TimeUpdateToCurrent();
77
78      // Any command through this function will unceremoniously end the
79      // _TPM_Hash_Data/_TPM_Hash_End sequence.
80      if(g_DRTMHandle != TPM_RH_UNASSIGNED)
81          ObjectTerminateEvent();
82
83      // Get command buffer size and command buffer.
84      size = requestSize;
85      buffer = request;
86
87      // Parse command header: tag, commandSize and commandCode.
88      // First parse the tag. The unmarshaling routine will validate
89      // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
90      result = TPMI_ST_COMMAND_TAG_Unmarshal(&tag, &buffer, &size);
91      if(result != TPM_RC_SUCCESS)
92          goto Cleanup;
93
94      // Unmarshal the commandSize indicator.
95      result = UINT32_Unmarshal(&commandSize, &buffer, &size);
96      if(result != TPM_RC_SUCCESS)
97          goto Cleanup;
98
99      // On a TPM that receives bytes on a port, the number of bytes that were
100     // received on that port is requestSize it must be identical to commandSize.
101     // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
102     // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
103     // as the input processing (the function that receives the command bytes and
104     // places them in the input buffer) would likely have the input truncated when
105     // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
106     if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
107     {
108         result = TPM_RC_COMMAND_SIZE;
109         goto Cleanup;
110     }
111
112     // Unmarshal the command code.
113     result = TPM_CC_Unmarshal(&commandCode, &buffer, &size);
114     if(result != TPM_RC_SUCCESS)
115         goto Cleanup;
116
```

```
117         // Check to see if the command is implemented.
118         if(!CommandIsImplemented(commandCode))
119         {
120             result = TPM_RC_COMMAND_CODE;
121             goto Cleanup;
122         }
123 
124     #if  FIELD_UPGRADE_IMPLEMENTED  == YES
125         // If the TPM is in FUM, then the only allowed command is
126         // TPM_CC_FieldUpgradeData.
127         if(IsFieldUgradeMode() && (commandCode != TPM_CC_FieldUpgradeData))
128         {
129             result = TPM_RC_UPGRADE;
130             goto Cleanup;
131         }
132         else
133     #endif
134             // Excepting FUM, the TPM only accepts TPM2_Startup() after
135             // _TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
136             // is no longer allowed.
137             if((   !TPMIsStarted() && commandCode != TPM_CC_Startup)
138                 || (TPMIsStarted() && commandCode == TPM_CC_Startup))
139             {
140                 result = TPM_RC_INITIALIZE;
141                 goto Cleanup;
142             }
143 
144         // Start regular command process.
145         // Parse Handle buffer.
146         result = ParseHandleBuffer(commandCode, &buffer, &size, handles, &handleNum);
147         if(result != TPM_RC_SUCCESS)
148             goto Cleanup;
149 
150         // Number of handles retrieved from handle area should be less than
151         // MAX_HANDLE_NUM.
152         pAssert(handleNum <= MAX_HANDLE_NUM);
153 
154         // All handles in the handle area are required to reference TPM-resident
155         // entities.
156         for(i = 0; i < handleNum; i++)
157         {
158             result = EntityGetLoadStatus(&handles[i], commandCode);
159             if(result != TPM_RC_SUCCESS)
160             {
161                 if(result == TPM_RC_REFERENCE_H0)
162                     result = result + i;
163                 else
164                     result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
165                 goto Cleanup;
166             }
167         }
168 
169         // Authorization session handling for the command.
170         if(tag == TPM_ST_SESSIONS)
171         {
172             BYTE        *sessionBufferStart;// address of the session area first byte
173                                             // in the input buffer
174 
175             UINT32      authorizationSize;  // number of bytes in the session area
176 
177             // Find out session buffer size.
178             result = UINT32_Unmarshal(&authorizationSize, &buffer, &size);
179             if(result != TPM_RC_SUCCESS)
180                 goto Cleanup;
181 
182             // Perform sanity check on the unmarshaled value. If it is smaller than
```

```
183             // the smallest possible session or larger than the remaining size of
184             // the command, then it is an error. NOTE: This check could pass but the
185             // session size could still be wrong. That will be determined after the
186             // sessions are unmarshaled.
187             if(   authorizationSize < 9
188                || authorizationSize > (UINT32) size)
189             {
190                 result = TPM_RC_SIZE;
191                 goto Cleanup;
192             }
193
194             // The sessions, if any, follows authorizationSize.
195             sessionBufferStart = buffer;
196
197             // The parameters follow the session area.
198             parmBufferStart = sessionBufferStart + authorizationSize;
199
200             // Any data left over after removing the authorization sessions is
201             // parameter data. If the command does not have parameters, then an
202             // error will be returned if the remaining size is not zero. This is
203             // checked later.
204             parmBufferSize = size - authorizationSize;
205
206             // The actions of ParseSessionBuffer() are described in the introduction.
207             result = ParseSessionBuffer(commandCode,
208                                         handleNum,
209                                         handles,
210                                         sessionBufferStart,
211                                         authorizationSize,
212                                         parmBufferStart,
213                                         parmBufferSize);
214             if(result != TPM_RC_SUCCESS)
215                 goto Cleanup;
216         }
217         else
218         {
219             // Whatever remains in the input buffer is used for the parameters of the
220             // command.
221             parmBufferStart = buffer;
222             parmBufferSize = size;
223
224             // The command has no authorization sessions.
225             // If the command requires authorizations, then CheckAuthNoSession() will
226             // return an error.
227             result = CheckAuthNoSession(commandCode, handleNum, handles,
228                                         parmBufferStart, parmBufferSize);
229             if(result != TPM_RC_SUCCESS)
230                 goto Cleanup;
231         }
232
233         // CommandDispatcher returns a response handle buffer and a response parameter
234         // buffer if it succeeds. It will also set the parameterSize field in the
235         // buffer if the tag is TPM_RC_SESSIONS.
236         result = CommandDispatcher(tag,
237                                    commandCode,
238                                    (INT32 *) &parmBufferSize,
239                                    parmBufferStart,
240                                    handles,
241                                    &resHandleSize,
242                                    &resParmSize);
243         if(result != TPM_RC_SUCCESS)
244             goto Cleanup;
245
246         // Build the session area at the end of the parameter area.
247         BuildResponseSession(tag,
248                              commandCode,
```

```
                          resHandleSize,
                          resParmSize,
                          &resAuthSize);

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

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

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

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

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

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

    *response = MemoryGetResponseBuffer(commandCode);

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

    return;
}
```

6.2 ParseHandleBuffer()

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

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

DRAFT

6.3 SessionProcess.c

6.3.1 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.

6.3.2 Includes and Data Definitions

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

6.3.3 Authorization Support Functions

6.3.3.1 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;
            NvGetIndexInfo(handle, &nvIndex);
            result = (nvIndex.publicArea.attributes.TPMA_NV_NO_DA == SET);
            break;
        }
        case TPM_HT_PCR:
            // PCRs are always exempted from DA.
            result = TRUE;
            break;
        default:
            break;
    }
    return result;
}
```

6.3.3.2 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

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


    // Don't increment lockout unless the handle associated with the session
    // is DA protected or the session is bound to a DA protected entity.
    if(sessionHandle == TPM_RS_PW)
    {
        if(IsDAExempted(handle))
            return TPM_RC_BAD_AUTH;

    }
    else
    {
        session = SessionGet(sessionHandle);
        // If the session is bound to lockout, then use that as the relevant
        // handle. This means that an auth failure with a bound session
        // bound to lockoutAuth will take precedence over any other
        // lockout check
        if(session->attributes.isLockoutBound == SET)
            handle = TPM_RH_LOCKOUT;

        if(   session->attributes.isDaBound == CLEAR
           && IsDAExempted(handle)
          )
            // If the handle was changed to TPM_RH_LOCKOUT, this will not return
            // TPM_RC_BAD_AUTH
            return TPM_RC_BAD_AUTH;

    }
```

```
82
83      if(handle == TPM_RH_LOCKOUT)
84      {
85          pAssert(gp.lockOutAuthEnabled);
86          gp.lockOutAuthEnabled = FALSE;
87          // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
88          // the lockout auth will be reset at startup.
89          if(gp.lockoutRecovery != 0)
90          {
91              result = NvIsAvailable();
92              if(result != TPM_RC_SUCCESS)
93              {
94                  // No NV access for now. Put the TPM in pending mode.
95                  s_DAPendingOnNV = TRUE;
96              }
97              else
98              {
99                  // Update NV.
100                 NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
101                 g_updateNV = TRUE;
102             }
103         }
104     }
105     else
106     {
107         if(gp.recoveryTime != 0)
108         {
109             gp.failedTries++;
110             result = NvIsAvailable();
111             if(result != TPM_RC_SUCCESS)
112             {
113                 // No NV access for now.  Put the TPM in pending mode.
114                 s_DAPendingOnNV = TRUE;
115             }
116             else
117             {
118                 // Record changes to NV.
119                 NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
120                 g_updateNV = TRUE;
121             }
122         }
123     }
124
125     // Register a DA failure and reset the timers.
126     DARegisterFailure(handle);
127
128     return TPM_RC_AUTH_FAIL;
129 }
```

6.3.3.3 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

```
130 static BOOL
131 IsSessionBindEntity(
```

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

6.3.3.4 IsWriteOperation()

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

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

```
150     static BOOL
151     IsWriteOperation(
152         TPM_CC command_code
153     )
154     {
155         switch(command_code)
156         {
157             case TPM_CC_NV_Write:
158             case TPM_CC_NV_Increment:
159             case TPM_CC_NV_SetBits:
160             case TPM_CC_NV_Extend:
161                 return TRUE;
162             default:
163                 return FALSE;
164         }
165     }
```

6.3.3.5 IsPolicySessionRequired()

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

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 belongs to a policy group, and has its policy initialized

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

```
166  static BOOL
167  IsPolicySessionRequired(
168      TPM_CC      commandCode,        // IN: command code
169      UINT32      sessionIndex        // IN: session index
170  )
171  {
172      AUTH_ROLE       role = CommandAuthRole(commandCode, sessionIndex);
173      TPM_HT          type = HandleGetType(s_associatedHandles[sessionIndex]);
174
175      if(role == AUTH_DUP)
176          return TRUE;
177
178      if(role == AUTH_ADMIN)
179      {
180          if(type == TPM_HT_TRANSIENT)
181          {
182              OBJECT      *object = ObjectGet(s_associatedHandles[sessionIndex]);
183
184              if(object->publicArea.objectAttributes.adminWithPolicy == CLEAR)
185                  return FALSE;
186          }
187          return TRUE;
188      }
189
190      if(type == TPM_HT_PCR)
191      {
192          if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
193          {
194              TPM2B_DIGEST        policy;
195              TPMI_ALG_HASH       policyAlg;
196              policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
197                                           &policy);
198              if(policyAlg != TPM_ALG_NULL)
199                  return TRUE;
200          }
201      }
202      return FALSE;
203  }
```

6.3.3.6 IsAuthValueAvailable()

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

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

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

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

```
204  static BOOL
```

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

 Copyright © TCG 2006-2013

```
271             {
272                 NV_INDEX        nvIndex;
273                 NvGetIndexInfo(handle, &nvIndex);
274                 if(IsWriteOperation(commandCode))
275                 {
276                     if (nvIndex.publicArea.attributes.TPMA_NV_AUTHWRITE == SET)
277                         result = TRUE;
278 
279                 }
280                 else
281                 {
282                     if (nvIndex.publicArea.attributes.TPMA_NV_AUTHREAD == SET)
283                         result = TRUE;
284                 }
285             }
286             break;
287         case TPM_HT_PCR:
288             // PCR handle.
289             // authValue is always allowed for PCR
290             result =  TRUE;
291             break;
292         default:
293             // Otherwise, authValue is not available
294             break;
295     }
296     return result;
297 }
```

6.3.3.7 IsAuthPolicyAvailable()

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

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

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

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

```
321
322                 case TPM_RH_PLATFORM:
323                     if (gc.platformPolicy.t.size != 0)
324                         result = TRUE;
325                     break;
326                 default:
327                     break;
328             }
329             break;
330         case TPM_HT_TRANSIENT:
331             {
332                 // Object handle.
333                 // An evict object would already have been loaded and given a
334                 // transient object handle by this point.
335                 OBJECT  *object = ObjectGet(handle);
336                 // Policy authorization is not available for an object with only
337                 // public portion loaded.
338                 if(object->attributes.publicOnly == CLEAR)
339                 {
340                     // Policy authorization is always available for an object but
341                     // is never available for a sequence.
342                     if(!ObjectIsSequence(object))
343                         result = TRUE;
344                 }
345                 break;
346             }
347         case TPM_HT_NV_INDEX:
348             // An NV Index.
349             {
350                 NV_INDEX         nvIndex;
351                 NvGetIndexInfo(handle, &nvIndex);
352                 // If the policy size is not zero, check if policy can be used.
353                 if(nvIndex.publicArea.authPolicy.t.size != 0)
354                 {
355                     // If policy session is required for this handle, always
356                     // uses policy regardless of the attributes bit setting
357                     if(IsPolicySessionRequired(commandCode, sessionIndex))
358                         result = TRUE;
359                     // Otherwise, the presence of the policy depends on the NV
360                     // attributes.
361                     else if(IsWriteOperation(commandCode))
362                     {
363                         if (   nvIndex.publicArea.attributes.TPMA_NV_POLICYWRITE
364                             == SET)
365                             result = TRUE;
366                     }
367                     else
368                     {
369                         if (    nvIndex.publicArea.attributes.TPMA_NV_POLICYREAD
370                             ==  SET)
371                             result = TRUE;
372                     }
373                 }
374             }
375             break;
376         case TPM_HT_PCR:
377             // PCR handle.
378             if(PCRPolicyIsAvailable(handle))
379                 result = TRUE;
380             break;
381         default:
382             break;
383     }
384     return result;
385 }
```

DRAFT

6.3.4 Session Parsing Functions

6.3.4.1 ComputeCpHash()

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

```
386 static void
387 ComputeCpHash(
388     TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
389     TPM_CC           commandCode,       // IN: command code
390     UINT32           handleNum,         // IN: number of handles
391     TPM_HANDLE       handles[],         // IN: array of handles
392     UINT32           parmBufferSize,    // IN: size of input parameter area
393     BYTE            *parmBuffer,        // IN: input parameter area
394     TPM2B_DIGEST    *cpHash,            // OUT: cpHash
395     TPM2B_DIGEST    *nameHash           // OUT: name hash of command
396 )
397 {
398     UINT32           i;
399     HASH_STATE       hashState;
400     TPM2B_NAME       name;
401
402     // cpHash = hash(commandCode [ || authName1
403     //                           [ || authName2
404     //                           [ || authName 3 ]]]
405     //                           [ || parameters])
406     // A cpHash can contain just a commandCode only if the lone session is
407     // an audit session.
408
409     // Start cpHash.
410     cpHash->t.size = CryptStartHash(hashAlg, &hashState);
411
412     //  Add commandCode.
413     CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
414
415     //  Add authNames for each of the handles.
416     for(i = 0; i < handleNum; i++)
417     {
418         name.t.size = EntityGetName(handles[i], &name.t.name);
419         CryptUpdateDigest2B(&hashState, &name.b);
420     }
421
422     //  Add the parameters.
423     CryptUpdateDigest(&hashState, parmBufferSize, parmBuffer);
424
425     //  Complete the hash.
426     CryptCompleteHash2B(&hashState, &cpHash->b);
427
428     // If the nameHash is needed, compute it here.
429     if(nameHash != NULL)
430     {
431         // Start name hash. hashState may be reused.
432         nameHash->t.size = CryptStartHash(hashAlg, &hashState);
433
434         //  Adding names.
435         for(i = 0; i < handleNum; i++)
436         {
437             name.t.size = EntityGetName(handles[i], &name.t.name);
438             CryptUpdateDigest2B(&hashState, &name.b);
439         }
440         //  Complete hash.
441         CryptCompleteHash2B(&hashState, &nameHash->b);
442     }
443     return;
```

```
444 }
```

6.3.4.2 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

```
445 static TPM_RC
446 CheckPWAuthSession(
447     UINT32          sessionIndex       // IN: index of session to be processed
448 )
449 {
450     TPM2B_AUTH      authValue;
451     TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
452 
453     // Strip trailing zeros from the password.
454     MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
455 
456     // Get the auth value and size.
457     authValue.t.size = EntityGetAuthValue(associatedHandle, &authValue.t.buffer);
458 
459     // Success if the digests are identical.
460     if(Memory2BEqual(&s_inputAuthValues[sessionIndex].b, &authValue.b))
461     {
462         return TPM_RC_SUCCESS;
463     }
464     else                    // if the digests are not identical
465     {
466         // Invoke DA protection if applicable.
467         return IncrementLockout(sessionIndex);
468     }
469 }
```

6.3.4.3 ComputeCommandHMAC()

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

```
470 static void
471 ComputeCommandHMAC(
472     UINT32           sessionIndex,      // IN: index of session to be processed
473     TPM2B_DIGEST    *cpHash,            // IN: cpHash
474     TPM2B_DIGEST    *hmac               // OUT: authorization HMAC
475 )
476 {
477     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
478     TPM2B_KEY        key;
479     BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
480     BYTE            *buffer;
481     UINT32           marshalSize;
482     HMAC_STATE       hmacState;
483     TPM2B_NONCE     *nonceDecrypt;
484     TPM2B_NONCE     *nonceEncrypt;
485     SESSION         *session;
486 
487     nonceDecrypt = NULL;
488     nonceEncrypt = NULL;
```

```
489
490     // Determine if extra nonceTPM values are going to be required.
491     // If this is the first session (sessionIndex = 0) and it is an authorization
492     // session that uses an HMAC, then check if additional session nonces are to be
493     // included.
494     if(   sessionIndex == 0
495        && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
496     {
497         // If there is a decrypt session and if this is not the decrypt session,
498         // then an extra nonce may be needed.
499         if(   s_decryptSessionIndex != UNDEFINED_INDEX
500            && s_decryptSessionIndex != sessionIndex)
501         {
502             // Will add the nonce for the decrypt session.
503             session = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
504             nonceDecrypt = &session->nonceTPM;
505         }
506         // Now repeat for the encrypt session.
507         if(   s_encryptSessionIndex != UNDEFINED_INDEX
508            && s_encryptSessionIndex != sessionIndex
509            && s_encryptSessionIndex != s_decryptSessionIndex)
510         {
511             // Have to have the nonce for the encrypt session.
512             session = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
513             nonceEncrypt = &session->nonceTPM;
514         }
515     }
516
517     // Continue with the HMAC processing.
518     session = SessionGet(s_sessionHandles[sessionIndex]);
519
520     // Generate HMAC key.
521     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
522
523     // Check if the session has an associated handle and if the associated entity
524     // is the one to which the session is bound. If not, add the authValue of
525     // this entity to the HMAC key.
526     // If the session is bound to the object or the session is a policy session
527     // with no authValue required, do not include the authValue in the HMAC key.
528     // Note: For a policy session, its isBound attribute is CLEARED.
529     if(   s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED
530        && !(      HandleGetType(s_sessionHandles[sessionIndex])
531                == TPM_HT_POLICY_SESSION
532             && session->attributes.isAuthValueNeeded == CLEAR)
533        && !IsSessionBindEntity(s_associatedHandles[sessionIndex], session)
534       )
535     {
536         pAssert((sizeof(AUTH_VALUE) + key.t.size) <= <K>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      // if the HMAC key size for a policy session is 0, a NULL string HMAC is
543      // allowed.
544      if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
545          && key.t.size == 0
546          && s_inputAuthValues[sessionIndex].t.size == 0)
547      {
548          hmac->t.size = 0;
549          return;
550      }
551
552     // Start HMAC
553     hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
554
```

```
555         //  Add cpHash
556         CryptUpdateDigest2B(&hmacState, &cpHash->b);
557 
558         //  Add nonceCaller
559         CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
560 
561         //  Add nonceTPM
562         CryptUpdateDigest2B(&hmacState, &session->nonceTPM.b);
563 
564         //  If needed, add nonceTPM for decrypt session
565         if(nonceDecrypt != NULL)
566             CryptUpdateDigest2B(&hmacState, &nonceDecrypt->b);
567 
568         //  If needed, add nonceTPM for encrypt session
569         if(nonceEncrypt != NULL)
570             CryptUpdateDigest2B(&hmacState, &nonceEncrypt->b);
571 
572         //  Add sessionAttributes
573         buffer = marshalBuffer;
574         marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
575                                            &buffer, NULL);
576         CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
577 
578         // Complete the HMAC computation
579         CryptCompleteHMAC2B(&hmacState, &hmac->b);
580 
581         return;
582     }
```

6.3.4.4 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

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

```
603 }
```

6.3.4.5 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

```
604 static TPM_RC
605 CheckPolicyAuthSession(
606     UINT32          sessionIndex,   // IN: index of session to be processed
607     TPM_CC          commandCode,    // IN: command code
608     TPM2B_DIGEST   *cpHash,         // IN: cpHash using the algorithm of
609                                     //     this session
610     TPM2B_DIGEST   *nameHash        // IN: nameHash using the session algorithm
611 )
612 {
613     TPM_RC           result = TPM_RC_SUCCESS;
614     SESSION         *session;
615     TPM2B_DIGEST     authPolicy;
616     TPMI_ALG_HASH    policyAlg;
617     UINT8            locality;
618
619     // Initialize pointer to the auth session.
620     session = SessionGet(s_sessionHandles[sessionIndex]);
621
622     // If the command is TPM_RC_PolicySecret(), make sure that
623     // either password or authValue is required
624     if(     commandCode == TPM_CC_PolicySecret
625         &&  session->attributes.isPasswordNeeded == CLEAR
626         &&  session->attributes.isAuthValueNeeded == CLEAR)
627         return TPM_RC_MODE;
628
629     // See if the PCR counter for the session is still valid.
630     if( !SessionPCRValueIsCurrent(s_sessionHandles[sessionIndex]) )
631         return TPM_RC_PCR_CHANGED;
632
```

```
633         // Get authPolicy.
634         policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
635                                         &authPolicy);
636         // Compare authPolicy.
637         if(!Memory2BEqual(&session->u2.policyDigest.b, &authPolicy.b))
638             return TPM_RC_POLICY_FAIL;
639
640         // Policy is OK so check if the other factors are correct
641
642         // Compare policy hash algorithm.
643         if(policyAlg != session->authHashAlg)
644             return TPM_RC_POLICY_FAIL;
645
646         // Compare timeout.
647         if(session->timeOut != 0)
648         {
649             // Cannot compare time if clock stop advancing.  An TPM_RC_NV_UNAVAILABLE
650             // or TPM_RC_NV_RATE error may be returned here.
651             result = NvIsAvailable();
652             if(result != TPM_RC_SUCCESS)
653                 return result;
654
655             if(session->timeOut < go.clock)
656                 return TPM_RC_EXPIRED;
657         }
658
659         // If command code is provided it must match
660         if(session->commandCode != 0)
661         {
662             if(session->commandCode != commandCode)
663                 return TPM_RC_POLICY_CC;
664         }
665         else
666         {
667             // If command requires a DUP or ADMIN authorization, the session must have
668             // command code set.
669             AUTH_ROLE   role = CommandAuthRole(commandCode, sessionIndex);
670             if(role == AUTH_ADMIN || role == AUTH_DUP)
671                 return TPM_RC_POLICY_FAIL;
672         }
673         // Check command locality.
674         {
675             BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
676             BYTE        *buffer = sessionLocality;
677
678             // Get existing locality setting in canonical form
679             TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
680
681             // See if the locality has been set
682             if(sessionLocality[0] != 0)
683             {
684                 // If so, get the current locality
685                 locality = _plat__LocalityGet();
686                 if (locality < 5)
687                 {
688                     if(    ((sessionLocality[0] & (1 << locality)) == 0)
689                         || sessionLocality[0] > 31)
690                         return TPM_RC_LOCALITY;
691                 }
692                 else if (locality > 31)
693                 {
694                     if(sessionLocality[0] != locality)
695                         return TPM_RC_LOCALITY;
696                 }
697                 else
698                 {
```

```
699                 // Could throw an assert here but a locality error is just
700                 // as good. It just means that, whatever the locality is, it isn't
701                 // the locality requested so...
702                 return TPM_RC_LOCALITY;
703             }
704         }
705     } // end of locality check
706
707     // Check physical presence.
708     if(   session->attributes.isPPRequired == SET
709        && !_plat__PhysicalPresenceAsserted())
710         return TPM_RC_PP;
711
712     // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
713     // DUP role for this handle.
714     if(session->u1.cpHash.b.size != 0)
715     {
716         if(session->attributes.iscpHashDefined)
717         {
718             // Compare cpHash.
719             if(!Memory2BEqual(&session->u1.cpHash.b, &cpHash->b))
720                 return TPM_RC_POLICY_FAIL;
721         }
722         else
723         {
724             // Compare nameHash.
725             // When cpHash is not defined, nameHash is placed in its space.
726             if(!Memory2BEqual(&session->u1.cpHash.b, &nameHash->b))
727                 return TPM_RC_POLICY_FAIL;
728         }
729     }
730     if(session->attributes.checkNvWritten)
731     {
732         NV_INDEX        nvIndex;
733
734         // If this is not an NV index, the policy makes no sense so fail it.
735         if(HandleGetType(s_associatedHandles[sessionIndex])!= TPM_HT_NV_INDEX)
736             return TPM_RC_POLICY_FAIL;
737
738         // Get the index data
739         NvGetIndexInfo(s_associatedHandles[sessionIndex], &nvIndex);
740
741         // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
742         if(   (nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
743            != (session->attributes.nvWrittenState == SET))
744            return TPM_RC_POLICY_FAIL;
745     }
746
747     return TPM_RC_SUCCESS;
748 }
```

6.3.4.6 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

```
749 static TPM_RC
```

```
RetrieveSessionData (
    TPM_CC       commandCode,       // IN: command code
    UINT32      *sessionCount,      // OUT: number of sessions found
    BYTE        *sessionBuffer,     // IN: pointer to the session buffer
    INT32        bufferSize         // IN: size of the session buffer
)
{
    int          sessionIndex;
    int          i;
    TPM_RC       result;
    SESSION     *session;
    TPM_HT       sessionType;

    s_decryptSessionIndex = UNDEFINED_INDEX;
    s_encryptSessionIndex = UNDEFINED_INDEX;
    s_auditSessionIndex = UNDEFINED_INDEX;

    for(sessionIndex = 0; bufferSize > 0; sessionIndex++)
    {
        // If maximum allowed number of sessions has been parsed, return a size
        // error with a session number that is larger than the number of allowed
        // sessions
        if(sessionIndex == MAX_SESSION_NUM)
            return TPM_RC_SIZE + TPM_RC_S + g_rcIndex[sessionIndex+1];

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

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

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

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

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

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

            //     The nonce size must be zero.
```

```
816             if(s_nonceCaller[sessionIndex].t.size != 0)
817                 return TPM_RC_NONCE + TPM_RC_S + g_rcIndex[sessionIndex];
818 
819             continue;
820         }
821         // For not password sessions...
822 
823         // Find out if the session is loaded.
824         if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
825             return TPM_RC_REFERENCE_S0 + sessionIndex;
826 
827         sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
828         session = SessionGet(s_sessionHandles[sessionIndex]);
829         // Check if the session is an HMAC/policy session.
830         if(    (   session->attributes.isPolicy == SET
831                 && sessionType == TPM_HT_HMAC_SESSION
832                )
833            || (   session->attributes.isPolicy == CLEAR
834                 &&  sessionType == TPM_HT_POLICY_SESSION
835                )
836           )
837             return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
838 
839         // Check that this handle has not previously been used.
840         for(i = 0; i < sessionIndex; i++)
841         {
842             if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
843                 return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
844         }
845 
846         // If the session is used for parameter encryption or audit as well, set
847         // the corresponding indices.
848 
849         // First process decrypt.
850         if(s_attributes[sessionIndex].decrypt)
851         {
852             // Check if the commandCode allows command parameter encryption.
853             if(DecryptSize(commandCode) == 0)
854                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
855 
856             // Encrypt attribute can only appear in one session
857             if(s_decryptSessionIndex != UNDEFINED_INDEX)
858                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
859 
860             // Can't decrypt if the sessioh's symmetrci algorithm is TPM_ALG_NULL
861             if(session->symmetric.algorithm == TPM_ALG_NULL)
862                 return TPM_RC_SYMMETRIC + TPM_RC_S + g_rcIndex[sessionIndex];
863 
864             // All checks passed, so set the index for the session used to decrypt
865             // a command parameter.
866             s_decryptSessionIndex = sessionIndex;
867         }
868 
869         // Now process encrypt.
870         if(s_attributes[sessionIndex].encrypt)
871         {
872             // Check if the commandCode allows response parameter encryption.
873             if(EncryptSize(commandCode) == 0)
874                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
875 
876             // Encrypt attribute can only appear in one session.
877             if(s_encryptSessionIndex != UNDEFINED_INDEX)
878                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
879 
880             // Can't encrypt if the session's symmetrci algorithm is TPM_ALG_NULL
881             if(session->symmetric.algorithm == TPM_ALG_NULL)
```

```
882                 return TPM_RC_SYMMETRIC + TPM_RC_S + g_rcIndex[sessionIndex];
883
884             // All checks passed, so set the index for the session used to encrypt
885             // a response parameter.
886             s_encryptSessionIndex = sessionIndex;
887         }
888
889         // At last process audit.
890         if(s_attributes[sessionIndex].audit)
891         {
892             // Audit attribute can only appear in one session.
893             if(s_auditSessionIndex != UNDEFINED_INDEX)
894                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
895
896             // An audit session can not be policy session.
897             if(   HandleGetType(s_sessionHandles[sessionIndex])
898                == TPM_HT_POLICY_SESSION)
899                 return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
900
901             // If this is a reset of the audit session, or the first use
902             // of the session as an audit session, it doesn't matter what
903             // the exclusive state is. The session will become exclusive.
904             if(   s_attributes[sessionIndex].auditReset == CLEAR
905                && session->attributes.isAudit == SET)
906             {
907                 // Not first use or reset. If auditExlusive is SET, then this
908                 // session must be the current exclusive session.
909                 if(   s_attributes[sessionIndex].auditExclusive == SET
910                    && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
911                    return TPM_RC_EXCLUSIVE;
912             }
913
914             s_auditSessionIndex = sessionIndex;
915         }
916
917         // Initialize associated handle as undefined. This will be changed when
918         // the handles are processed.
919         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
920
921     }
922
923
924     // Set the number of sessions found.
925     *sessionCount = sessionIndex;
926     return TPM_RC_SUCCESS;
927 }
```

6.3.4.7 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

```
928 static TPM_RC
929 CheckLockedOut(
```

```
930         BOOL        lockoutAuthCheck        // IN: TRUE if checking is for lockoutAuth
931     )
932     {
933         TPM_RC      result;
934
935         // If NV is unavailable, and current cycle state recorded in NV is not
936         // SHUTDOWN_NONE, refuse to check any authorization because we would
937         // not be able to handle a DA failure.
938         result = NvIsAvailable();
939         if(result != TPM_RC_SUCCESS && gp.orderlyState != SHUTDOWN_NONE)
940             return result;
941
942         // Check if DA info needs to be updated in NV.
943         if(s_DAPendingOnNV)
944         {
945             // If NV is accessible, ...
946             if(result == TPM_RC_SUCCESS)
947             {
948                 // ... write the pending DA data and proceed.
949                 NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED,
950                                 &gp.lockOutAuthEnabled);
951                 NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
952                 g_updateNV = TRUE;
953                 s_DAPendingOnNV = FALSE;
954             }
955             else
956             {
957                 // Otherwise no authorization can be checked.
958                 return result;
959             }
960         }
961
962         // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
963         // is disabled...
964         if(lockoutAuthCheck)
965         {
966             if(gp.lockOutAuthEnabled == FALSE)
967                 return TPM_RC_LOCKOUT;
968         }
969         else
970         {
971             // ... or if the number of failed tries has been maxed out.
972             if(gp.failedTries >= gp.maxTries)
973                 return TPM_RC_LOCKOUT;
974         }
975         return TPM_RC_SUCCESS;
976     }
```

6.3.4.8 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

```
977  static TPM_RC
978  CheckAuthSession(
979      TPM_CC            commandCode,          // IN: commandCode
980      UINT32            sessionIndex,         // IN: index of session to be processed
981      TPM2B_DIGEST     *cpHash,               // IN: cpHash
982      TPM2B_DIGEST     *nameHash              // IN: nameHash
983  )
984  {
985      TPM_RC            result;
986      SESSION          *session = NULL;
987      TPM_HANDLE        sessionHandle = s_sessionHandles[sessionIndex];
988      TPM_HANDLE        associatedHandle = s_associatedHandles[sessionIndex];
989      TPM_HT            sessionHandleType = HandleGetType(sessionHandle);
990
991      pAssert(sessionHandle != TPM_RH_UNASSIGNED);
992
993      if(sessionHandle != TPM_RS_PW)
994          session = SessionGet(sessionHandle);
995
996      // If the authorization session is not a policy session, or if the policy
997      // session requires authorization, then check lockout.
998      if(    HandleGetType(sessionHandle) != 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(HandleGetType(s_sessionHandles[sessionIndex]) == 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
```

6.3.4.9 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 handles
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
```

6.3.4.10 ParseSessionBuffer()

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

Error Returns	Meaning
Parsing Error	failure

```
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 handles
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                if(session->attributes.isDaBound == SET)
1183                {
1184                    result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1185                    if(result != TPM_RC_SUCCESS)
1186                        return result;
1187                }
1188                // If the current cpHash is the right one, don't re-compute.
1189                if(cpHashAlg != session->authHashAlg)   // different so compute
1190                {
1191                    cpHashAlg = session->authHashAlg;   // save this new algID
```

```
1192                ComputeCpHash(session->authHashAlg, commandCode, handleNum,
1193                              handles, parmBufferSize, parmBufferStart,
1194                              &cpHash, &nameHash);
1195            }
1196            // If this session is for auditing, save the cpHash.
1197            if(s_attributes[sessionIndex].audit)
1198                s_cpHashForAudit = cpHash;
1199        }
1200
1201        // if the session has an associated handle, check the auth
1202        if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1203        {
1204            result = CheckAuthSession(commandCode, sessionIndex,
1205                                      &cpHash, &nameHash);
1206            if(result != TPM_RC_SUCCESS)
1207                return RcSafeAddToResult(result,
1208                                         TPM_RC_S + g_rcIndex[sessionIndex]);
1209        }
1210        else
1211        {
1212            // a session that is not for authorization must either be encrypt,
1213            // decrypt, or audit
1214            if(     s_attributes[sessionIndex].audit == CLEAR
1215                &&  s_attributes[sessionIndex].encrypt == CLEAR
1216                &&  s_attributes[sessionIndex].decrypt == CLEAR)
1217                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
1218
1219            // check HMAC for encrypt/decrypt/audit only sessions
1220            result =  CheckSessionHMAC(sessionIndex, &cpHash);
1221            if(result != TPM_RC_SUCCESS)
1222                return RcSafeAddToResult(result,
1223                                         TPM_RC_S + g_rcIndex[sessionIndex]);
1224        }
1225    }
1226
1227  #ifdef  TPM_CC_GetCommandAuditDigest
1228    // Check if the command should be audited.
1229    result = CheckCommandAudit(commandCode, handleNum, handles,
1230                               parmBufferStart, parmBufferSize);
1231    if(result != TPM_RC_SUCCESS)
1232        return result;              // No session number to reference
1233  #endif
1234
1235    // Decrypt the first parameter if applicable. This should be the last operation
1236    // in session processing.
1237    // If the encrypt session is associated with a handle and the handle's
1238    // authValue is available, then authValue is concatenated with sessionAuth to
1239    // generate encryption key, no matter if the handle is the session bound entity
1240    // or not.
1241    if(s_decryptSessionIndex != UNDEFINED_INDEX)
1242    {
1243        // Get size of the leading size field in decrypt parameter
1244        if(   s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED
1245           && IsAuthValueAvailable(s_associatedHandles[s_decryptSessionIndex],
1246                                   commandCode,
1247                                   s_decryptSessionIndex)
1248          )
1249        {
1250            extraKey.b.size=
1251                EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1252                                   &extraKey.t.buffer);
1253        }
1254        else
1255        {
1256            extraKey.b.size = 0;
1257        }
```

```
1258            size = DecryptSize(commandCode);
1259            result = CryptParameterDecryption(
1260                        s_sessionHandles[s_decryptSessionIndex],
1261                        &s_nonceCaller[s_decryptSessionIndex].b,
1262                        parmBufferSize, (UINT16)size,
1263                        &extraKey,
1264                        parmBufferStart);
1265            if(result != TPM_RC_SUCCESS)
1266                return RcSafeAddToResult(result,
1267                                         TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1268        }
1269
1270        return TPM_RC_SUCCESS;
1271    }
```

6.3.4.11 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

```
1272    TPM_RC
1273    CheckAuthNoSession(
1274        TPM_CC            commandCode,          // IN: Command Code
1275        UINT32            handleNum,            // IN: number of handles in command
1276        TPM_HANDLE        handles[],            // IN: array of handles
1277        BYTE             *parmBufferStart,      // IN: start of parameter buffer
1278        UINT32            parmBufferSize        // IN: size of parameter buffer
1279    )
1280    {
1281        UINT32 i;
1282        TPM_RC            result = TPM_RC_SUCCESS;
1283
1284        // Check if the commandCode requires authorization
1285        for(i = 0; i < handleNum; i++)
1286        {
1287            if(CommandAuthRole(commandCode, i) != AUTH_NONE)
1288                return TPM_RC_AUTH_MISSING;
1289        }
1290
1291    #ifdef  TPM_CC_GetCommandAuditDigest
1292        // Check if the command should be audited.
1293        result = CheckCommandAudit(commandCode, handleNum, handles,
1294                                   parmBufferStart, parmBufferSize);
1295        if(result != TPM_RC_SUCCESS) return result;
1296    #endif
1297
1298        // Initialize number of sessions to be 0
1299        s_sessionNum = 0;
1300
1301        return TPM_RC_SUCCESS;
1302    }
```

6.3.5 Response Session Processing

6.3.5.1 Introduction

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

6.3.5.2 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.

```
1303  static void
1304  ComputeRpHash(
1305      TPM_ALG_ID      hashAlg,            // IN: hash algorithm to compute rpHash
1306      TPM_CC          commandCode,        // IN: commandCode
1307      UINT32          resParmBufferSize,  // IN: size of response parameter buffer
1308      BYTE            *resParmBuffer,     // IN: response parameter buffer
1309      TPM2B_DIGEST    *rpHash             // OUT: rpHash
1310  )
1311  {
1312      // The command result in rpHash is always TPM_RC_SUCCESS.
1313      TPM_RC      responseCode = TPM_RC_SUCCESS;
1314      HASH_STATE  hashState;
1315
1316      //   rpHash := hash(responseCode || commandCode || parameters)
1317
1318      // Initiate hash creation.
1319      rpHash->t.size = CryptStartHash(hashAlg, &hashState);
1320
1321      // Add hash constituents.
1322      CryptUpdateDigestInt(&hashState, sizeof(TPM_RC), &responseCode);
1323      CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
1324      CryptUpdateDigest(&hashState, resParmBufferSize, resParmBuffer);
1325
1326      // Complete hash computation.
1327      CryptCompleteHash2B(&hashState, &rpHash->b);
1328
1329      return;
1330  }
```

6.3.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```
1331  static void
1332  InitAuditSession(
1333      SESSION         *session        // session to be initialized
1334  )
1335  {
1336      // Mark session as an audit session.
1337      session->attributes.isAudit = SET;
1338
1339      // Audit session can not be bound.
1340      session->attributes.isBound = CLEAR;
1341
1342      // Size of the audit log is the size of session hash algorithm digest.
1343      session->u2.auditDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
1344
1345      // Set the original digest value to be 0.
1346      MemorySet(&session->u2.auditDigest.t.buffer,
1347                0,
1348                session->u2.auditDigest.t.size);
1349
1350      return;
1351  }
```

DRAFT

6.3.5.4 Audit()

This function updates the audit digest in an audit session.

```
1352  static void
1353  Audit(
1354      SESSION         *auditSession,      // IN: loaded audit session
1355      TPM_CC           commandCode,       // IN: commandCode
1356      UINT32           resParmBufferSize, // IN: size of response parameter buffer
1357      BYTE            *resParmBuffer      // IN: response parameter buffer
1358  )
1359  {
1360      TPM2B_DIGEST     rpHash;            // rpHash for response
1361      HASH_STATE       hashState;
1362
1363      // Compute rpHash
1364      ComputeRpHash(auditSession->authHashAlg,
1365                    commandCode,
1366                    resParmBufferSize,
1367                    resParmBuffer,
1368                    &rpHash);
1369
1370      // auditDigestnew :=  hash (auditDigestold || cpHash || rpHash)
1371
1372      // Start hash computation.
1373      CryptStartHash(auditSession->authHashAlg, &hashState);
1374
1375      // Add old digest.
1376      CryptUpdateDigest2B(&hashState, &auditSession->u2.auditDigest.b);
1377
1378      // Add cpHash and rpHash.
1379      CryptUpdateDigest2B(&hashState, &s_cpHashForAudit.b);
1380      CryptUpdateDigest2B(&hashState, &rpHash.b);
1381
1382      // Finalize the hash.
1383      CryptCompleteHash2B(&hashState, &auditSession->u2.auditDigest.b);
1384
1385      return;
1386  }
1387  #ifdef  TPM_CC_GetCommandAuditDigest
```

6.3.5.5 CommandAudit()

This function updates the command audit digest.

```
1388  static void
1389  CommandAudit(
1390      TPM_CC      commandCode,        // IN: commandCode
1391      UINT32      resParmBufferSize,  // IN: size of response parameter buffer
1392      BYTE        *resParmBuffer      // IN: response parameter buffer
1393  )
1394  {
1395      if(CommandAuditIsRequired(commandCode))
1396      {
1397          TPM2B_DIGEST    rpHash;         // rpHash for response
1398          HASH_STATE      hashState;
1399
1400          // Compute rpHash.
1401          ComputeRpHash(gp.auditHashAlg, commandCode, resParmBufferSize,
1402                        resParmBuffer, &rpHash);
1403
1404          // If the digest.size is one, it indicates the special case of changing
1405          // the audit hash algorithm. For this case, no audit is done on exit.
1406          // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
```

```
1407            // force an update to the NV on exit so that the change in digest will
1408            // be recorded. So, it is safe to exit here without setting any flags
1409            // because the digest change will be written to NV when this code exits.
1410            if(gr.commandAuditDigest.t.size == 1)
1411            {
1412                gr.commandAuditDigest.t.size = 0;
1413                return;
1414            }
1415
1416            // If the digest size is zero, need to start a new digest and increment
1417            // the audit counter.
1418            if(gr.commandAuditDigest.t.size == 0)
1419            {
1420                gr.commandAuditDigest.t.size = CryptGetHashDigestSize(gp.auditHashAlg);
1421                MemorySet(gr.commandAuditDigest.t.buffer,
1422                          0,
1423                          gr.commandAuditDigest.t.size);
1424
1425                // Bump the counter and save its value to NV.
1426                gp.auditCounter++;
1427                NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
1428                g_updateNV = TRUE;
1429            }
1430
1431            // auditDigestnew :=  hash (auditDigestold || cpHash || rpHash)
1432
1433            //  Start hash computation.
1434            CryptStartHash(gp.auditHashAlg, &hashState);
1435
1436            //  Add old digest.
1437            CryptUpdateDigest2B(&hashState, &gr.commandAuditDigest.b);
1438
1439            //  Add cpHash
1440            CryptUpdateDigest2B(&hashState, &s_cpHashForCommandAudit.b);
1441
1442            //  Add rpHash
1443            CryptUpdateDigest2B(&hashState, &rpHash.b);
1444
1445            //  Finalize the hash.
1446            CryptCompleteHash2B(&hashState, &gr.commandAuditDigest.b);
1447        }
1448        return;
1449    }
1450    #endif
```

6.3.5.6 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.

```
1451    static void
1452    UpdateAuditSessionStatus(
1453        TPM_CC           commandCode,       // IN: commandCode
1454        UINT32           resParmBufferSize, // IN: size of response parameter buffer
1455        BYTE            *resParmBuffer      // IN: response parameter buffer
1456    )
1457    {
```

```
1458        UINT32          i;
1459        TPM_HANDLE      auditSession = TPM_RH_UNASSIGNED;
1460
1461        // Iterate through sessions
1462        for (i = 0; i < s_sessionNum; i++)
1463        {
1464            SESSION     *session;
1465
1466            // PW session do not have a loaded session and can not be an audit
1467            // session either.  Skip it.
1468            if(s_sessionHandles[i] == TPM_RS_PW) continue;
1469
1470            session = SessionGet(s_sessionHandles[i]);
1471
1472            // If a session is used for audit
1473            if(s_attributes[i].audit == SET)
1474            {
1475                // An audit session has been found
1476                auditSession = s_sessionHandles[i];
1477
1478                // If the session has not been an audit session yet, or
1479                // the auditSetting bits indicate a reset, initialize it and set
1480                // it to be the exclusive session
1481                if(   session->attributes.isAudit == CLEAR
1482                   || s_attributes[i].auditReset == SET
1483                  )
1484                {
1485                    InitAuditSession(session);
1486                    g_exclusiveAuditSession = auditSession;
1487                }
1488                else
1489                {
1490                    // Check if the audit session is the current exclusive audit
1491                    // session and, if not, clear previous exclusive audit session.
1492                    if(g_exclusiveAuditSession != auditSession)
1493                        g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1494                }
1495
1496                // Report audit session exclusivity.
1497                if(g_exclusiveAuditSession == auditSession)
1498                {
1499                    s_attributes[i].auditExclusive = SET;
1500                }
1501                else
1502                {
1503                    s_attributes[i].auditExclusive = CLEAR;
1504                }
1505
1506                // Extend audit log.
1507                Audit(session, commandCode, resParmBufferSize, resParmBuffer);
1508            }
1509        }
1510
1511        // If no audit session is found in the command, and the command allows
1512        // a session then, clear the current exclusive
1513        // audit session.
1514        if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(commandCode))
1515        {
1516            g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1517        }
1518
1519        return;
1520    }
```

6.3.5.7 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```
1521  static void
1522  ComputeResponseHMAC(
1523      UINT32          sessionIndex,           // IN: session index to be processed
1524      SESSION         *session,               // IN: loaded session
1525      TPM_CC          commandCode,            // IN: commandCode
1526      TPM2B_NONCE     *nonceTPM,              // IN: nonceTPM
1527      UINT32          resParmBufferSize,      // IN: size of response parameter
1528      //     buffer
1529      BYTE            *resParmBuffer,         // IN: response parameter buffer
1530      TPM2B_DIGEST    *hmac                   // OUT: authHMAC
1531  )
1532  {
1533      TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1534      TPM2B_KEY        key;       // HMAC key
1535      BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
1536      BYTE            *buffer;
1537      UINT32           marshalSize;
1538      HMAC_STATE       hmacState;
1539      TPM2B_DIGEST     rp_hash;
1540
1541      // Compute rpHash.
1542      ComputeRpHash(session->authHashAlg, commandCode, resParmBufferSize,
1543                    resParmBuffer, &rp_hash);
1544
1545      // Generate HMAC key
1546      MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1547
1548      // Check if the session has an associated handle and the associated entity is
1549      // the one that the session is bound to.
1550      // If not bound, add the authValue of this entity to the HMAC key.
1551      if(   s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED
1552         &&    !( HandleGetType(s_sessionHandles[sessionIndex])
1553                   == TPM_HT_POLICY_SESSION
1554            &&   session->attributes.isAuthValueNeeded == CLEAR)
1555         && !session->attributes.requestWasBound)
1556      {
1557          pAssert((sizeof(AUTH_VALUE) + key.t.size) <= <K>sizeof(key.t.buffer));
1558          key.t.size = key.t.size +
1559                       EntityGetAuthValue(s_associatedHandles[sessionIndex],
1560                                          (AUTH_VALUE *)&key.t.buffer[key.t.size]);
1561      }
1562
1563      // if the HMAC key size for a policy session is 0, the response HMAC is
1564      // computed according to the input HMAC
1565      if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1566          && key.t.size == 0
1567          && s_inputAuthValues[sessionIndex].t.size == 0)
1568      {
1569          hmac->t.size = 0;
1570          return;
1571      }
1572
1573      // Start HMAC computation.
1574      hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);
1575
1576      // Add hash components.
1577      CryptUpdateDigest2B(&hmacState, &rp_hash.b);
1578      CryptUpdateDigest2B(&hmacState, &nonceTPM->b);
1579      CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);
1580
1581      // Add session attributes.
```

```
1582        buffer = marshalBuffer;
1583        marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1584        CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);
1585
1586        // Finalize HMAC.
1587        CryptCompleteHMAC2B(&hmacState, &hmac->b);
1588
1589        return;
1590    }
```

6.3.5.8 BuildSingleResponseAuth()

Function to compute response for an authorization session.

```
1591    static void
1592    BuildSingleResponseAuth(
1593        UINT32       sessionIndex,          // IN: session index to be processed
1594        TPM_CC       commandCode,           // IN: commandCode
1595        UINT32       resParmBufferSize,     // IN: size of response parameter buffer
1596        BYTE        *resParmBuffer,         // IN: response parameter buffer
1597        TPM2B_AUTH  *auth                   // OUT: authHMAC
1598    )
1599    {
1600        // For password authorization, field is empty.
1601        if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1602        {
1603            auth->t.size = 0;
1604        }
1605        else
1606        {
1607            // Fill in policy/HMAC based session response.
1608            SESSION     *session = SessionGet(s_sessionHandles[sessionIndex]);
1609
1610            // If the session is a policy session with isPasswordNeeded SET, the auth
1611            // field is empty.
1612            if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1613                    && session->attributes.isPasswordNeeded == SET)
1614                auth->t.size = 0;
1615            else
1616                // Compute response HMAC.
1617                ComputeResponseHMAC(sessionIndex,
1618                                    session,
1619                                    commandCode,
1620                                    &session->nonceTPM,
1621                                    resParmBufferSize,
1622                                    resParmBuffer,
1623                                    auth);
1624        }
1625
1626        return;
1627    }
```

6.3.5.9 UpdateTPMNonce()

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

```
1628    static void
1629    UpdateTPMNonce(
1630        TPM2B_NONCE      nonces[]            // OUT: nonceTPM
1631    )
1632    {
1633        UINT32      i;
1634        for(i = 0; i < s_sessionNum; i++)
```

```
1635        {
1636            SESSION     *session;
1637            // For PW session, nonce is 0.
1638            if(s_sessionHandles[i] == TPM_RS_PW)
1639            {
1640                nonces[i].t.size = 0;
1641                continue;
1642            }
1643            session = SessionGet(s_sessionHandles[i]);
1644            // Update nonceTPM in both internal session and response.
1645            CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
1646            nonces[i] = session->nonceTPM;
1647        }
1648        return;
1649    }
```

6.3.5.10 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time.

b) Clears a policy session since nonce is rolling.

```
1650    static void
1651    UpdateInternalSession(void)
1652    {
1653        UINT32      i;
1654        for(i = 0; i < s_sessionNum; i++)
1655        {
1656            // For PW session, no update.
1657            if(s_sessionHandles[i] == TPM_RS_PW) continue;
1658
1659            if(s_attributes[i].continueSession == CLEAR)
1660            {
1661                // Close internal session.
1662                SessionFlush(s_sessionHandles[i]);
1663            }
1664            else
1665            {
1666                // If nonce is rolling in a policy session, the policy related data
1667                // will be re-initialized.
1668                if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION)
1669                {
1670                    SESSION     *session = SessionGet(s_sessionHandles[i]);
1671
1672                    // When the nonce rolls it starts a new timing interval for the
1673                    // policy session.
1674                    SessionResetPolicyData(session);
1675                    session->startTime = go.clock;
1676                }
1677            }
1678        }
1679        return;
1680    }
```

6.3.5.11 BuildResponseSession()

Function to build Session buffer in a response.

```
1681    void
1682    BuildResponseSession(
1683        TPM_ST      tag,                    // IN: tag
```

```
1684        TPM_CC      commandCode,            // IN: commandCode
1685        UINT32      resHandleSize,          // IN: size of response handle buffer
1686        UINT32      resParmSize,            // IN: size of response parameter buffer
1687        UINT32      *resSessionSize         // OUT: response session area
1688    )
1689    {
1690        BYTE            *resParmBuffer;
1691        TPM2B_NONCE  responseNonces[MAX_SESSION_NUM];
1692
1693        // Compute response parameter buffer start.
1694        resParmBuffer = MemoryGetResponseBuffer(commandCode) + sizeof(TPM_ST) +
1695                        sizeof(UINT32) + sizeof(TPM_RC) + resHandleSize;
1696
1697        // For TPM_ST_SESSIONS, there is parameterSize field.
1698        if(tag == TPM_ST_SESSIONS)
1699            resParmBuffer += sizeof(UINT32);
1700
1701        // Session nonce should be updated before parameter encryption
1702        if(tag == TPM_ST_SESSIONS)
1703        {
1704            UpdateTPMNonce(responseNonces);
1705
1706            // Encrypt first parameter if applicable. Parameter encryption should
1707            // happen after nonce update and before any rpHash is computed.
1708            // If the encrypt session is associated with a handle, the authValue of
1709            // this handle will be concatenated with sessionAuth to generate
1710            // encryption key, no matter if the handle is the session bound entity
1711            // or not. The authValue is added to sessionAuth only when the authValue
1712            // is available.
1713            if(s_encryptSessionIndex != UNDEFINED_INDEX)
1714            {
1715                UINT32          size;
1716                TPM2B_AUTH      extraKey;
1717
1718                // Get size of the leading size field
1719                if(   s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED
1720                   && IsAuthValueAvailable(s_associatedHandles[s_encryptSessionIndex],
1721                                           commandCode, s_encryptSessionIndex)
1722                  )
1723                {
1724                    extraKey.b.size =
1725                        EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1726                                           &extraKey.t.buffer);
1727                }
1728                else
1729                {
1730                    extraKey.b.size = 0;
1731                }
1732                size = EncryptSize(commandCode);
1733                CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1734                                         &s_nonceCaller[s_encryptSessionIndex].b,
1735                                         (UINT16)size,
1736                                         &extraKey,
1737                                         resParmBuffer);
1738
1739            }
1740
1741        }
1742        // Audit session should be updated first regardless of the tag.
1743        // A command with no session may trigger a change of the exclusivity state.
1744        UpdateAuditSessionStatus(commandCode, resParmSize, resParmBuffer);
1745
1746        // Audit command.
1747        CommandAudit(commandCode, resParmSize, resParmBuffer);
1748
1749        // Process command with sessions.
```

```
1750        if(tag == TPM_ST_SESSIONS)
1751        {
1752            UINT32           i;
1753            BYTE            *buffer;
1754            TPM2B_DIGEST     responseAuths[MAX_SESSION_NUM];
1755
1756            pAssert(s_sessionNum > 0);
1757
1758            // Iterate over each session in the command session area, and create
1759            // corresponding sessions for response.
1760            for(i = 0; i < s_sessionNum; i++)
1761            {
1762                BuildSingleResponseAuth(
1763                                        i,
1764                                        commandCode,
1765                                        resParmSize,
1766                                        resParmBuffer,
1767                                        &responseAuths[i]);
1768                // Make sure that continueSession is SET on any Password session.
1769                // This makes it marginally easier for the management software
1770                // to keep track of the closed sessions.
1771                if(   s_attributes[i].continueSession == CLEAR
1772                   && s_sessionHandles[i] == TPM_RS_PW)
1773                {
1774                    s_attributes[i].continueSession = SET;
1775                }
1776            }
1777
1778            // Assemble Response Sessions.
1779            *resSessionSize = 0;
1780            buffer = resParmBuffer + resParmSize;
1781            for(i = 0; i < s_sessionNum; i++)
1782            {
1783                *resSessionSize += TPM2B_NONCE_Marshal(&responseNonces[i],
1784                                                      &buffer, NULL);
1785                *resSessionSize += TPMA_SESSION_Marshal(&s_attributes[i],
1786                                                       &buffer, NULL);
1787                *resSessionSize += TPM2B_DIGEST_Marshal(&responseAuths[i],
1788                                                       &buffer, NULL);
1789            }
1790
1791            // Update internal sessions after completing response buffer computation.
1792            UpdateInternalSession();
1793        }
1794        else
1795        {
1796            // Process command with no session.
1797            *resSessionSize = 0;
1798        }
1799
1800        return;
1801    }
```

DRAFT

7 Command Support Functions

7.1 Introduction

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

7.2 Attestation Command Support (Attest_spt.c)

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

7.2.1.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

Error Returns	Meaning
TPM_RC_KEY	key referenced by *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))
            attest->qualifiedSigner.t.size = 0;
        else
            ObjectGetQualifiedName(signHandle, &attest->qualifiedSigner);
    }

    // current clock in plain text
```

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

7.2.1.2 SignAttestInfo()

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

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*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)

```
89      TPM_RC
```

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


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

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

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

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

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

```
156
157         return TPM_RC_SUCCESS;
158     }
```

DRAFT

7.3 Context Management Command Support (Context_spt.c)

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

7.3.1.1 ComputeContextProtectionKey()

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

```
void
ComputeContextProtectionKey(
    TPMS_CONTEXT        *contextBlob,   // IN: context blob
    TPM2B_SYM_KEY       *symKey,        // OUT: the symmetric key
    TPM2B_IV            *iv             // OUT: the IV.
)
{
    UINT16           symKeyBits;    // number of bits in the parent's
                                    //   symmetric key
    TPM2B_AUTH      *proof = NULL;  // the proof value to use. Is null for
                                    //   everything but a primary object in
                                    //   the Endorsement Hierarchy

    BYTE             kdfResult[sizeof(TPMU_HA) * 2];// Value produced by the KDF

    TPM2B_DATA       sequence2B, handle2B;

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

    // Get sequence value in 2B format
    sequence2B.t.size = sizeof(contextBlob->sequence);
    MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence,
               sizeof(contextBlob->sequence),
               sizeof(sequence2B.t.buffer));

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

    // Get the symmetric encryption key size
    symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
    symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
    // Get the size of the IV for the algorithm
    iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);

    // KDFa to generate symmetric key and IV value
    KDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, "CONTEXT", &sequence2B.b,
         &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL);

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

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

    return;
}
```

7.3.1.2 ComputeContextIntegrity()

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

```
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
76      // Adding total reset counter so that the context cannot be
77      // used after a TPM Reset
78      CryptUpdateDigestInt(&hmacState, sizeof(gp.totalResetCount),
79                           &gp.totalResetCount);
80
81      // If this is a ST_CLEAR object, add the clear coount
82      // so taht this contest cannot be loaded after a TPM Restart
83      if(contextBlob->savedHandle == 0x80000002)
84          CryptUpdateDigestInt(&hmacState, sizeof(gr.clearCount), &gr.clearCount);
85
86      // Adding sequence number to the HMAC to make sure that it doesn't
87      // get changed
88      CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->sequence),
89                           &contextBlob->sequence);
90
91      // Protect the handle
92      CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->savedHandle),
93                           &contextBlob->savedHandle);
94
95      // Adding sensitive contextData, skip the leading integrity area
96      CryptUpdateDigest(&hmacState, contextBlob->contextBlob.t.size - integritySize,
97                        contextBlob->contextBlob.t.buffer + integritySize);
98
99      // Complete HMAC
100     CryptCompleteHMAC2B(&hmacState, &integrity->b);
101
102     return;
103 }
```

7.3.1.3 SequenceDataImportExport()

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

```
104 void
105 SequenceDataImportExport(
106     OBJECT              *object,        // IN: the object containing the
```

```
107                                          //      sequence data
108      OBJECT              *exportObject,  // IN/OUT: the object structure
109                                          //      that will get the exporte hash state
110      IMPORT_EXPORT        direction
111     )
112  {
113      int                   count = 1;
114      HASH_OBJECT          *internalFmt = (HASH_OBJECT *)object;
115      HASH_OBJECT          *externalFmt = (HASH_OBJECT *)exportObject;
116
117
118      if(object->attributes.eventSeq)
119          count = HASH_COUNT;
120      for(; count; count--)
121          CryptHashStateImportExport(&internalFmt->state.hashState[count - 1],
122                                     externalFmt->state.hashState, direction);
123  }
```

7.4 Policy Command Support (Policy_spt.c)

```
#include "InternalRoutines.h"
#include "Policy_spt_fp.h"
#include "PolicySigned_fp.h"
#include "PolicySecret_fp.h"
#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*.

```
TPM_RC
PolicyParameterChecks(
    SESSION          *session,
    UINT64            authTimeout,
    TPM2B_DIGEST     *cpHashA,
    TPM2B_NONCE      *nonce,
    TPM_RC            nonceParameterNumber,
    TPM_RC            cpHashParameterNumber,
    TPM_RC            expirationParameterNumber
    )
{
    TPM_RC            result;
    // Validate that input nonceTPM is correct if present
    if(nonce != NULL && nonce->t.size != 0)
    {
        if(!Memory2BEqual(&nonce->b, &session->nonceTPM.b))
            return TPM_RC_NONCE + RC_PolicySigned_nonceTPM;
    }
    // If authTimeout is set (expiration != 0...
    if(authTimeout != 0)
    {
        // ...then nonce must be present
        // nonce present isn't checked in PolicyTicket
        if(nonce != NULL && nonce->t.size == 0)
            // This error says that the time has expired but it is pointing
            // at the nonceTPM value.
            return TPM_RC_EXPIRED + nonceParameterNumber;

        // Validate input expiration.
        // Cannot compare time if clock stop advancing.  A TPM_RC_NV_UNAVAILABLE
        // or TPM_RC_NV_RATE error may be returned here.
        result = NvIsAvailable();
        if(result != TPM_RC_SUCCESS)
            return result;

        if(authTimeout < go.clock)
            return TPM_RC_EXPIRED + expirationParameterNumber;
    }
    // If the cpHash is present, then check it
    if(cpHashA != NULL && cpHashA->t.size != 0)
    {
        // The cpHash input has to have the correct size
        if(cpHashA->t.size != session->u2.policyDigest.t.size)
            return TPM_RC_SIZE + cpHashParameterNumber;

        // If the cpHash has already been set, then this input value
        // must match the curent value.
        if(     session->u1.cpHash.b.size != 0
            &&  !Memory2BEqual(&cpHashA->b, &session->u1.cpHash.b))
                return TPM_RC_CPHASH;
```

DRAFT

```
56      }
57      return TPM_RC_SUCCESS;
58  }
```

7.4.2 UpdateTimeout()

Update timeout in a policy session

```
59  void
60  UpdateTimeout(
61      UINT64               timeout,           // IN: the new timeout value
62      SESSION             *session            // IN: the session
63  )
64  {
65      // only do something is a timeout is specified
66      if(timeout != 0)
67      {
68      // If the timeout has not been set, then set it to the new value
69          if(session->timeOut == 0)
70              session->timeOut = timeout;
71          else if(session->timeOut > timeout)
72              session->timeOut =  timeout;
73      }
74      return;
75  }
```

7.4.3 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.

```
76  void
77  PolicyContextUpdate(
78      TPM_CC               commandCode,       // IN: command code
79      TPM2B_NAME          *name,              // IN: name of entity
80      TPM2B_NONCE         *ref,               // IN: the reference data
81      TPM2B_DIGEST        *cpHash,            // IN: the cpHash (optional)
82      UINT64               policyTimeout,
83      SESSION             *session            // IN/OUT: policy session to be updated
84  )
85  {
86      HASH_STATE           hashState;
87      UINT16               policyDigestSize;
88
89      // Start hash
90      policyDigestSize = CryptStartHash(session->authHashAlg, &hashState);
91
92      // policyDigest size should always be the digest size of session hash alg.
93      pAssert(session->u2.policyDigest.t.size == policyDigestSize);
94
95      // add old digest
96      CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);
97
98      // add commandCode
99      CryptUpdateDigestInt(&hashState, sizeof(commandCode), &commandCode);
100
101     // add name if applicable
102     if(name != NULL)
103         CryptUpdateDigest2B(&hashState, &name->b);
104
105     // Complete the digest and get the results
106     CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);
107
```

```
108         // Start second hash computation
109         CryptStartHash(session->authHashAlg, &hashState);
110 
111         // add policyDigest
112         CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);
113 
114         // add policyRef
115         if(ref != NULL)
116             CryptUpdateDigest2B(&hashState, &ref->b);
117 
118         // Complete second digest
119         CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);
120 
121         // Deal with the cpHash. If the cpHash value is present
122         // then it would have already been checked to make sure that
123         // it is compatible with the current value so all we need
124         // to do here is copy it and set the iscoHashDefined attribute
125         if(cpHash != NULL && cpHash->t.size != 0)
126         {
127             session->u1.cpHash = *cpHash;
128             session->attributes.iscpHashDefined = SET;
129         }
130 
131         // update the timeout if it is specified
132         if(policyTimeout!= 0)
133         {
134         // If the timeout has not been set, then set it to the new value
135             if(session->timeOut == 0)
136                 session->timeOut = policyTimeout;
137             else if(session->timeOut > policyTimeout)
138                 session->timeOut =  policyTimeout;
139         }
140 
141         return;
142     }
```

DRAFT

7.5 NV Command Support (NV_spt.c)

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

7.5.1 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	*authHandle* is not allowed to authorize read of the index
TPM_RC_NV_LOCKED	Read locked
TPM_RC_NV_UNINITIALIZED	Try to read an uninitialized index

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

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

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

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

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

    return TPM_RC_SUCCESS;
}
```

7.5.2 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

```
47  TPM_RC
48  NvWriteAccessChecks(
49      TPM_HANDLE      authHandle,     // IN: the handle that provided the
50                                      //     authorization
51      TPM_HANDLE      nvHandle        // IN: the handle of the NV index to be
52                                      //     written
53  )
54  {
55      NV_INDEX         nvIndex;
56
57      // Get NV index info
58      NvGetIndexInfo(nvHandle, &nvIndex);
59
60      // If data is write locked, returns an error
61      if(nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED == SET)
62          return TPM_RC_NV_LOCKED;
63
64      // If the authorization was provided by the owner or platform, then check
65      // that the attributes allow the write.  If the authorization handle
66      // is the same as the index, then the checks were made when the authorization
67      // was checked..
68      if(authHandle == TPM_RH_OWNER)
69      {
70          // If Owner provided auth then ONWERWRITE must be SET
71          if(! nvIndex.publicArea.attributes.TPMA_NV_OWNERWRITE)
72              return TPM_RC_NV_AUTHORIZATION;
73      }
74      else if(authHandle == TPM_RH_PLATFORM)
75      {
76          // If Platform provided auth then PPWRITE must be SET
77          if(!nvIndex.publicArea.attributes.TPMA_NV_PPWRITE)
78              return TPM_RC_NV_AUTHORIZATION;
79      }
80      // If neither Owner nor Platform provided auth, make sure that it was
81      // provided by this index.
82      else if(authHandle != nvHandle)
83              return TPM_RC_NV_AUTHORIZATION;
84
85      return TPM_RC_SUCCESS;
86  }
```

DRAFT

7.6 Object Command Support (Object_spt.c)

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

7.6.1 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

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

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

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

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

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

    return TPM_RC_SUCCESS;
}
```

7.6.2 AreAttributesForParent()

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

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

```
37  BOOL
38  AreAttributesForParent(
39      OBJECT      *parentObject           // IN: parent handle
40  )
41  {
42      // This function is only called when a parent is needed. Any
43      // time a "parent" is used, it must be authorized. When
44      // the authorization is checked, both the public and sensitivve
45      // areas must be loaded. Just make sure...
46      pAssert(parentObject->attributes.publicOnly == CLEAR);
47
48
49      if(ObjectDataIsStorage(&parentObject->publicArea))
50          return TRUE;
51      else
52          return FALSE;
53  }
```

7.6.3 SchemeChecks()

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

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public params
TPM_RC_ATTRIBUTES	attempt to inject sensitive data for an asymmetric key; or attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area
TPM_RC_SCHEME	inconsistent attributes *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

```
54  TPM_RC
55  SchemeChecks(
56      BOOL                load,               // IN: TRUE if load checks, FALSE if
57                                              //     TPM2_Create()
58      TPMI_DH_OBJECT      parentHandle,       // IN: input parent handle
59      TPMT_PUBLIC         *publicArea         // IN: public area of the object
60  )
61  {
62
63      // Checks for an asymmetric key
64      if(CryptIsAsymAlgorithm(publicArea->type))
```

```
65      {
66          TPMT_ASYM_SCHEME        *keyScheme;
67          keyScheme = &publicArea->parameters.asymDetail.scheme;
68
69          // An asymmetric key can't be injected
70          // This is only checked when creating an object
71          if(!load && (publicArea->objectAttributes.sensitiveDataOrigin == CLEAR))
72              return TPM_RC_ATTRIBUTES;
73
74          if(load && !CryptAreKeySizesConsistent(publicArea))
75              return TPM_RC_KEY;
76
77          // Keys that are both signing and decrypting must have TPM_ALG_NULL
78          // for scheme
79          if(    publicArea->objectAttributes.sign == SET
80             && publicArea->objectAttributes.decrypt == SET
81             && keyScheme->scheme != TPM_ALG_NULL)
82              return TPM_RC_SCHEME;
83
84          // A restrict sign key must have a non-NULL scheme
85          if(     publicArea->objectAttributes.restricted == SET
86              &&  publicArea->objectAttributes.sign == SET
87              &&  keyScheme->scheme == TPM_ALG_NULL)
88              return TPM_RC_SCHEME;
89
90          // Keys must have a valid sign or decrypt scheme, or a TPM_ALG_NULL
91          // scheme
92          if(   keyScheme->scheme != TPM_ALG_NULL
93             && (    (   publicArea->objectAttributes.sign == SET
94                      && !CryptIsSignScheme(keyScheme->scheme)
95                     )
96                  || (   publicArea->objectAttributes.decrypt == SET
97                      && !CryptIsDecryptScheme(keyScheme->scheme)
98                     )
99                 )
100           )
101             return TPM_RC_SCHEME;
102
103         // Special checks for an ECC key
104 #ifdef TPM_ALG_ECC
105         if(publicArea->type == TPM_ALG_ECC)
106         {
107             TPM_ECC_CURVE        curveID = publicArea->parameters.eccDetail.curveID;
108             const TPMT_ECC_SCHEME  *curveScheme = CryptGetCurveSignScheme(curveID);
109             // The curveId must be valid or the unmarshaling is busted.
110             pAssert(curveScheme != NULL);
111
112             // If the curveID requires a specific scheme, then the key must select
113             // the same scheme
114             if(curveScheme->scheme != TPM_ALG_NULL)
115             {
116                 if(keyScheme->scheme != curveScheme->scheme)
117                     return TPM_RC_SCHEME;
118                 // The scheme can allow any hash, or not...
119                 if(   curveScheme->details.any.hashAlg != TPM_ALG_NULL
120                    && (   keyScheme->details.anySig.hashAlg
121                        != curveScheme->details.any.hashAlg
122                       )
123                   )
124                     return TPM_RC_SCHEME;
125             }
126             // For now, the KDF must be TPM_ALG_NULL
127             if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
128                 return TPM_RC_KDF;
129         }
130 #endif
```

```
131
132         // Checks for a storage key (restricted + decryption)
133         if(   publicArea->objectAttributes.restricted == SET
134            && publicArea->objectAttributes.decrypt == SET)
135         {
136             // A storage key must have a valid protection key
137             if(   publicArea->parameters.asymDetail.symmetric.algorithm
138                == TPM_ALG_NULL)
139                 return TPM_RC_SYMMETRIC;
140
141             // A storage key must have a null scheme
142             if(publicArea->parameters.asymDetail.scheme.scheme != TPM_ALG_NULL)
143                 return TPM_RC_SCHEME;
144
145             // A storage key must match its parent algorithms unless
146             // it is duplicable or a primary (including Temporary Primary Objects)
147             if(   HandleGetType(parentHandle) != TPM_HT_PERMANENT
148                && publicArea->objectAttributes.fixedParent == SET
149               )
150             {
151                 // If the object to be created is a storage key, and is fixedParent,
152                 // its crypto set has to match its parent's crypto set. TPM_RC_TYPE,
153                 // TPM_RC_HASH or TPM_RC_ASYMMETRIC may be returned at this point
154                 return EqualCryptSet(publicArea,
155                                      &(ObjectGet(parentHandle)->publicArea));
156             }
157         }
158         else
159         {
160             // Non-storage keys must have TPM_ALG_NULL for the symmetric algorithm
161             if(   publicArea->parameters.asymDetail.symmetric.algorithm
162                != TPM_ALG_NULL)
163                 return TPM_RC_SYMMETRIC;
164
165         }// End of asymmetric decryption key checks
166     } // End of asymmetric checks
167
168     // Check for bit attributes
169     else if(publicArea->type == TPM_ALG_KEYEDHASH)
170     {
171         TPMT_KEYEDHASH_SCHEME   *scheme
172             = &publicArea->parameters.keyedHashDetail.scheme;
173         // If both sign and decrypt are set the scheme must be TPM_ALG_NULL
174         // and the scheme selected when the key is used.
175         // If neither sign nor decrypt is set, the scheme must be TPM_ALG_NULL
176         // because this is a data object.
177         if(    (    publicArea->objectAttributes.sign == SET
178                  && publicArea->objectAttributes.decrypt == SET
179                 )
180            || (   publicArea->objectAttributes.sign == CLEAR
181                && publicArea->objectAttributes.decrypt == CLEAR
182               )
183           )
184         {
185             if(scheme->scheme != TPM_ALG_NULL)
186                 return TPM_RC_SCHEME;
187             return TPM_RC_SUCCESS;
188         }
189         // If this is a decryption key, make sure that is is XOR and that there
190         // is a KDF
191         else if(publicArea->objectAttributes.decrypt)
192         {
193             if(   scheme->scheme != TPM_ALG_XOR
194                || scheme->details.xor.hashAlg == TPM_ALG_NULL)
195                 return TPM_RC_SCHEME;
196             if(scheme->details.xor.kdf == TPM_ALG_NULL)
```

```
197                 return TPM_RC_KDF;
198             return TPM_RC_SUCCESS;
199
200         }
201         // only supported signing scheme for keyedHash object is HMAC
202         if(   scheme->scheme != TPM_ALG_HMAC
203            || scheme->details.hmac.hashAlg == TPM_ALG_NULL)
204             return TPM_RC_SCHEME;
205
206         // end of the checks for keyedHash
207         return TPM_RC_SUCCESS;
208     }
209     else if (publicArea->type == TPM_ALG_SYMCIPHER)
210     {
211         // Must be a decrypting key and may not be a signing key
212         if(   publicArea->objectAttributes.decrypt == CLEAR
213            || publicArea->objectAttributes.sign == SET
214           )
215             return TPM_RC_ATTRIBUTES;
216         pAssert(publicArea->type != TPM_ALG_NULL);
217     }
218     else
219         return TPM_RC_TYPE;
220
221     return TPM_RC_SUCCESS;
222 }
```

7.6.4 PublicAttributesValidation()

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

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public params
TPM_RC_ATTRIBUTES	*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

```
223 TPM_RC
224 PublicAttributesValidation(
225     BOOL                load,           // IN: TRUE if load checks, FALSE if
```

```
226                                             //     TPM2_Create()
227      TPMI_DH_OBJECT       parentHandle,      // IN: input parent handle
228      TPMT_PUBLIC         *publicArea         // IN: public area of the object
229  )
230  {
231      OBJECT              *parentObject = NULL;
232
233      if(HandleGetType(parentHandle) != TPM_HT_PERMANENT)
234          parentObject = ObjectGet(parentHandle);
235
236      // Check authPolicy digest consistency
237      if(   publicArea->authPolicy.t.size != 0
238         && (   publicArea->authPolicy.t.size
239             != CryptGetHashDigestSize(publicArea->nameAlg)
240            )
241        )
242          return TPM_RC_SIZE;
243
244      // If the parent is fixedTPM (including a Primary Object) the object must have
245      // the same value for fixedTPM and fixedParent
246      if(    parentObject == NULL
247          || parentObject->publicArea.objectAttributes.fixedTPM == SET)
248      {
249          if(   publicArea->objectAttributes.fixedParent
250             != publicArea->objectAttributes.fixedTPM
251            )
252              return TPM_RC_ATTRIBUTES;
253      }
254      else
255          // The parent is not fixedTPM so the object can't be fixedTPM
256          if(publicArea->objectAttributes.fixedTPM == SET)
257              return  TPM_RC_ATTRIBUTES;
258
259      // A restricted object cannot be both sign and decrypt and it can't be neither
260      // sign not decrypt
261      if (   publicArea->objectAttributes.restricted == SET
262          && (   publicArea->objectAttributes.decrypt
263              == publicArea->objectAttributes.sign)
264         )
265          return TPM_RC_ATTRIBUTES;
266
267      // A fixedTPM object can not have encryptedDuplication bit SET
268      if(    publicArea->objectAttributes.fixedTPM == SET
269          && publicArea->objectAttributes.encryptedDuplication == SET)
270          return TPM_RC_ATTRIBUTES;
271
272      // If a parent object has fixedTPM CLEAR, the child must have the
273      // same encryptedDuplication value as its parent.
274      // Primary objects are considered to have a fixedTPM parent (the seeds).
275     if(      (   parentObject != NULL
276                && parentObject->publicArea.objectAttributes.fixedTPM == CLEAR)
277         // Get here if parent is not fixed TPM
278         &&  (   publicArea->objectAttributes.encryptedDuplication
279              != parentObject->publicArea.objectAttributes.encryptedDuplication
280             )
281        )
282          return TPM_RC_ATTRIBUTES;
283
284     return SchemeChecks(load, parentHandle, publicArea);
285  }
```

7.6.5 FillInCreationData()

Fill in creation data for an object.

```
286 void
287 FillInCreationData(
288     TPMI_DH_OBJECT       parentHandle,      // IN: handle of parent
289     TPMI_ALG_HASH        nameHashAlg,       // IN: name hash algorithm
290     TPML_PCR_SELECTION  *creationPCR,       // IN: PCR selection
291     TPM2B_DATA          *outsideData,       // IN: outside data
292     TPM2B_CREATION_DATA *outCreation,       // OUT: creation data for output
293     TPM2B_DIGEST        *creationDigest     // OUT: creation digest
294 )
295 {
296     BYTE                 creationBuffer[sizeof(TPMS_CREATION_DATA)];
297     BYTE                *buffer;
298     HASH_STATE           hashState;
299
300     // Fill in TPMS_CREATION_DATA in outCreation
301
302     // Compute PCR digest
303     PCRComputeCurrentDigest(nameHashAlg, creationPCR,
304                             &outCreation->t.creationData.pcrDigest);
305
306     // Put back PCR selection list
307     outCreation->t.creationData.pcrSelect = *creationPCR;
308
309     // Get locality
310     outCreation->t.creationData.locality
311         = LocalityGetAttributes(_plat__LocalityGet());
312
313     outCreation->t.creationData.parentNameAlg = TPM_ALG_NULL;
314
315     // If the parent is is either a primary seed or TPM_ALG_NULL, then  the Name
316     // and QN of the parent are the parent's handle.
317     if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
318     {
319         BYTE        *buffer = &outCreation->t.creationData.parentName.t.name[0];
320         outCreation->t.creationData.parentName.t.size =
321             TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
322
323         // Parent qualified name of a Temporary Object is the same as parent's
324         // name
325         MemoryCopy2B(&outCreation->t.creationData.parentQualifiedName.b,
326                      &outCreation->t.creationData.parentName.b,
327                     sizeof(outCreation->t.creationData.parentQualifiedName.t.name));
328
329     }
330     else         // Regular object
331     {
332         OBJECT          *parentObject = ObjectGet(parentHandle);
333
334         // Set name algorithm
335         outCreation->t.creationData.parentNameAlg =
336             parentObject->publicArea.nameAlg;
337         // Copy parent name
338         outCreation->t.creationData.parentName = parentObject->name;
339
340         // Copy parent qualified name
341         outCreation->t.creationData.parentQualifiedName =
342             parentObject->qualifiedName;
343     }
344
345     // Copy outside information
346     outCreation->t.creationData.outsideInfo = *outsideData;
347
348     // Marshal creation data to canonical form
349     buffer = creationBuffer;
350     outCreation->t.size = TPMS_CREATION_DATA_Marshal(&outCreation->t.creationData,
351                           &buffer, NULL);
```

```
352
353         // Compute hash for creation field in public template
354         creationDigest->t.size = CryptStartHash(nameHashAlg, &hashState);
355         CryptUpdateDigest(&hashState, outCreation->t.size, creationBuffer);
356         CryptCompleteHash2B(&hashState, &creationDigest->b);
357
358         return;
359     }
```

7.6.6 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

```
360     static UINT16
361     GetIV2BSize(
362         TPM_HANDLE          protectorHandle         // IN: the protector handle
363     )
364     {
365         OBJECT              *protector = NULL; // Pointer to the protector object
366         TPM_ALG_ID          symAlg;
367         UINT16              keyBits;
368
369         // Determine the symmetric algorithm and size of key
370         if(protectorHandle == TPM_RH_NULL)
371         {
372             // Use the context encryption algorithm and key size
373             symAlg = CONTEXT_ENCRYPT_ALG;
374             keyBits = CONTEXT_ENCRYPT_KEY_BITS;
375         }
376         else
377         {
378             protector = ObjectGet(protectorHandle);
379             symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
380             keyBits= protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
381         }
382
383         // The IV size is a UINT16 size field plus the block size of the symmetric
384         // algorithm
385         return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
386     }
```

7.6.7 GetSeedForKDF()

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

```
387     TPM2B_SEED*
388     GetSeedForKDF(
389         TPM_HANDLE           protectorHandle,    // IN: the protector handle
390         TPM2B_SEED          *seedIn              // IN: the optional input seed
391         )
392     {
393         OBJECT              *protector = NULL; // Pointer to the protector
394
395         // Get seed for encryption key.  Use input seed if provided.
396         // Otherwise, using protector object's seedValue.  TPM_RH_NULL is the only
397         // exception that we may not have a loaded object as protector.  In such a
398         // case, use nullProof as seed.
399         if(seedIn != NULL)
```

```
400         {
401             return seedIn;
402         }
403         else
404         {
405             if(protectorHandle == TPM_RH_NULL)
406             {
407                 return (TPM2B_SEED *) &gr.nullProof;
408             }
409             else
410             {
411                 protector = ObjectGet(protectorHandle);
412                 return (TPM2B_SEED *) &protector->sensitive.seedValue;
413             }
414         }
415     }
```

7.6.8 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

```
416     static void
417     ComputeProtectionKeyParms(
418         TPM_HANDLE          protectorHandle,    // IN: the protector handle
419         TPM_ALG_ID          hashAlg,            // IN: hash algorithm for KDFa
420         TPM2B_NAME          *name,              // IN: name of the object
421         TPM2B_SEED          *seedIn,            // IN: optional seed for duplication
422                                                 //     blob.  For non duplication blob,
423                                                 //     this parameter should be NULL
424         TPM_ALG_ID          *symAlg,            // OUT: the symmetric algorithm
425         UINT16              *keyBits,           // OUT: the symmetric key size in bits
426         TPM2B_SYM_KEY       *symKey             // OUT: the symmetric key
427     )
428     {
429         TPM2B_SEED          *seed = NULL;
430         OBJECT              *protector = NULL; // Pointer to the protector
431
432         // Determine the algorithms for the KDF and the encryption/decryption
433         // For TPM_RH_NULL, using context settings
434         if(protectorHandle == TPM_RH_NULL)
435         {
436             // Use the context encryption algorithm and key size
437             *symAlg = CONTEXT_ENCRYPT_ALG;
438             symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
439             *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
440         }
441         else
442         {
443             TPMT_SYM_DEF_OBJECT *symDef;
444             protector = ObjectGet(protectorHandle);
445             symDef = &protector->publicArea.parameters.asymDetail.symmetric;
446             *symAlg = symDef->algorithm;
447             *keyBits= symDef->keyBits.sym;
448             symKey->t.size = (*keyBits + 7) / 8;
449         }
450
451         // Get seed for KDF
452         seed = GetSeedForKDF(protectorHandle, seedIn);
453
454         // KDFa to generate symmetric key and IV value
455         KDFa(hashAlg, (TPM2B *)seed, "STORAGE", (TPM2B *)name, NULL,
```

```
456             symKey->t.size * 8, symKey->t.buffer, NULL);
457
458     return;
459 }
```

7.6.9 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.

```
460 static void
461 ComputeOuterIntegrity(
462     TPM2B_NAME          *name,              // IN: the name of the object
463     TPM_HANDLE           protectorHandle,   // IN: The handle of the object
464                                             //     that provides protection.  For
465                                             //     object, it is parent handle.
466                                             //     For credential, it is the handle
467                                             //     of encrypt object.  For a
468                                             //     Temporary Object, it is
469                                             //     TPM_RH_NULL
470     TPMI_ALG_HASH        hashAlg,           // IN: algorithm to use for integrity
471     TPM2B_SEED          *seedIn,            // IN: an external seed may be
472                                             //     provided for duplication blob.
473                                             //     For non duplication blob, this
474                                             //     parameter should be NULL
475     UINT32               sensitiveSize,     // IN: size of the marshaled sensitive
476                                             //     data
477     BYTE                *sensitiveData,     // IN: sensitive area
478     TPM2B_DIGEST        *integrity          // OUT: integrity
479 )
480 {
481     HMAC_STATE          hmacState;
482
483     TPM2B_DIGEST        hmacKey;
484     TPM2B_SEED          *seed = NULL;
485
486     // Get seed for KDF
487     seed = GetSeedForKDF(protectorHandle, seedIn);
488
489     // Determine the HMAC key bits
490     hmacKey.t.size = CryptGetHashDigestSize(hashAlg);
491
492     // KDFa to generate HMAC key
493     KDFa(hashAlg, (TPM2B *)seed, "INTEGRITY", NULL, NULL,
494          hmacKey.t.size * 8, hmacKey.t.buffer, NULL);
495
496     // Start HMAC and get the size of the digest which will become the integrity
497     integrity->t.size = CryptStartHMAC2B(hashAlg, &hmacKey.b, &hmacState);
498
499     // Adding the marshaled sensitive area to the integrity value
500     CryptUpdateDigest(&hmacState, sensitiveSize, sensitiveData);
501
502     // Adding name
503     CryptUpdateDigest2B(&hmacState, (TPM2B *)name);
504
505     // Compute HMAC
506     CryptCompleteHMAC2B(&hmacState, &integrity->b);
507
508     return;
509 }
```

7.6.10 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap

```
510  static void
511  ComputeInnerIntegrity(
512      TPM_ALG_ID              hashAlg,        // IN: hash algorithm for inner wrap
513      TPM2B_NAME              *name,          // IN: the name of the object
514      UINT16                  dataSize,       // IN: the size of sensitive data
515      BYTE                    *sensitiveData, // IN: sensitive data
516      TPM2B_DIGEST            *integrity      // OUT: inner integrity
517  )
518  {
519      HASH_STATE      hashState;
520
521      // Start hash and get the size of the digest which will become the integrity
522      integrity->t.size = CryptStartHash(hashAlg, &hashState);
523
524      // Adding the marshaled sensitive area to the integrity value
525      CryptUpdateDigest(&hashState, dataSize, sensitiveData);
526
527      // Adding name
528      CryptUpdateDigest2B(&hashState, &name->b);
529
530      // Compute hash
531      CryptCompleteHash2B(&hashState, &integrity->b);
532
533      return;
534
535  }
```

7.6.11 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

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

```
561
562         // Add integrity at the beginning of inner buffer
563         buffer = innerBuffer;
564         TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
565
566         return dataSize + integritySize;
567     }
```

7.6.12 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

```
568     static TPM_RC
569     CheckInnerIntegrity(
570         TPM2B_NAME              *name,          // IN: the name of the object
571         TPM_ALG_ID              hashAlg,        // IN: hash algorithm for inner wrap
572         UINT16                  dataSize,       // IN: the size of sensitive data,
573                                                 //     including the leading integrity
574                                                 //     buffer size
575         BYTE                    *innerBuffer    // IN/OUT: inner buffer with
576                                                 //     sensitive data in it
577     )
578     {
579         TPM_RC          result;
580
581         TPM2B_DIGEST    integrity;
582         TPM2B_DIGEST    integrityToCompare;
583         BYTE            *buffer;                // Auxiliary buffer pointer
584         INT32           size;
585
586         // Unmarshal integrity
587         buffer = innerBuffer;
588         size = (INT32) dataSize;
589         result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
590         if(result == TPM_RC_SUCCESS)
591         {
592             // Compute integrity to compare
593             ComputeInnerIntegrity(hashAlg, name, (UINT16) size, buffer,
594                                   &integrityToCompare);
595
596             // Compare outer blob integrity
597             if(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
598                 result = TPM_RC_INTEGRITY;
599         }
600         return result;
601     }
```

7.6.13 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

 Copyright © TCG 2006-2013

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

```
602 UINT16
603 ProduceOuterWrap(
604     TPM_HANDLE      protector,              // IN: The handle of the object
605                                            //     that provides protection.  For
606                                            //     object, it is parent handle.
607                                            //     For credential, it is the handle
608                                            //     of encrypt object.
609     TPM2B_NAME      *name,                 // IN: the name of the object
610     TPM_ALG_ID      hashAlg,               // IN: hash algorithm for outer wrap
611     TPM2B_SEED      *seed,                 // IN: an external seed may be
612                                            //     provided for duplication blob.
613                                            //     For non duplication blob, this
614                                            //     parameter should be NULL
615     BOOL            useIV,                 // IN: indicate if an IV is used
616     UINT16          dataSize,              // IN: the size of sensitive data,
617                                            //     excluding the leading integrity
618                                            //     buffer size or the optional iv
619                                            //     size
620     BYTE            *outerBuffer           // IN/OUT: outer buffer with
621                                            //         sensitive data in it
622 )
623 {
624     TPM_ALG_ID      symAlg;
625     UINT16          keyBits;
626     TPM2B_SYM_KEY   symKey;
627     TPM2B_IV        ivRNG;          // IV from RNG
628     TPM2B_IV        *iv = NULL;
629     UINT16          ivSize = 0;     // size of iv area, including the size field
630
631     BYTE            *sensitiveData; // pointer to the sensitive data
632
633     TPM2B_DIGEST    integrity;
634     UINT16          integritySize;
635     BYTE            *buffer;        // Auxiliary buffer pointer
636
637     // Compute the beginning of sensitive data.  The outer integrity should
638     // always exist if this function function is called to make an outer wrap
639     integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
640     sensitiveData = outerBuffer + integritySize;
641
642     // If iv is used, adjust the pointer of sensitive data and add iv before it
643     if(useIV)
644     {
645         ivSize = GetIV2BSize(protector);
646
647         // Generate IV from RNG.  The iv data size should be the total IV area
648         // size minus the size of size field
649         ivRNG.t.size = ivSize - sizeof(UINT16);
650         CryptGenerateRandom(ivRNG.t.size, ivRNG.t.buffer);
651
652         // Marshal IV to buffer
653         buffer = sensitiveData;
654         TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
655
656         // adjust sensitive data starting after IV area
657         sensitiveData += ivSize;
658
659         // Use iv for encryption
660         iv = &ivRNG;
661     }
662
663     // Compute symmetric key parameters for outer buffer encryption
664     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
```

```
665                                 &symAlg, &keyBits, &symKey);
666         // Encrypt inner buffer in place
667         CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
668                               TPM_ALG_CFB, symKey.t.buffer, iv, dataSize,
669                               sensitiveData);
670
671         // Compute outer integrity.  Integrity computation includes the optional IV
672         // area
673         ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
674                               outerBuffer + integritySize, &integrity);
675
676         // Add integrity at the beginning of outer buffer
677         buffer = outerBuffer;
678         TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
679
680         // return the total size in outer wrap
681         return dataSize + integritySize + ivSize;
682
683     }
```

7.6.14 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

```
684     TPM_RC
685     UnwrapOuter(
686         TPM_HANDLE          protector,              // IN: The handle of the object
687                                                     //     that provides protection.  For
688                                                     //     object, it is parent handle.
689                                                     //     For credential, it is the handle
690                                                     //     of encrypt object.
691         TPM2B_NAME          *name,                  // IN: the name of the object
692         TPM_ALG_ID           hashAlg,               // IN: hash algorithm for outer wrap
693         TPM2B_SEED          *seed,                  // IN: an external seed may be
694                                                     //     provided for duplication blob.
695                                                     //     For non duplication blob, this
696                                                     //     parameter should be NULL.
697         BOOL                 useIV,                 // IN: indicates if an IV is used
698         UINT16               dataSize,              // IN: size of sensitive data in
699                                                     //     outerBuffer, including the
700                                                     //     leading integrity buffer size,
701                                                     //     and an optional iv area
702         BYTE                *outerBuffer            // IN/OUT: sensitive data
703     )
704     {
705         TPM_RC              result;
706         TPM_ALG_ID          symAlg = TPM_ALG_NULL;
707         TPM2B_SYM_KEY       symKey;
708         UINT16              keyBits = 0;
709         TPM2B_IV            ivIn;               // input IV retrieved from input buffer
710         TPM2B_IV            *iv = NULL;
```

```
711
712         BYTE            *sensitiveData;     // pointer to the sensitive data
713
714         TPM2B_DIGEST    integrityToCompare;
715         TPM2B_DIGEST    integrity;
716         INT32           size;
717
718         // Unmarshal integrity
719         sensitiveData = outerBuffer;
720         size = (INT32) dataSize;
721         result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
722         if(result == TPM_RC_SUCCESS)
723         {
724             // Compute integrity to compare
725             ComputeOuterIntegrity(name, protector, hashAlg, seed,
726                                   (UINT16) size, sensitiveData,
727                                   &integrityToCompare);
728
729             // Compare outer blob integrity
730             if(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
731                 return TPM_RC_INTEGRITY;
732
733             // Get the symmetric algorithm parameters used for encryption
734             ComputeProtectionKeyParms(protector, hashAlg, name, seed,
735                                       &symAlg, &keyBits, &symKey);
736
737             // Retrieve IV if it is used
738             if(useIV)
739             {
740                 result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
741                 if(result == TPM_RC_SUCCESS)
742                 {
743                     // The input iv size for CFB must match the encryption algorithm
744                     // block size
745                     if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
746                         result = TPM_RC_VALUE;
747                     else
748                         iv = &ivIn;
749                 }
750             }
751         }
752         // If no errors, decrypt private in place
753         if(result == TPM_RC_SUCCESS)
754             CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
755                                   TPM_ALG_CFB, symKey.t.buffer, iv,
756                                   (UINT16) size, sensitiveData);
757
758         return result;
759
760     }
```

7.6.15 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.

```
761     void
762     SensitiveToPrivate(
763         TPMT_SENSITIVE          *sensitive,     // IN: sensitive structure
764         TPM2B_NAME              *name,          // IN: the name of the object
```

```
765         TPM_HANDLE              parentHandle,   // IN: The parent's handle
766         TPM_ALG_ID              nameAlg,        // IN: hash algorithm in public
767                                                 //     area.  This parameter is used
768                                                 //     when parentHandle is NULL, in
769                                                 //     which case the object is
770                                                 //     temporary.
771         TPM2B_PRIVATE           *outPrivate     // OUT: output private structure
772     )
773     {
774         BYTE                *buffer;            // Auxiliary buffer pointer
775         BYTE                *sensitiveData;     // pointer to the sensitive data
776         UINT16              dataSize;           // data blob size
777         TPMI_ALG_HASH       hashAlg;            // hash algorithm for integrity
778         UINT16              integritySize;
779         UINT16              ivSize;
780
781         pAssert(name != NULL && name->t.size != 0);
782
783         // Find the hash algorithm for integrity computation
784         if(parentHandle == TPM_RH_NULL)
785         {
786             // For Temporary Object, using self name algorithm
787             hashAlg = nameAlg;
788         }
789         else
790         {
791             // Otherwise, using parent's name algorithm
792             hashAlg = ObjectGetNameAlg(parentHandle);
793         }
794
795         // Starting of sensitive data without wrappers
796         sensitiveData = outPrivate->t.buffer;
797
798         // Compute the integrity size
799         integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
800
801         // Reserve space for integrity
802         sensitiveData += integritySize;
803
804         // Get iv size
805         ivSize = GetIV2BSize(parentHandle);
806
807         // Reserve space for iv
808         sensitiveData += ivSize;
809
810         // Marshal sensitive area, leaving the leading 2 bytes for size
811         buffer = sensitiveData + sizeof(UINT16);
812         dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
813
814         // Adding size before the data area
815         buffer = sensitiveData;
816         UINT16_Marshal(&dataSize, &buffer, NULL);
817
818         // Adjust the dataSize to include the size field
819         dataSize += sizeof(UINT16);
820
821         // Adjust the pointer to inner buffer including the iv
822         sensitiveData = outPrivate->t.buffer + ivSize;
823
824         //Produce outer wrap, including encryption and HMAC
825         outPrivate->t.size = ProduceOuterWrap(parentHandle, name, hashAlg, NULL,
826                                               TRUE, dataSize, outPrivate->t.buffer);
827
828         return;
829     }
```

DRAFT

7.6.16 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

```
830  TPM_RC
831  PrivateToSensitive(
832      TPM2B_PRIVATE           *inPrivate,     // IN: input private structure
833      TPM2B_NAME              *name,          // IN: the name of the object
834      TPM_HANDLE              parentHandle,   // IN: The parent's handle
835      TPM_ALG_ID              nameAlg,        // IN: hash algorithm in public
836                                              //     area.  It is passed separately
837                                              //     because we only pass name,
838                                              //     rather than the whole public
839                                              //     area of the object.  This
840                                              //     parameter is used in
841                                              //     the following two cases: 1.
842                                              //     primary objects. 2. duplication
843                                              //     blob with inner wrap.  In other
844                                              //     cases, this parameter will be
845                                              //     ignored
846      TPMT_SENSITIVE          *sensitive      // OUT: sensitive structure
847  )
848  {
849      TPM_RC          result;
850
851      BYTE            *buffer;
852      INT32           size;
853      BYTE            *sensitiveData; // pointer to the sensitive data
854      UINT16          dataSize;
855      UINT16          dataSizeInput;
856      TPMI_ALG_HASH   hashAlg;        // hash algorithm for integrity
857      OBJECT          *parent = NULL;
858
859      UINT16          integritySize;
860      UINT16          ivSize;
861
862      // Make sure that name is provided
863      pAssert(name != NULL && name->t.size != 0);
864
865      // Find the hash algorithm for integrity computation
866      if(parentHandle == TPM_RH_NULL)
867      {
868          // For Temporary Object, using self name algorithm
869          hashAlg = nameAlg;
870      }
871      else
872      {
873          // Otherwise, using parent's name algorithm
874          hashAlg = ObjectGetNameAlg(parentHandle);
875      }
```

```
876
877         // unwrap outer
878         result = UnwrapOuter(parentHandle, name, hashAlg, NULL, TRUE,
879                              inPrivate->t.size, inPrivate->t.buffer);
880         if(result != TPM_RC_SUCCESS)
881             return result;
882
883         // Compute the inner integrity size.
884         integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
885
886         // Get iv size
887         ivSize = GetIV2BSize(parentHandle);
888
889         // The starting of sensitive data and data size without outer wrapper
890         sensitiveData = inPrivate->t.buffer + integritySize + ivSize;
891         dataSize = inPrivate->t.size - integritySize - ivSize;
892
893         // Unmarshal input data size
894         buffer = sensitiveData;
895         size = (INT32) dataSize;
896         result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
897         if(result == TPM_RC_SUCCESS)
898         {
899             if((dataSizeInput + sizeof(UINT16)) != dataSize)
900                 result = TPM_RC_SENSITIVE;
901             else
902             {
903                 // Unmarshal sensitive buffer to sensitive structure
904                 result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
905                 if(result != TPM_RC_SUCCESS || size != 0)
906                 {
907                     pAssert(    (parent == NULL)
908                             || parent->publicArea.objectAttributes.fixedTPM == CLEAR);
909                     result = TPM_RC_SENSITIVE;
910                 }
911                 else
912                 {
913                     // Always remove trailing zeros at load so that it is not necessary
914                     // to check
915                     // each time auth is checked.
916                     MemoryRemoveTrailingZeros(&(sensitive->authValue));
917                 }
918             }
919         }
920         return result;
921     }
```

7.6.17 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

```
922     void
923     SensitiveToDuplicate(
924         TPMT_SENSITIVE          *sensitive,     // IN: sensitive structure
925         TPM2B_NAME              *name,          // IN: the name of the object
926         TPM_HANDLE              parentHandle,   // IN: The new parent's handle
927         TPM_ALG_ID              nameAlg,        // IN: hash algorithm in public
928                                                 //     area.  It is passed separately
929                                                 //     because we only pass name,
```

```
930                                             //     rather than the whole public
931                                             //     area of the object.
932     TPM2B_SEED              *seed,          // IN: the external seed.
933                                             //     If external seed is provided
934                                             //     with size of 0, no outer wrap
935                                             //     should be applied to duplication
936                                             //     blob.
937     TPMT_SYM_DEF_OBJECT     *symDef,        // IN: Symmetric key definition.
938                                             //     If the symmetric key algorithm
939                                             //     is NULL, no inner wrap should be
940                                             //     applied
941     TPM2B_DATA              *innerSymKey,   // IN: a symmetric key may be
942                                             //     provided to encrypt the inner
943                                             //     wrap of a duplication blob.
944     TPM2B_PRIVATE           *outPrivate     // OUT: output private structure
945 )
946 {
947     BYTE            *buffer;        // Auxiliary buffer pointer
948     BYTE            *sensitiveData; // pointer to the sensitive data
949     TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;// The hash algorithm for outer wrap
950     TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;// The hash algorithm for inner wrap
951     UINT16          dataSize;       // data blob size
952     BOOL            doInnerWrap = FALSE;
953     BOOL            doOuterWrap = FALSE;
954
955     // Make sure that name is provided
956     pAssert(name != NULL && name->t.size != 0);
957
958     // Make sure symDef and innerSymKey are not NULL
959     pAssert(symDef != NULL && innerSymKey != NULL);
960
961     // Starting of sensitive data without wrappers
962     sensitiveData = outPrivate->t.buffer;
963
964     // Find out if inner wrap is required
965     if(symDef->algorithm != TPM_ALG_NULL)
966     {
967         doInnerWrap = TRUE;
968         // Use self nameAlg as inner hash algorithm
969         innerHash = nameAlg;
970         // Adjust sensitive data pointer
971         sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
972     }
973
974     // Find out if outer wrap is required
975     if(seed->t.size != 0)
976     {
977         doOuterWrap = TRUE;
978         // Use parent nameAlg as outer hash algorithm
979         outerHash = ObjectGetNameAlg(parentHandle);
980         // Adjust sensitive data pointer
981         sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
982     }
983
984     // Marshal sensitive area, leaving the leading 2 bytes for size
985     buffer = sensitiveData + sizeof(UINT16);
986     dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
987
988     // Adding size before the data area
989     buffer = sensitiveData;
990     UINT16_Marshal(&dataSize, &buffer, NULL);
991
992     // Adjust the dataSize to include the size field
993     dataSize += sizeof(UINT16);
994
995     // Apply inner wrap for duplication blob.  It includes both integrity and
```

```
    // encryption
    if(doInnerWrap)
    {
        BYTE            *innerBuffer = NULL;
        BOOL            symKeyInput = TRUE;
        innerBuffer = outPrivate->t.buffer;
        // Skip outer integrity space
        if(doOuterWrap)
            innerBuffer += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
        dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
                                         innerBuffer);

        // Generate inner encryption key if needed
        if(innerSymKey->t.size == 0)
        {
            innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
            CryptGenerateRandom(innerSymKey->t.size, innerSymKey->t.buffer);

            // TPM generates symmetric encryption.  Set the flag to FALSE
            symKeyInput = FALSE;
        }
        else
        {
            // assume the input key size should matches the symmetric definition
            pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);

        }

        // Encrypt inner buffer in place
        CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
                              symDef->keyBits.sym, TPM_ALG_CFB,
                              innerSymKey->t.buffer, NULL, dataSize,
                              innerBuffer);

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

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

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

    return;
}
```

7.6.18 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

```
1049  TPM_RC
1050  DuplicateToSensitive(
1051      TPM2B_PRIVATE           *inPrivate,     // IN: input private structure
1052      TPM2B_NAME              *name,          // IN: the name of the object
1053      TPM_HANDLE              parentHandle,   // IN: The parent's handle
1054      TPM_ALG_ID              nameAlg,        // IN: hash algorithm in public
1055                                              //     area.
1056      TPM2B_SEED              *seed,          // IN: an external seed may be
1057                                              //     provided.
1058                                              //     If external seed is provided
1059                                              //     with size of 0, no outer wrap
1060                                              //     is applied
1061      TPMT_SYM_DEF_OBJECT     *symDef,        // IN: Symmetric key definition.
1062                                              //     If the symmetric key algorithm
1063                                              //     is NULL, no inner wrap is
1064                                              //     applied
1065      TPM2B_DATA              *innerSymKey,   // IN: a symmetric key may be
1066                                              //     provided to decrypt the inner
1067                                              //     wrap of a duplication blob.
1068      TPMT_SENSITIVE          *sensitive      // OUT: sensitive structure
1069  )
1070  {
1071      TPM_RC          result;
1072
1073      BYTE            *buffer;
1074      INT32           size;
1075      BYTE            *sensitiveData; // pointer to the sensitive data
1076      UINT16          dataSize;
1077      UINT16          dataSizeInput;
1078
1079      // Make sure that name is provided
1080      pAssert(name != NULL && name->t.size != 0);
1081
1082      // Make sure symDef and innerSymKey are not NULL
1083      pAssert(symDef != NULL && innerSymKey != NULL);
1084
1085      // Starting of sensitive data
1086      sensitiveData = inPrivate->t.buffer;
1087      dataSize = inPrivate->t.size;
1088
1089      // Find out if inner wrap is applied
1090      if(seed->t.size != 0)
1091      {
1092          TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;
1093
1094          // Use parent nameAlg as outer hash algorithm
1095          outerHash = ObjectGetNameAlg(parentHandle);
1096          result = UnwrapOuter(parentHandle, name, outerHash, seed, FALSE,
1097                               dataSize, sensitiveData);
1098          if(result != TPM_RC_SUCCESS)
1099              return result;
1100
1101          // Adjust sensitive data pointer and size
1102          sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1103          dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1104      }
1105      // Find out if inner wrap is applied
1106      if(symDef->algorithm != TPM_ALG_NULL)
```

```
1107        {
1108            TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;
1109
1110            // assume the input key size should matches the symmetric definition
1111            pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1112
1113            // Decrypt inner buffer in place
1114            CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
1115                                  symDef->keyBits.sym, TPM_ALG_CFB,
1116                                  innerSymKey->t.buffer, NULL, dataSize,
1117                                  sensitiveData);
1118
1119            // Use self nameAlg as inner hash algorithm
1120            innerHash = nameAlg;
1121
1122            // Check inner integrity
1123            result = CheckInnerIntegrity(name, innerHash, dataSize, sensitiveData);
1124            if(result != TPM_RC_SUCCESS)
1125                return result;
1126
1127            // Adjust sensitive data pointer and size
1128            sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
1129            dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
1130        }
1131
1132        // Unmarshal input data size
1133        buffer = sensitiveData;
1134        size = (INT32) dataSize;
1135        result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1136        if(result == TPM_RC_SUCCESS)
1137        {
1138            if((dataSizeInput + sizeof(UINT16)) != dataSize)
1139                result = TPM_RC_SIZE;
1140            else
1141            {
1142                // Unmarshal sensitive buffer to sensitive structure
1143                result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1144                // if the results is OK make sure that all the data was unmarshaled
1145                if(result == TPM_RC_SUCCESS && size != 0)
1146                    result = TPM_RC_SIZE;
1147            }
1148        }
1149        // Always remove trailing zeros at load so that it is not necessary to check
1150        // each time auth is checked.
1151        if(result == TPM_RC_SUCCESS)
1152            MemoryRemoveTrailingZeros(&(sensitive->authValue));
1153        return result;
1154    }
```

7.6.19 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

```
1155    void
1156    SecretToCredential(
1157        TPM2B_DIGEST        *secret,        // IN: secret information
```


DRAFT

```
1158        TPM2B_NAME          *name,          // IN: the name of the object
1159        TPM2B_SEED          *seed,          // IN: an external seed.
1160        TPM_HANDLE           protector,     // IN: The protector's handle
1161        TPM2B_ID_OBJECT     *outIDObject    // OUT: output credential
1162    )
1163    {
1164        BYTE                *buffer;        // Auxiliary buffer pointer
1165        BYTE                *sensitiveData; // pointer to the sensitive data
1166        TPMI_ALG_HASH        outerHash;     // The hash algorithm for outer wrap
1167        UINT16               dataSize;      // data blob size
1168
1169        pAssert(secret != NULL && outIDObject != NULL);
1170
1171        // use protector's name algorithm as outer hash
1172        outerHash = ObjectGetNameAlg(protector);
1173
1174        // Marshal secret area to credential buffer, leave space for integrity
1175        sensitiveData = outIDObject->t.credential
1176                      + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1177
1178        // Marshal secret area
1179        buffer = sensitiveData;
1180        dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1181
1182        // Apply outer wrap
1183        outIDObject->t.size = ProduceOuterWrap(protector,
1184                                               name,
1185                                               outerHash,
1186                                               seed,
1187                                               FALSE,
1188                                               dataSize,
1189                                               outIDObject->t.credential);
1190        return;
1191    }
```

7.6.20 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

```
1192    TPM_RC
1193    CredentialToSecret(
1194        TPM2B_ID_OBJECT         *inIDObject,    // IN: input credential blob
1195        TPM2B_NAME              *name,          // IN: the name of the object
1196        TPM2B_SEED              *seed,          // IN: an external seed.
1197        TPM_HANDLE               protector,     // IN: The protector's handle
1198        TPM2B_DIGEST            *secret         // OUT: secret information
1199    )
1200    {
1201        TPM_RC                   result;
```

```
1202        BYTE                    *buffer;
1203        INT32                    size;
1204        TPMI_ALG_HASH            outerHash;     // The hash algorithm for outer wrap
1205        BYTE                    *sensitiveData; // pointer to the sensitive data
1206        UINT16                   dataSize;
1207
1208        // use protector's name algorithm as outer hash
1209        outerHash = ObjectGetNameAlg(protector);
1210
1211        // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1212        result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
1213                             inIDObject->t.size, inIDObject->t.credential);
1214        if(result == TPM_RC_SUCCESS)
1215        {
1216            // Compute the beginning of sensitive data
1217            sensitiveData = inIDObject->t.credential
1218                          + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
1219            dataSize = inIDObject->t.size
1220                     - (sizeof(UINT16) + CryptGetHashDigestSize(outerHash));
1221
1222            // Unmarshal secret buffer to TPM2B_DIGEST structure
1223            buffer = sensitiveData;
1224            size = (INT32) dataSize;
1225            result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1226            // If there were no other unmarshaling errors, make sure that the
1227            // expected amount of data was recovered
1228            if(result == TPM_RC_SUCCESS && size != 0)
1229                return TPM_RC_SIZE;
1230        }
1231        return result;
1232    }
```

DRAFT

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
)
{
    if(type == SU_RESET)
```

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

8.1.3.3 CommandAuditSet()

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

This function is only used by TPM2_SetCommandCodeAuditStatus().

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

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

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

8.1.3.4 CommandAuditClear()

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

This function is only used by TPM2_SetCommandCodeAuditStatus().

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

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

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

8.1.3.5 CommandAuditIsRequired()

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

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

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

8.1.3.6 CommandAuditCapGetCCList()

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

The list starts at the input *commandCode*.

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

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

8.1.3.7 CommandAuditGetDigest

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

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

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

8.2 DA.c

8.2.1 Introduction

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

8.2.2 Includes and Data Definitions

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

8.2.2.1 DAPreInstall_Init

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

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

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

```
void
DAPreInstall_Init(void)
{
    gp.failedTries = 0;
    gp.maxTries = 3;
    gp.recoveryTime = 1000;         // in seconds (~16.67 minutes)
    gp.lockoutRecovery = 1000;      // in seconds
    gp.lockOutAuthEnabled = TRUE;   // Use of lockoutAuth is enabled

    // Record persistent DA parameter changes to NV
    NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
    NvWriteReserved(NV_MAX_TRIES, &gp.maxTries);
    NvWriteReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
    NvWriteReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
    NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);

    return;
}
```

8.2.2.2 DAStartup()

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

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

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

```
34          }
35      }
36
37      // If DA has not been disabled and the previous shutdown is not orderly
38      // failedTries is not already at its maximum then increment 'failedTries'
39      if(    gp.recoveryTime != 0
40          && g_prevOrderlyState == SHUTDOWN_NONE
41          && gp.failedTries < gp.maxTries)
42      {
43          gp.failedTries++;
44          // Record the change to NV
45          NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
46      }
47
48      // Reset self healing timers
49      s_selfHealTimer = g_time;
50      s_lockoutTimer = g_time;
51
52      return;
53  }
```

8.2.2.3 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.

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

8.2.2.4 DASelfHeal()

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

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

```
67  void
68  DASelfHeal(void)
69  {
70      // Regular auth self healing logic
71      // If no failed authorization tries, do nothing.  Otherwise, try to
72      // decrease failedTries
73      if(gp.failedTries != 0)
74      {
75          // if recovery time is 0, DA logic has been disabled.  Clear failed tries
76          // immediately
77          if(gp.recoveryTime == 0)
78          {
79              gp.failedTries = 0;
80              // Update NV record
81              NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
```

```
82          }
83          else
84          {
85              UINT64          decreaseCount;
86
87              // In the unlikely event that failedTries should become larger than
88              // maxTries
89              if(gp.failedTries > gp.maxTries)
90                  gp.failedTries = gp.maxTries;
91
92              // How much can failedTried be decreased
93              decreaseCount = ((g_time - s_selfHealTimer) / 1000) / gp.recoveryTime;
94
95              if(gp.failedTries <= (UINT32) decreaseCount)
96                  // should not set failedTries below zero
97                  gp.failedTries = 0;
98              else
99                  gp.failedTries -= (UINT32) decreaseCount;
100
101             // the cast prevents overflow of the product
102             s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
103             if(decreaseCount != 0)
104                 // If there was a change to the failedTries, record the changes
105                 // to NV
106                 NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
107         }
108     }
109
110
111     // LockoutAuth self healing logic
112     // If lockoutAuth is enabled, do nothing.  Otherwise, try to see if we
113     // may enable it
114     if(!gp.lockOutAuthEnabled)
115     {
116         // if lockout authorization recovery time is 0, a reboot is required to
117         // re-enable use of lockout authorization.  Self-healing would not
118         // apply in this case.
119         if(gp.lockoutRecovery != 0)
120         {
121             if(((g_time - s_lockoutTimer)/1000) >= gp.lockoutRecovery)
122             {
123                 gp.lockOutAuthEnabled = TRUE;
124                 // Record the changes to NV
125                 NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
126             }
127         }
128     }
129
130     return;
131 }
```

DRAFT

8.3 Hierarchy.c

8.3.1 Introduction

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

8.3.2 Includes

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

8.3.2.1 HierarchyPreInstall()

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

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

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

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

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

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

    // Write hierarchy data to NV
    NvWriteReserved(NV_DISABLE_CLEAR, &gp.disableClear);
    NvWriteReserved(NV_EP_SEED, &gp.EPSeed);
    NvWriteReserved(NV_SP_SEED, &gp.SPSeed);
    NvWriteReserved(NV_PP_SEED, &gp.PPSeed);
    NvWriteReserved(NV_OWNER_AUTH, &gp.ownerAuth);
    NvWriteReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
    NvWriteReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
    NvWriteReserved(NV_OWNER_ALG, &gp.ownerAlg);
    NvWriteReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
    NvWriteReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
    NvWriteReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
```

```
47      NvWriteReserved(NV_PH_PROOF, &gp.phProof);
48      NvWriteReserved(NV_SH_PROOF, &gp.shProof);
49      NvWriteReserved(NV_EH_PROOF, &gp.ehProof);
50
51      return;
52  }
```

8.3.2.2 HierarchyStartup()

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

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

8.3.2.3 HierarchyGetProof()

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

```
84  TPM2B_AUTH *
85  HierarchyGetProof(
86      TPMI_RH_HIERARCHY           hierarchy       // IN: hierarchy constant
87  )
88  {
89      TPM2B_AUTH          *auth = NULL;
90
91      switch(hierarchy)
92      {
93      case TPM_RH_PLATFORM:
94          // phProof for TPM_RH_PLATFORM
95          auth = &gp.phProof;
96          break;
97      case TPM_RH_ENDORSEMENT:
98          // ehProof for TPM_RH_ENDORSEMENT
99          auth = &gp.ehProof;
```

```
100             break;
101         case TPM_RH_OWNER:
102             // shProof for TPM_RH_OWNER
103             auth = &gp.shProof;
104             break;
105         case TPM_RH_NULL:
106             // nullProof for TPM_RH_NULL
107             auth = &gr.nullProof;
108             break;
109         default:
110             pAssert(FALSE);
111             break;
112         }
113         return auth;
114
115     }
```

8.3.2.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```
116     TPM2B_SEED *
117     HierarchyGetPrimarySeed(
118         TPMI_RH_HIERARCHY   hierarchy       // IN: hierarchy
119     )
120     {
121         TPM2B_SEED          *seed = NULL;
122         switch(hierarchy)
123         {
124         case TPM_RH_PLATFORM:
125             seed = &gp.PPSeed;
126             break;
127         case TPM_RH_OWNER:
128             seed = &gp.SPSeed;
129             break;
130         case TPM_RH_ENDORSEMENT:
131             seed = &gp.EPSeed;
132             break;
133         case TPM_RH_NULL:
134             return &gr.nullSeed;
135         default:
136             pAssert(FALSE);
137             break;
138         }
139         return seed;
140     }
```

8.3.2.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE: The TPM_RH_NULL hierarchy is always enabled.

Return Value	Meaning
TRUE	hierarchy is enabled
FALSE	hierarchy is disabled

```
141     BOOL
142     HierarchyIsEnabled(
143         TPMI_RH_HIERARCHY   hierarchy       // IN: hierarchy
```

```
144 )
145 {
146     BOOL            enabled = FALSE;
147 
148     switch(hierarchy)
149     {
150     case TPM_RH_PLATFORM:
151         enabled = g_phEnable;
152         break;
153     case TPM_RH_OWNER:
154         enabled = gc.shEnable;
155         break;
156     case TPM_RH_ENDORSEMENT:
157         enabled = gc.ehEnable;
158         break;
159     case TPM_RH_NULL:
160         enabled = TRUE;
161         break;
162     default:
163         pAssert(FALSE);
164         break;
165     }
166     return enabled;
167 }
```

DRAFT

8.4 NV.c

8.4.1 Introduction

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

8.4.2 Includes, Defines and Data Definitions

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

NV Index/evict object iterator value

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

8.4.3 NV Utility Functions

8.4.3.1 NvCheckState()

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

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

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

8.4.3.2 NvIsAvailable()

This function returns the NV availability parameter.

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

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

8.4.3.3 NvCommit

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

```
33  BOOL
34  NvCommit(void)
35  {
36      BOOL    success = (_plat__NvCommit() == 0);
37      return success;
38  }
```

8.4.3.4 NvReadMaxCount()

This function returns the max NV counter value.

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

8.4.3.5 NvWriteMaxCount()

This function updates the max counter value to NV memory.

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

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.

```
54  static UINT32
55  NvNext(
56      NV_ITER         *iter
57  )
58  {
59      NV_ITER     currentIter;
60
61      // If iterator is at the beginning of list
62      if(*iter == NV_ITER_INIT)
63      {
64          // Initialize iterator
65          *iter = s_evictNvStart;
66      }
67
68      // If iterator reaches the end of NV space, or iterator indicates list end
69      if(*iter + sizeof(UINT32) > s_evictNvEnd || *iter == 0)
70          return 0;
71
72      // Save the current iter offset
73      currentIter = *iter;
74
75      // Adjust iter pointer pointing to next entity
76      // Read pointer value
77      _plat__NvMemoryRead(*iter, sizeof(UINT32), iter);
78
79      if(*iter == 0) return 0;
80
81      return currentIter + sizeof(UINT32);      // entity stores after the pointer
82  }
```

8.4.4.3 NvGetEnd()

Function to find the end of the NV dynamic data list

```
83  static UINT32
84  NvGetEnd(void)
85  {
86      NV_ITER         iter = NV_ITER_INIT;
87      UINT32          endAddr = s_evictNvStart;
88      UINT32          currentAddr;
89
90      while((currentAddr = NvNext(&iter)) != 0)
91          endAddr = currentAddr;
92
93      if(endAddr != s_evictNvStart)
94      {
95          // Read offset
96          endAddr -= sizeof(UINT32);
97          _plat__NvMemoryRead(endAddr, sizeof(UINT32), &endAddr);
```

```
98          }
99
100         return endAddr;
101     }
```

8.4.4.4 NvGetFreeByte

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

```
102     static UINT32
103     NvGetFreeByte(void)
104     {
105         return s_evictNvEnd - NvGetEnd();
106     }
```

8.4.4.5 NvGetEvictObjectSize

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

```
107     static UINT32
108     NvGetEvictObjectSize(void)
109     {
110         return sizeof(TPM_HANDLE) + sizeof(OBJECT) + sizeof(UINT32);
111     }
```

8.4.4.6 NvGetCounterSize

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

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

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

```
119     static BOOL
120     NvTestSpace(
121         UINT32              size,           // IN: size of the entity to be added
122         BOOL                isIndex         // IN: TRUE if the entity is an index
123     )
124     {
125         UINT32      remainByte = NvGetFreeByte();
126
127         // For NV Index, need to make sure that we do not allocate and Index if this
128         // would mean that the TPM cannot allocate the minimum number of evict
129         // objects.
130         if(isIndex)
131         {
```

```
132             // Get the number of persistent objects allocated
133             UINT32      persistentNum = NvCapGetPersistentNumber();
134 
135             // If we have not allocated the requisite number of evict objects, then we
136             // need to reserve space for them.
137             // NOTE: some of this is not written as simply as it might seem because
138             // the values are all unsigned and subtracting needs to be done carefully
139             // so that an underflow doesn't cause problems.
140             if(persistentNum < MIN_EVICT_OBJECTS)
141             {
142                 UINT32      needed = (MIN_EVICT_OBJECTS - persistentNum)
143                                      * NvGetEvictObjectSize();
144                 if(needed > remainByte)
145                     remainByte = 0;
146                 else
147                     remainByte -= needed;
148             }
149             // if the requisite number of evict objects have been allocated then
150             // no need to reserve additional space
151         }
152         // This checks for the size of the value being added plus the index value.
153         // NOTE: This does not check to see if the end marker can be placed in
154         // memory because the end marker will not be written if it will not fit.
155         return (size + sizeof(UINT32) <= remainByte);
156     }
```

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).

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

```
188
189         // Set the flag so that NV changes are committed before the command completes.
190         g_updateNV = TRUE;
191     }
```

8.4.4.9 NvDelete()

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

```
192     static void
193     NvDelete(
194         UINT32          entityAddr              // IN: address of entity to be deleted
195     )
196     {
197         UINT32          next;
198         UINT32          entrySize;
199         UINT32          entryAddr = entityAddr - sizeof(UINT32);
200         UINT32          listEnd = 0;
201
202         // Get the offset of the next entry.
203         _plat__NvMemoryRead(entryAddr, sizeof(UINT32), &next);
204
205         // The size of this entry is the difference between the current entry and the
206         // next entry.
207         entrySize = next - entryAddr;
208
209         // Move each entry after the current one to fill the freed space.
210         // Stop when we have reached the end of all the indexes. There are two
211         // ways to detect the end of the list. The first is to notice that there
212         // is no room for anything else because we are at the end of NV. The other
213         // indication is that we find an end marker.
214
215         // The loop condition checks for the end of NV.
216         while(next + sizeof(UINT32) <= s_evictNvEnd)
217         {
218             UINT32      size, oldAddr, newAddr;
219
220             // Now check for the end marker
221             _plat__NvMemoryRead(next, sizeof(UINT32), &oldAddr);
222             if(oldAddr == 0)
223                 break;
224
225             size = oldAddr - next;
226
227             // Move entry
228             _plat__NvMemoryMove(next, next - entrySize, size);
229
230             // Update forward link
231             newAddr = oldAddr - entrySize;
232             _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &newAddr);
233             next = oldAddr;
234         }
235         // Mark the end of list
236         _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &listEnd);
237
238         // Set the flag so that NV changes are committed before the command completes.
239         g_updateNV = TRUE;
240     }
```

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

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

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.

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

```
276         pAssert(currAddr < s_ramIndexSize);
277         return currAddr + sizeof(TPMI_RH_NV_INDEX) + sizeof(UINT32);
278     }
```

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.

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

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

```
300     static void
301     NvDeleteRAM(
302         TPMI_RH_NV_INDEX            handle      // IN: NV handle
303     )
304     {
305         UINT32          nodeOffset;
306         UINT32          nextNode;
307         UINT32          size;
308
309         nodeOffset = NvGetRAMIndexOffset(handle);
310
311         // Move the pointer back to get the size field of this node
312         nodeOffset -= sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX);
313
314         // Get node size
315         size = * (UINT32 *) &s_ramIndex[nodeOffset];
316
317         // Get the offset of next node
318         nextNode = nodeOffset + sizeof(UINT32) + size;
319
320         // Move data
321         MemoryMove(s_ramIndex + nodeOffset, s_ramIndex + nextNode,
322                    s_ramIndexSize - nextNode, s_ramIndexSize - nextNode);
323
324         // Update RAM size
```

```
325         s_ramIndexSize -= size + sizeof(UINT32);
326
327         // Update NV version of s_ramIndexSize
328         _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
329
330         // Write reserved RAM space to NV to reflect the newly delete NV Index
331         _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
332
333         return;
334     }
```

8.4.6 Utility Functions

8.4.6.1 NvInitStatic()

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

```
335     static void
336     NvInitStatic(void)
337     {
338         UINT16      i;
339         UINT32      reservedAddr;
340
341         s_reservedSize[NV_DISABLE_CLEAR] = sizeof(gp.disableClear);
342         s_reservedSize[NV_OWNER_ALG] = sizeof(gp.ownerAlg);
343         s_reservedSize[NV_ENDORSEMENT_ALG] = sizeof(gp.endorsementAlg);
344         s_reservedSize[NV_OWNER_POLICY] = sizeof(gp.ownerPolicy);
345         s_reservedSize[NV_ENDORSEMENT_POLICY] = sizeof(gp.endorsementPolicy);
346         s_reservedSize[NV_OWNER_AUTH] = sizeof(gp.ownerAuth);
347         s_reservedSize[NV_ENDORSEMENT_AUTH] = sizeof(gp.endorsementAuth);
348         s_reservedSize[NV_LOCKOUT_AUTH] = sizeof(gp.lockoutAuth);
349         s_reservedSize[NV_EP_SEED] = sizeof(gp.EPSeed);
350         s_reservedSize[NV_SP_SEED] = sizeof(gp.SPSeed);
351         s_reservedSize[NV_PP_SEED] = sizeof(gp.PPSeed);
352         s_reservedSize[NV_PH_PROOF] = sizeof(gp.phProof);
353         s_reservedSize[NV_SH_PROOF] = sizeof(gp.shProof);
354         s_reservedSize[NV_EH_PROOF] = sizeof(gp.ehProof);
355         s_reservedSize[NV_TOTAL_RESET_COUNT] = sizeof(gp.totalResetCount);
356         s_reservedSize[NV_RESET_COUNT] = sizeof(gp.resetCount);
357         s_reservedSize[NV_PCR_POLICIES] = sizeof(gp.pcrPolicies);
358         s_reservedSize[NV_PCR_ALLOCATED] = sizeof(gp.pcrAllocated);
359         s_reservedSize[NV_PP_LIST] = sizeof(gp.ppList);
360         s_reservedSize[NV_FAILED_TRIES] = sizeof(gp.failedTries);
361         s_reservedSize[NV_MAX_TRIES] = sizeof(gp.maxTries);
362         s_reservedSize[NV_RECOVERY_TIME] = sizeof(gp.recoveryTime);
363         s_reservedSize[NV_LOCKOUT_RECOVERY] = sizeof(gp.lockoutRecovery);
364         s_reservedSize[NV_LOCKOUT_AUTH_ENABLED] = sizeof(gp.lockOutAuthEnabled);
365         s_reservedSize[NV_ORDERLY] = sizeof(gp.orderlyState);
366         s_reservedSize[NV_AUDIT_COMMANDS] = sizeof(gp.auditComands);
367         s_reservedSize[NV_AUDIT_HASH_ALG] = sizeof(gp.auditHashAlg);
368         s_reservedSize[NV_AUDIT_COUNTER] = sizeof(gp.auditCounter);
369         s_reservedSize[NV_ALGORITHM_SET] = sizeof(gp.algorithmSet);
370         s_reservedSize[NV_FIRMWARE_V1] = sizeof(gp.firmwareV1);
371         s_reservedSize[NV_FIRMWARE_V2] = sizeof(gp.firmwareV2);
372         s_reservedSize[NV_ORDERLY_DATA] = sizeof(go);
373         s_reservedSize[NV_STATE_CLEAR] = sizeof(gc);
374         s_reservedSize[NV_STATE_RESET] = sizeof(gr);
375
376         // Initialize reserved data address.  In this implementation, reserved data
377         // is stored at the start of NV memory
378         reservedAddr = 0;
379         for(i = 0; i < NV_RESERVE_LAST; i++)
380         {
```

```
381             s_reservedAddr[i] = reservedAddr;
382             reservedAddr += s_reservedSize[i];
383         }
384
385         // Initialize auxiliary variable space for index/evict implementation.
386         // Auxiliary variables are stored after reserved data area
387         // RAM index copy starts at the beginning
388         s_ramIndexSizeAddr = reservedAddr;
389         s_ramIndexAddr = s_ramIndexSizeAddr + sizeof(UINT32);
390
391         // Maximum counter value
392         s_maxCountAddr = s_ramIndexAddr + RAM_INDEX_SPACE;
393
394         // dynamic memory start
395         s_evictNvStart = s_maxCountAddr + sizeof(UINT64);
396
397         // dynamic memory ends that the end of NV memory
398         s_evictNvEnd = NV_MEMORY_SIZE;
399
400         return;
401     }
```

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.

```
402     void
403     NvInit(void)
404     {
405         UINT32      nullPointer = 0;
406         UINT64      zeroCounter = 0;
407
408         // Initialize static variables
409         NvInitStatic();
410
411         // Initialize RAM index space as un-used
412         _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &nullPointer);
413
414         // Initialize max counter value to 0
415         _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &zeroCounter);
416
417         // Initialize the next offset of the first entry in evict/index list to 0
418         _plat__NvMemoryWrite(s_evictNvStart, sizeof(TPM_HANDLE), &nullPointer);
419
420         return;
421
422     }
```

8.4.6.3 NvReadReserved()

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

```
423     void
424     NvReadReserved(
425         NV_RESERVE          type,                   // IN: type of reserved data
426         void                *buffer                 // OUT: buffer receives the
427         // data.
428     )
429     {
```

```
430         // Input type should be valid
431         pAssert(type >= 0 && type < NV_RESERVE_LAST);
432 
433         _plat__NvMemoryRead(s_reservedAddr[type], s_reservedSize[type], buffer);
434         return;
435     }
```

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.

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

8.4.6.5 NvReadPersistent()

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

```
451     void
452     NvReadPersistent(void)
453     {
454         // Hierarchy persistent data
455         NvReadReserved(NV_DISABLE_CLEAR, &gp.disableClear);
456         NvReadReserved(NV_OWNER_ALG, &gp.ownerAlg);
457         NvReadReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
458         NvReadReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
459         NvReadReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
460         NvReadReserved(NV_OWNER_AUTH, &gp.ownerAuth);
461         NvReadReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
462         NvReadReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
463         NvReadReserved(NV_EP_SEED, &gp.EPSeed);
464         NvReadReserved(NV_SP_SEED, &gp.SPSeed);
465         NvReadReserved(NV_PP_SEED, &gp.PPSeed);
466         NvReadReserved(NV_PH_PROOF, &gp.phProof);
467         NvReadReserved(NV_SH_PROOF, &gp.shProof);
468         NvReadReserved(NV_EH_PROOF, &gp.ehProof);
469 
470         // Time persistent data
471         NvReadReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
472         NvReadReserved(NV_RESET_COUNT, &gp.resetCount);
473 
474         // PCR persistent data
475         NvReadReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
476         NvReadReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
477 
478         // Physical Presence persistent data
479         NvReadReserved(NV_PP_LIST, &gp.ppList);
480 
481         // Dictionary attack values persistent data
```

```
482         NvReadReserved(NV_FAILED_TRIES, &gp.failedTries);
483         NvReadReserved(NV_MAX_TRIES, &gp.maxTries);
484         NvReadReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
485         NvReadReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
486         NvReadReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
487
488         // Orderly State persistent data
489         NvReadReserved(NV_ORDERLY, &gp.orderlyState);
490
491         // Command audit values persistent data
492         NvReadReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
493         NvReadReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
494         NvReadReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
495
496         // Algorithm selection persistent data
497         NvReadReserved(NV_ALGORITHM_SET, &gp.algorithmSet);
498
499         // Firmware version persistent data
500         NvReadReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
501         NvReadReserved(NV_FIRMWARE_V2, &gp.firmwareV2);
502
503         return;
504     }
```

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

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

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

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

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.

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

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.

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

8.4.6.10 NvFindHandle()

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

```
557  static UINT32
558  NvFindHandle(
559      TPM_HANDLE              handle
560  )
561  {
562      UINT32          addr;
563      NV_ITER         iter = NV_ITER_INIT;
564
```

```
565         while((addr = NvNext(&iter)) != 0)
566         {
567             TPM_HANDLE          entityHandle;
568             // Read handle
569             _plat__NvMemoryRead(addr, sizeof(TPM_HANDLE), &entityHandle);
570             if(entityHandle == handle)
571                 return addr;
572         }
573
574         pAssert(addr == 0);
575         return addr;
576     }
```

8.4.6.11 NvPowerOn()

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

```
577     void
578     NvPowerOn(void)
579     {
580         NvInitStatic();
581
582         return;
583     }
```

8.4.6.12 NvStateSave()

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

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

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.

```
597     void
598     NvEntityStartup(
599         STARTUP_TYPE        type        // IN: start up type
600     )
601     {
602         NV_ITER             iter = NV_ITER_INIT;
```

```
603         UINT32              currentAddr;        // offset points to the current entity
604
605         // Restore RAM index data
606         _plat__NvMemoryRead(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
607         _plat__NvMemoryRead(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);
608
609         // If recovering from state save, do nothing
610         if(type == SU_RESUME)
611             return;
612
613         // Iterate all the NV Index to clear the locks
614         while((currentAddr = NvNextIndex(&iter)) != 0)
615         {
616             NV_INDEX    nvIndex;
617             UINT32      indexAddr;              // NV address points to index info
618             indexAddr = currentAddr + sizeof(TPM_HANDLE);
619
620             // Read NV Index info structure
621             _plat__NvMemoryRead(indexAddr, sizeof(NV_INDEX), &nvIndex);
622
623             // Clear read/write lock
624             if(nvIndex.publicArea.attributes.TPMA_NV_READLOCKED == SET)
625                 nvIndex.publicArea.attributes.TPMA_NV_READLOCKED = CLEAR;
626             if(     nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED == SET
627                 &&  nvIndex.publicArea.attributes.TPMA_NV_WRITEDEFINE == CLEAR)
628                 nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED = CLEAR;
629
630             // Reset NV data for TPMA_NV_CLEAR_STCLEAR
631             if(nvIndex.publicArea.attributes.TPMA_NV_CLEAR_STCLEAR == SET)
632                 nvIndex.publicArea.attributes.TPMA_NV_WRITTEN = CLEAR;
633
634             // Reset NV data for orderly values that are not counters
635             // NOTE: The function has already exited on a TPM Resume, so the only
636             // things being processed are TPM Restart and TPM Reset
637             if(     type == SU_RESET
638                 &&  nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET
639                 &&  nvIndex.publicArea.attributes.TPMA_NV_COUNTER == CLEAR)
640                 nvIndex.publicArea.attributes.TPMA_NV_WRITTEN = CLEAR;
641
642             // Write NV Index info back
643             _plat__NvMemoryWrite(indexAddr, sizeof(NV_INDEX), &nvIndex);
644
645             // Set the flag that a NV write happens
646             g_updateNV = TRUE;
647
648             // Set the lower bits in an orderly counter to 1 for a non-orderly startup
649             if(   g_prevOrderlyState == SHUTDOWN_NONE
650                && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
651             {
652                 if(   nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET
653                    && nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET)
654                 {
655                     TPMI_RH_NV_INDEX    nvHandle;
656                     UINT64              counter;
657
658                     // Read NV handle
659                     _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);
660
661                     // Read the counter value saved to NV upon the last roll over.
662                     // Do not use RAM backed storage for this once.
663                     nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = CLEAR;
664                     NvGetIntIndexData(nvHandle, &nvIndex, &counter);
665                     nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = SET;
666
667                     // Set the lower bits of counter to 1
668                     counter |= MAX_ORDERLY_COUNT;
```

```
669
670                     // Write back to RAM
671                     NvWriteIndexData(nvHandle, &nvIndex, 0, 8, &counter);
672
673                     // No write to NV because an orderly shutdown will update the
674                     // counters.
675
676                 }
677             }
678         }
679
680         return;
681
682     }
```

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

```
683     BOOL
684     NvIsUndefinedIndex(
685         TPMI_RH_NV_INDEX handle      // IN: handle
686     )
687     {
688         UINT32          entityAddr;         // offset points to the entity
689
690         pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
691
692         // Find the address of index
693         entityAddr = NvFindHandle(handle);
694
695         // If handle is not found, return TPM_RC_SUCCESS
696         if(entityAddr == 0)
697             return TPM_RC_SUCCESS;
698
699         // NV Index is defined
700         return TPM_RC_NV_DEFINED;
701     }
```

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 platfrom auth

```
TPM_RC
NvIndexIsAccessible(
    TPMI_RH_NV_INDEX    handle          // IN: handle
)
{
    UINT32              entityAddr;     // offset points to the entity

    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);

    // Find the address of index
    entityAddr = NvFindHandle(handle);

    // If handle is not found, return TPM_RC_HANDLE
    if(entityAddr == 0)
        return TPM_RC_HANDLE;

    if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
    {
        NV_INDEX            nvIndex;

        // Read NV Index info structure
        _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
                            &nvIndex);

        // if shEnable is CLEAR, an ownerCreate NV Index should not be
        // indicated as present
        if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
        {
            if(gc.shEnable == FALSE)
                return TPM_RC_HANDLE;
        }
        // if phEnableNV is CLEAR, a plaform creaed Index should not
        // be visible
        else if(gc.phEnableNV == FALSE)
            return TPM_RC_HANDLE;
    }

    // NV Index is accessible
    return TPM_RC_SUCCESS;
}
```

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

```
static BOOL
NvIsUndefinedEvictHandle(
    TPM_HANDLE       handle         // IN: handle
)
{
```

```
747         UINT32          entityAddr;     // offset points to the entity
748         pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
749
750         // Find the address of evict object
751         entityAddr = NvFindHandle(handle);
752
753         // If handle is not found, return TRUE
754         if(entityAddr == 0)
755             return TRUE;
756         else
757             return FALSE;
758     }
```

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

```
759     TPM_RC
760     NvGetEvictObject(
761         TPM_HANDLE                handle,     // IN: handle
762         OBJECT                   *object      // OUT: object data
763     )
764     {
765         UINT32          entityAddr;         // offset points to the entity
766         TPM_RC          result = TPM_RC_SUCCESS;
767
768         pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
769
770         // Find the address of evict object
771         entityAddr = NvFindHandle(handle);
772
773         // If handle is not found, return an error
774         if(entityAddr == 0)
775             result = TPM_RC_HANDLE;
776         else
777             // Read evict object
778             _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE),
779                                 sizeof(OBJECT),
780                                 object);
781
782         // whether there is an error or not, make sure that the evict
783         // status of the object is set so that the slot will get freed on exit
784         object->attributes.evict = SET;
785
786         return result;
787     }
```

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.

```
788     void
789     NvGetIndexInfo(
```

```
790         TPMI_RH_NV_INDEX     handle,         // IN: handle
791         NV_INDEX            *nvIndex         // OUT: NV index structure
792     )
793     {
794         UINT32               entityAddr;     // offset points to the entity
795
796         pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
797
798         // Find the address of nv index
799         entityAddr = NvFindHandle(handle);
800         pAssert(entityAddr != 0);
801
802         // This implementation uses the default format so just
803         // read the data in
804         _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
805                             nvIndex);
806
807         return;
808     }
```

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.

```
809     UINT64
810     NvInitialCounter(void)
811     {
812         UINT64          maxCount;
813         NV_ITER         iter = NV_ITER_INIT;
814         UINT32          currentAddr;
815
816         // Read the maxCount value
817         maxCount = NvReadMaxCount();
818
819         // Iterate all existing counters
820         while((currentAddr = NvNextIndex(&iter)) != 0)
821         {
822             TPMI_RH_NV_INDEX    nvHandle;
823             NV_INDEX            nvIndex;
824
825             // Read NV handle
826             _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);
827
828             // Get NV Index
829             NvGetIndexInfo(nvHandle, &nvIndex);
830             if(    nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
831                 && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
832             {
833                 UINT64      countValue;
834                 // Read counter value
835                 NvGetIntIndexData(nvHandle, &nvIndex, &countValue);
836                 if(countValue > maxCount)
837                     maxCount = countValue;
838             }
839         }
840         // Initialize the new counter value to be maxCount + 1
841         // A counter is only initialized the first time it is written. The
842         // way to write a counter is with TPM2_NV_INCREMENT(). Since the
843         // "initial" value of a defined counter is the largest count value that
844         // may have existed in this index previously, then the first use would
845         // add one to that value.
846         return maxCount;
```

```
847 }
```

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.

```
848 void
849 NvGetIndexData(
850     TPMI_RH_NV_INDEX     handle,             // IN: handle
851     NV_INDEX            *nvIndex,            // IN: RAM image of index header
852     UINT32               offset,             // IN: offset of NV data
853     UINT16               size,               // IN: size of NV data
854     void                *data                // OUT: data buffer
855 )
856 {
857
858     pAssert(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET);
859
860     if(   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
861        || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET)
862     {
863         // Read bit or counter data in canonical form
864         UINT64      dataInInt;
865         NvGetIntIndexData(handle, nvIndex, &dataInInt);
866         UINT64_TO_BYTE_ARRAY(dataInInt, (BYTE *)data);
867     }
868     else
869     {
870         if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
871         {
872             UINT32      ramAddr;
873
874             // Get data from RAM buffer
875             ramAddr = NvGetRAMIndexOffset(handle);
876             MemoryCopy(data, s_ramIndex + ramAddr + offset, size, size);
877         }
878         else
879         {
880             UINT32      entityAddr;
881             entityAddr = NvFindHandle(handle);
882             // Get data from NV
883             // Skip NV Index info, read data buffer
884             entityAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
885             // Read the data
886             _plat__NvMemoryRead(entityAddr, size, data);
887         }
888     }
889     return;
890 }
```

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.

```
891 void
892 NvGetIntIndexData(
893     TPMI_RH_NV_INDEX     handle,        // IN: handle
```

```
894         NV_INDEX            *nvIndex,       // IN: RAM image of NV Index header
895         UINT64              *data           // IN: UINT64 pointer for counter or bits
896     )
897     {
898         // Validate that index has been written and is the right type
899         pAssert(   nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET
900                 && (   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
901                     || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET
902                    )
903                );
904
905         // bit and counter value is store in native format for TPM CPU.  So we directly
906         // copy the contents of NV to output data buffer
907         if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
908         {
909             UINT32      ramAddr;
910
911             // Get data from RAM buffer
912             ramAddr = NvGetRAMIndexOffset(handle);
913             MemoryCopy(data, s_ramIndex + ramAddr, sizeof(*data), sizeof(*data));
914         }
915         else
916         {
917             UINT32      entityAddr;
918             entityAddr = NvFindHandle(handle);
919
920             // Get data from NV
921             // Skip NV Index info, read data buffer
922             _plat__NvMemoryRead(
923                 entityAddr + sizeof(TPM_HANDLE) + sizeof(NV_INDEX),
924                 sizeof(UINT64), data);
925         }
926
927         return;
928     }
```

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

```
929     TPM_RC
930     NvWriteIndexInfo(
931         TPMI_RH_NV_INDEX     handle,    // IN: handle
932         NV_INDEX            *nvIndex    // IN: NV Index info to be written
933     )
934     {
935         UINT32      entryAddr;
936         TPM_RC      result;
937
938         // Get the starting offset for the index in the RAM image of NV
939         entryAddr = NvFindHandle(handle);
940         pAssert(entryAddr != 0);
941
942         // Step over the link value
943         entryAddr = entryAddr + sizeof(TPM_HANDLE);
944
945         // If the index data is actually changed, then a write to NV is required
```

```
946         if(_plat__NvIsDifferent(entryAddr, sizeof(NV_INDEX),nvIndex))
947         {
948             // Make sure that NV is available
949             result = NvIsAvailable();
950             if(result != TPM_RC_SUCCESS)
951                 return result;
952             _plat__NvMemoryWrite(entryAddr, sizeof(NV_INDEX), nvIndex);
953             g_updateNV = TRUE;
954         }
955         return TPM_RC_SUCCESS;
956     }
```

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

```
957     TPM_RC
958     NvWriteIndexData(
959         TPMI_RH_NV_INDEX     handle,     // IN: handle
960         NV_INDEX            *nvIndex,    // IN: RAM copy of NV Index
961         UINT32               offset,     // IN: offset of NV data
962         UINT32               size,       // IN: size of NV data
963         void                *data        // OUT: data buffer
964     )
965     {
966         TPM_RC               result;
967         // Validate that write falls within range of the index
968         pAssert(nvIndex->publicArea.dataSize >= offset + size);
969
970         // Update TPMA_NV_WRITTEN bit if necessary
971         if(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
972         {
973             nvIndex->publicArea.attributes.TPMA_NV_WRITTEN = SET;
974             result = NvWriteIndexInfo(handle, nvIndex);
975             if(result != TPM_RC_SUCCESS)
976                 return result;
977         }
978
979         // Check to see if process for an orderly index is required.
980         if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
981         {
982             UINT32      ramAddr;
983
984             // Write data to RAM buffer
985             ramAddr = NvGetRAMIndexOffset(handle);
986             MemoryCopy(s_ramIndex + ramAddr + offset, data, size,
987                        sizeof(s_ramIndex) - ramAddr - offset);
988
989             // NV update does not happen for orderly index.  Have
990             // to clear orderlyState to reflect that we have changed the
991             // NV and an orderly shutdown is required. Only going to do this if we
992             // are not processing a counter that has just rolled over
993             if(g_updateNV == FALSE)
994                 g_clearOrderly = TRUE;
995         }
```

```
996         // Need to process this part if the Index isn't orderly or if it is
997         // an orderly counter that just rolled over.
998         if(g_updateNV || nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == CLEAR)
999         {
1000            // Processing for an index with TPMA_NV_ORDERLY CLEAR
1001            UINT32      entryAddr = NvFindHandle(handle);
1002
1003            pAssert(entryAddr != 0);
1004
1005            // Offset into the index to the first byte of the data to be written
1006            entryAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
1007
1008            // If the data is actually changed, then a write to NV is required
1009            if(_plat__NvIsDifferent(entryAddr, size, data))
1010            {
1011                // Make sure that NV is available
1012                result = NvIsAvailable();
1013                if(result != TPM_RC_SUCCESS)
1014                    return result;
1015                _plat__NvMemoryWrite(entryAddr, size, data);
1016                g_updateNV = TRUE;
1017            }
1018        }
1019
1020        return TPM_RC_SUCCESS;
1021    }
```

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.

```
1022    UINT16
1023    NvGetName(
1024        TPMI_RH_NV_INDEX    handle,     // IN: handle of the index
1025        NAME               *name        // OUT: name of the index
1026    )
1027    {
1028        UINT16               dataSize, digestSize;
1029        NV_INDEX             nvIndex;
1030        BYTE                 marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
1031        BYTE                *buffer;
1032        HASH_STATE           hashState;
1033
1034        // Get NV public info
1035        NvGetIndexInfo(handle, &nvIndex);
1036
1037        // Marshal public area
1038        buffer = marshalBuffer;
1039        dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex.publicArea, &buffer, NULL);
1040
1041        // hash public area
1042        digestSize = CryptStartHash(nvIndex.publicArea.nameAlg, &hashState);
1043        CryptUpdateDigest(&hashState, dataSize, marshalBuffer);
1044
1045        // Complete digest leaving room for the nameAlg
1046        CryptCompleteHash(&hashState, digestSize, &((BYTE *)name)[2]);
1047
1048        // Include the nameAlg
1049        UINT16_TO_BYTE_ARRAY(nvIndex.publicArea.nameAlg, (BYTE *)name);
1050        return digestSize + 2;
```

```
1051    }
```

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

```
1052    TPM_RC
1053    NvDefineIndex(
1054        TPMS_NV_PUBLIC      *publicArea,     // IN: A template for an area to create.
1055        TPM2B_AUTH          *authValue       // IN: The initial authorization value
1056    )
1057    {
1058        // The buffer to be written to NV memory
1059        BYTE            nvBuffer[sizeof(TPM_HANDLE) + sizeof(NV_INDEX)];
1060
1061        NV_INDEX        *nvIndex;           // a pointer to the NV_INDEX data in
1062                                            //   nvBuffer
1063        UINT16          entrySize;          // size of entry
1064
1065        entrySize = sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + publicArea->dataSize;
1066
1067        // Check if we have enough space to create the NV Index
1068        // In this implementation, the only resource limitation is the available NV
1069        // space.  Other implementation may have other limitation on counter or on
1070        // NV slot
1071        if(!NvTestSpace(entrySize, TRUE)) return TPM_RC_NV_SPACE;
1072
1073        // if the index to be defined is RAM backed, check RAM space availability
1074        // as well
1075        if(publicArea->attributes.TPMA_NV_ORDERLY == SET
1076                && !NvTestRAMSpace(publicArea->dataSize))
1077            return TPM_RC_NV_SPACE;
1078
1079
1080        // Copy input value to nvBuffer
1081            // Copy handle
1082        * (TPM_HANDLE *) nvBuffer = publicArea->nvIndex;
1083
1084            // Copy NV_INDEX
1085        nvIndex = (NV_INDEX *) (nvBuffer + sizeof(TPM_HANDLE));
1086        nvIndex->publicArea = *publicArea;
1087        nvIndex->authValue = *authValue;
1088
1089        // Add index to NV memory
1090        NvAdd(entrySize, sizeof(TPM_HANDLE) + sizeof(NV_INDEX), nvBuffer);
1091
1092        // If the data of NV Index is RAM backed, add the data area in RAM as well
1093        if(publicArea->attributes.TPMA_NV_ORDERLY == SET)
1094            NvAddRAM(publicArea->nvIndex, publicArea->dataSize);
1095
1096        return TPM_RC_SUCCESS;
1097    }
```

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

```
1098  TPM_RC
1099  NvAddEvictObject(
1100      TPMI_DH_OBJECT       evictHandle,  // IN: new evict handle
1101      OBJECT              *object        // IN: object to be added
1102  )
1103  {
1104      // The buffer to be written to NV memory
1105      BYTE             nvBuffer[sizeof(TPM_HANDLE) + sizeof(OBJECT)];
1106
1107      OBJECT           *nvObject;          // a pointer to the OBJECT data in
1108                                           // nvBuffer
1109      UINT16           entrySize;          // size of entry
1110
1111      // evict handle type should match the object hierarchy
1112      pAssert(    (   NvIsPlatformPersistentHandle(evictHandle)
1113                   && object->attributes.ppsHierarchy == SET)
1114              || (   NvIsOwnerPersistentHandle(evictHandle)
1115                   && (   object->attributes.spsHierarchy == SET
1116                       || object->attributes.epsHierarchy == SET)));
1117
1118      // An evict needs 4 bytes of handle + sizeof OBJECT
1119      entrySize = sizeof(TPM_HANDLE) + sizeof(OBJECT);
1120
1121      // Check if we have enough space to add the evict object
1122      // An evict object needs 8 bytes in index table + sizeof OBJECT
1123      // In this implementation, the only resource limitation is the available NV
1124      // space.  Other implementation may have other limitation on evict object
1125      // handle space
1126      if(!NvTestSpace(entrySize, FALSE)) return TPM_RC_NV_SPACE;
1127
1128      // Allocate a new evict handle
1129      if(!NvIsUndefinedEvictHandle(evictHandle))
1130          return TPM_RC_NV_DEFINED;
1131
1132      // Copy evict object to nvBuffer
1133          // Copy handle
1134      * (TPM_HANDLE *) nvBuffer = evictHandle;
1135
1136          // Copy OBJECT
1137      nvObject = (OBJECT *) (nvBuffer + sizeof(TPM_HANDLE));
1138      *nvObject = *object;
1139
1140      // Set evict attribute and handle
1141      nvObject->attributes.evict = SET;
1142      nvObject->evictHandle = evictHandle;
1143
1144      // Add evict to NV memory
1145      NvAdd(entrySize, entrySize, nvBuffer);
1146
1147      return TPM_RC_SUCCESS;
1148
1149  }
```

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.

```
1150  void
1151  NvDeleteEntity(
1152      TPM_HANDLE    handle        // IN: handle of entity to be deleted
1153  )
1154  {
1155      UINT32      entityAddr;     // pointer to entity
1156
1157      entityAddr = NvFindHandle(handle);
1158      pAssert(entityAddr != 0);
1159
1160      if(HandleGetType(handle) == TPM_HT_NV_INDEX)
1161      {
1162          NV_INDEX    nvIndex;
1163
1164          // Read the NV Index info
1165          _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
1166                              &nvIndex);
1167
1168          // If the entity to be deleted is a counter with the maximum counter
1169          // value, record it in NV memory
1170          if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
1171                  && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
1172          {
1173              UINT64      countValue;
1174              UINT64      maxCount;
1175              NvGetIntIndexData(handle, &nvIndex, &countValue);
1176              maxCount = NvReadMaxCount();
1177              if(countValue > maxCount)
1178                  NvWriteMaxCount(countValue);
1179          }
1180          // If the NV Index is RAM back, delete the RAM data as well
1181          if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
1182              NvDeleteRAM(handle);
1183      }
1184      NvDelete(entityAddr);
1185
1186      return;
1187
1188  }
```

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*.

```
1189  void
1190  NvFlushHierarchy(
1191      TPMI_RH_HIERARCHY           hierarchy     // IN: hierarchy to be flushed.
1192  )
1193  {
1194      NV_ITER         iter = NV_ITER_INIT;
1195      UINT32          currentAddr;
1196
1197      while((currentAddr = NvNext(&iter)) != 0)
1198      {
1199          TPM_HANDLE      entityHandle;
1200
1201          // Read handle information.
1202          _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1203
1204          if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
1205          {
1206              // Handle NV Index
1207              NV_INDEX    nvIndex;
```

```
1208
1209            // If flush endorsement or platform hierarchy, no NV Index would be
1210            // flushed
1211            if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
1212                continue;
1213            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1214                                sizeof(NV_INDEX), &nvIndex);
1215
1216            // For storage hierarchy, flush OwnerCreated index
1217            if(  nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
1218            {
1219                // Delete the NV Index
1220                NvDelete(currentAddr);
1221
1222                // Re-iterate from beginning after a delete
1223                iter = NV_ITER_INIT;
1224
1225                // If the NV Index is RAM back, delete the RAM data as well
1226                if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
1227                    NvDeleteRAM(entityHandle);
1228            }
1229        }
1230        else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
1231        {
1232            OBJECT          object;
1233
1234            // Get evict object
1235            NvGetEvictObject(entityHandle, &object);
1236
1237            // If the evict object belongs to the hierarchy to be flushed
1238            if(   (    hierarchy == TPM_RH_PLATFORM
1239                   &&  object.attributes.ppsHierarchy == SET)
1240               || (    hierarchy == TPM_RH_OWNER
1241                   &&  object.attributes.spsHierarchy == SET)
1242               || (    hierarchy == TPM_RH_ENDORSEMENT
1243                   &&  object.attributes.epsHierarchy == SET)
1244               )
1245            {
1246                // Delete the evict object
1247                NvDelete(currentAddr);
1248
1249                // Re-iterate from beginning after a delete
1250                iter = NV_ITER_INIT;
1251            }
1252        }
1253        else
1254        {
1255            pAssert(FALSE);
1256        }
1257    }
1258
1259    return;
1260 }
```

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().

```
1261 void
1262 NvSetGlobalLock(void)
1263 {
1264     NV_ITER         iter = NV_ITER_INIT;
1265     UINT32          currentAddr;
```

```
1266
1267        // Check all indices
1268        while((currentAddr = NvNextIndex(&iter)) != 0)
1269        {
1270            NV_INDEX    nvIndex;
1271
1272            // Read the index data
1273            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1274                                sizeof(NV_INDEX), &nvIndex);
1275
1276            // See if it should be locked
1277            if(nvIndex.publicArea.attributes.TPMA_NV_GLOBALLOCK == SET)
1278            {
1279
1280                // if so, lock it
1281                nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED = SET;
1282
1283                _plat__NvMemoryWrite(currentAddr + sizeof(TPM_HANDLE),
1284                                     sizeof(NV_INDEX), &nvIndex);
1285                // Set the flag that a NV write happens
1286                g_updateNV = TRUE;
1287            }
1288        }
1289
1290        return;
1291
1292    }
```

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

```
1293    static void
1294    InsertSort(
1295        TPML_HANDLE         *handleList,            // IN/OUT: sorted handle list
1296        UINT32              count,                  // IN: maximum count in the handle
1297                                                    //     list
1298        TPM_HANDLE          entityHandle            // IN: handle to be inserted
1299    )
1300    {
1301        UINT32          i, j;
1302        UINT32          originalCount;
1303
1304        // For a corner case that the maximum count is 0, do nothing
1305        if(count == 0) return;
1306
1307        // For empty list, add the handle at the beginning and return
1308        if(handleList->count == 0)
1309        {
1310            handleList->handle[0] = entityHandle;
1311            handleList->count++;
1312            return;
1313        }
1314
1315        // Check if the maximum of the list has been reached
1316        originalCount = handleList->count;
1317        if(originalCount < count)
1318            handleList->count++;
1319
1320        // Insert the handle to the list
1321        for(i = 0; i < originalCount; i++)
1322        {
1323            if(handleList->handle[i] > entityHandle)
```

```
1324            {
1325                for(j = handleList->count - 1; j > i; j--)
1326                {
1327                    handleList->handle[j] = handleList->handle[j-1];
1328                }
1329                break;
1330            }
1331        }
1332
1333        // If a slot was found, insert the handle in this position
1334        if(i < originalCount || handleList->count > originalCount)
1335            handleList->handle[i] = entityHandle;
1336
1337        return;
1338    }
```

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

```
1339    TPMI_YES_NO
1340    NvCapGetPersistent(
1341        TPMI_DH_OBJECT       handle,         // IN: start handle
1342        UINT32               count,          // IN: maximum number of returned handles
1343        TPML_HANDLE         *handleList      // OUT: list of handle
1344    )
1345    {
1346        TPMI_YES_NO          more = NO;
1347        NV_ITER              iter = NV_ITER_INIT;
1348        UINT32               currentAddr;
1349
1350        pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1351
1352        // Initialize output handle list
1353        handleList->count = 0;
1354
1355        // The maximum count of handles we may return is MAX_CAP_HANDLES
1356        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1357
1358        while((currentAddr = NvNextEvict(&iter)) != 0)
1359        {
1360            TPM_HANDLE      entityHandle;
1361
1362            // Read handle information.
1363            _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1364
1365            // Ignore persistent handles that have values less than the input handle
1366            if(entityHandle < handle)
1367                continue;
1368
1369            // if the handles in the list have reached the requested count, and there
1370            // are still handles need to be inserted, indicate that there are more.
1371            if(handleList->count == count)
1372                more = YES;
1373
```

```
1374            // A handle with a value larger than start handle is a candidate
1375            // for return. Insert sort it to the return list.  Insert sort algorithm
1376            // is chosen here for simplicity based on the assumption that the total
1377            // number of NV Indices is small.  For an implementation that may allow
1378            // large number of NV Indices, a more efficient sorting algorithm may be
1379            // used here.
1380            InsertSort(handleList, count, entityHandle);
1381
1382        }
1383        return more;
1384    }
```

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

```
1385    TPMI_YES_NO
1386    NvCapGetIndex(
1387        TPMI_DH_OBJECT       handle,         // IN: start handle
1388        UINT32               count,          // IN: maximum number of returned handles
1389        TPML_HANDLE         *handleList      // OUT: list of handle
1390    )
1391    {
1392        TPMI_YES_NO          more = NO;
1393        NV_ITER              iter = NV_ITER_INIT;
1394        UINT32               currentAddr;
1395
1396        pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1397
1398        // Initialize output handle list
1399        handleList->count = 0;
1400
1401        // The maximum count of handles we may return is MAX_CAP_HANDLES
1402        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1403
1404        while((currentAddr = NvNextIndex(&iter)) != 0)
1405        {
1406            TPM_HANDLE      entityHandle;
1407
1408            // Read handle information.
1409            _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);
1410
1411            // Ignore index handles that have values less than the 'handle'
1412            if(entityHandle < handle)
1413                continue;
1414
1415            // if the count of handles in the list has reached the requested count,
1416            // and there are still handles to report, set more.
1417            if(handleList->count == count)
1418                more = YES;
1419
1420            // A handle with a value larger than start handle is a candidate
1421            // for return. Insert sort it to the return list.  Insert sort algorithm
1422            // is chosen here for simplicity based on the assumption that the total
1423            // number of NV Indices is small.  For an implementation that may allow
1424            // large number of NV Indices, a more efficient sorting algorithm may be
1425            // used here.
```

```
1426            InsertSort(handleList, count, entityHandle);
1427        }
1428        return more;
1429    }
```

8.4.7.21 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```
1430    UINT32
1431    NvCapGetIndexNumber(void)
1432    {
1433        UINT32          num = 0;
1434        NV_ITER         iter = NV_ITER_INIT;
1435
1436        while(NvNextIndex(&iter) != 0) num++;
1437
1438        return num;
1439    }
```

8.4.7.22 NvCapGetPersistentNumber()

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

```
1440    UINT32
1441    NvCapGetPersistentNumber(void)
1442    {
1443        UINT32          num = 0;
1444        NV_ITER         iter = NV_ITER_INIT;
1445
1446        while(NvNextEvict(&iter) != 0) num++;
1447
1448        return num;
1449    }
```

8.4.7.23 NvCapGetPersistentAvail()

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

```
1450    UINT32
1451    NvCapGetPersistentAvail(void)
1452    {
1453        UINT32          availSpace;
1454        UINT32          objectSpace;
1455
1456        // Compute the available space in NV storage
1457        availSpace = NvGetFreeByte();
1458
1459        // Get the space needed to add a persistent object to NV storage
1460        objectSpace = NvGetEvictObjectSize();
1461
1462        return availSpace / objectSpace;
1463    }
```

8.4.7.24 NvCapGetCounterNumber()

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

```
1464    UINT32
```

```
1465  NvCapGetCounterNumber(void)
1466  {
1467      NV_ITER         iter = NV_ITER_INIT;
1468      UINT32          currentAddr;
1469      UINT32          num = 0;
1470
1471      while((currentAddr = NvNextIndex(&iter)) != 0)
1472      {
1473          NV_INDEX    nvIndex;
1474
1475          // Get NV Index info
1476          _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
1477                              sizeof(NV_INDEX), &nvIndex);
1478          if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET) num++;
1479      }
1480
1481      return num;
1482  }
```

8.4.7.25 NvCapGetCounterAvail()

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

```
1483  UINT32
1484  NvCapGetCounterAvail(void)
1485  {
1486      UINT32          availNVSpace;
1487      UINT32          availRAMSpace;
1488      UINT32          counterNVSpace;
1489      UINT32          counterRAMSpace;
1490      UINT32          persistentNum = NvCapGetPersistentNumber();
1491
1492      // Get the available space in NV storage
1493      availNVSpace = NvGetFreeByte();
1494
1495      if (persistentNum < MIN_EVICT_OBJECTS)
1496      {
1497          // Some space have to be reserved for evict object. Adjust availNVSpace.
1498          UINT32      reserved = (MIN_EVICT_OBJECTS - persistentNum)
1499                               * NvGetEvictObjectSize();
1500          if (reserved > availNVSpace)
1501              availNVSpace = 0;
1502          else
1503              availNVSpace -= reserved;
1504      }
1505
1506      // Get the space needed to add a counter index to NV storage
1507      counterNVSpace = NvGetCounterSize();
1508
1509      // Compute the available space in RAM
1510      availRAMSpace = RAM_INDEX_SPACE - s_ramIndexSize;
1511
1512      // Compute the space needed to add a counter index to RAM storage
1513      // It takes an size field, a handle and sizeof(UINT64) for counter data
1514      counterRAMSpace = sizeof(UINT32) + sizeof(TPM_HANDLE) + sizeof(UINT64);
1515
1516      // Return the min of counter number in NV and in RAM
1517      if(availNVSpace / counterNVSpace > availRAMSpace / counterRAMSpace)
1518          return availRAMSpace / counterRAMSpace;
1519      else
1520          return availNVSpace / counterNVSpace;
1521  }
```

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

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

8.5.3 Functions

8.5.3.1 ObjectStartup()

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

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

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

8.5.3.2 ObjectCleanupEvict()

In this implementation, a persistent object is moved from NV into an object slot for processing. It is flushed after command execution. This function is called from ExecuteCommand().

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

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

    return;
}
```

8.5.3.3 ObjectIsPresent()

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

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

```
34  BOOL
35  ObjectIsPresent(
36      TPMI_DH_OBJECT  handle      // IN: handle to be checked
37  )
38  {
39      UINT32          slotIndex;          // index of object slot
40
41      pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
42
43      // The index in the loaded object array is found by subtracting the first
44      // object handle number from the input handle number. If the indicated
45      // slot is occupied, then indicate that there is already is a loaded
46      //  object associated with the handle.
47      slotIndex = handle - TRANSIENT_FIRST;
48      if(slotIndex >= MAX_LOADED_OBJECTS)
49          return FALSE;
50
51      return s_objects[slotIndex].occupied;
52  }
```

8.5.3.4 ObjectIsSequence()

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

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

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

8.5.3.5 ObjectGet()

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

 Copyright © TCG 2006-2013

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

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

8.5.3.6 ObjectGetName()

This function is used to access the Name of the object. In this implementation, the Name is computed when the object is loaded and is saved in the internal representation of the object. This function copies the Name data from the object into the buffer at *name* and returns the number of octets copied.

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

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

8.5.3.7 ObjectGetNameAlg()

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

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

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

8.5.3.8 ObjectGetQualifiedName()

This function returns the Qualified Name of the object. In this implementation, the Qualified Name is computed when the object is loaded and is saved in the internal representation of the object. The alternative would be to retain the Name of the parent and compute the QN when needed. This would take the same amount of space so it is not recommended that the alternate be used.

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

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

8.5.3.9 ObjectDataGetHierarchy()

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

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

8.5.3.10 ObjectGetHierarchy()

This function returns the handle of the hierarchy to which a handle belongs. This function is similar to ObjectDataGetHierarchy() but this routine takes a handle but ObjectDataGetHierarchy() takes an pointer to an object.

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

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

```
148 }
```

8.5.3.11 ObjectAllocateSlot()

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

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

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

8.5.3.12 ObjectLoad()

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

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

```
179 TPM_RC
180 ObjectLoad(
181     TPMI_RH_HIERARCHY    hierarchy,         // IN: hierarchy to which the object
182                                             //     belongs
183     TPMT_PUBLIC         *publicArea,        // IN: public area
```

```
184         TPMT_SENSITIVE      *sensitive,         // IN: sensitive area (may be null)
185         TPM2B_NAME          *name,              // IN: object's name (may be null)
186         TPM_HANDLE           parentHandle,      // IN: handle of parent
187         BOOL                 skipChecks,        // IN: flag to indicate if it is OK to
188                                                 //     skip consistency checks.
189         TPMI_DH_OBJECT      *handle             // OUT: object handle
190     )
191     {
192         OBJECT               *object = NULL;
193         OBJECT               *parent = NULL;
194         TPM_RC                result = TPM_RC_SUCCESS;
195         TPM2B_NAME            parentQN;         // Parent qualified name
196
197         // Try to allocate a slot for new object
198         if(!ObjectAllocateSlot(handle, &object))
199             return TPM_RC_OBJECT_MEMORY;
200
201         // Initialize public
202         object->publicArea = *publicArea;
203         if(sensitive != NULL)
204             object->sensitive = *sensitive;
205
206         // Are the consistency checks needed
207         if(!skipChecks)
208         {
209             // Check if key size matches
210             if(!CryptObjectIsPublicConsistent(&object->publicArea))
211             {
212                 result = TPM_RC_KEY;
213                 goto ErrorExit;
214             }
215             if(sensitive != NULL)
216             {
217                 // Check if public type matches sensitive type
218                 result = CryptObjectPublicPrivateMatch(object);
219                 if(result != TPM_RC_SUCCESS)
220                     goto ErrorExit;
221             }
222         }
223         object->attributes.publicOnly = (sensitive == NULL);
224
225         // If 'name' is NULL, then there is nothing left to do for this
226         // object as it has no qualified name and it is not a member of any
227         // hierarchy and it is temporary
228         if(name == NULL || name->t.size == 0)
229         {
230             object->qualifiedName.t.size = 0;
231             object->name.t.size = 0;
232             object->attributes.temporary = SET;
233             return TPM_RC_SUCCESS;
234         }
235         // If parent handle is a permanent handle, it is a primary or temporary
236         // object
237         if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
238         {
239             // initialize QN
240             parentQN.t.size = 4;
241
242             // for a primary key, parent qualified name is the handle of hierarchy
243             UINT32_TO_BYTE_ARRAY(parentHandle, parentQN.t.name);
244         }
245         else
246         {
247             // Get hierarchy and qualified name of parent
248             ObjectGetQualifiedName(parentHandle, &parentQN);
249
```

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

8.5.3.13 AllocateSequenceSlot()

This function allocates a sequence slot and initializes the parts that are used by the normal objects so that a sequence object is not inadvertently used for an operation that is not appropriate for a sequence.

```
297     static BOOL
298     AllocateSequenceSlot(
299         TPM_HANDLE          *newHandle,         // OUT: receives the allocated handle
300         HASH_OBJECT        **object,            // OUT: receives pointer to allocated
301                                                 //      object
302         TPM2B_AUTH          *auth               // IN: the authValue for the slot
303     )
304     {
305         OBJECT              *objectHash;        // the hash as an object
306 
307         if(!ObjectAllocateSlot(newHandle, &objectHash))
```

```
308             return FALSE;
309
310         *object = (HASH_OBJECT *)objectHash;
311
312         // Validate that the proper location of the hash state data relative to the
313         // object state data.
314         pAssert(&((*object)->auth) == &objectHash->publicArea.authPolicy);
315
316         // Set the common values that a sequence object shares with an ordinary object
317         // The type is TPM_ALG_NULL
318         (*object)->type = TPM_ALG_NULL;
319
320         // This has no name algorithm and the name is the Empty Buffer
321         (*object)->nameAlg = TPM_ALG_NULL;
322
323         // Clear the attributes
324         MemorySet(&((*object)->objectAttributes), 0, sizeof(TPMA_OBJECT));
325
326         // A sequence object is DA exempt.
327         (*object)->objectAttributes.noDA = SET;
328
329         if(auth != NULL)
330         {
331             MemoryRemoveTrailingZeros(auth);
332             (*object)->auth = *auth;
333         }
334         else
335             (*object)->auth.t.size = 0;
336         return TRUE;
337     }
```

8.5.3.14 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

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

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

```
363         return TPM_RC_SUCCESS;
364     }
```

8.5.3.15 ObjectCreateHashSequence()

This function creates a hash sequence object.

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

```
365     TPM_RC
366     ObjectCreateHashSequence(
367         TPMI_ALG_HASH        hashAlg,            // IN: hash algorithm
368         TPM2B_AUTH           *auth,              // IN: authValue
369         TPMI_DH_OBJECT       *newHandle          // OUT: sequence object handle
370     )
371     {
372         HASH_OBJECT          *hashObject;
373
374         // Try to allocate a slot for new object
375         if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
376             return TPM_RC_OBJECT_MEMORY;
377
378         // Set hash sequence bit
379         hashObject->attributes.hashSeq = SET;
380
381         // Start hash for hash sequence
382         CryptStartHashSequence(hashAlg, &hashObject->state.hashState[0]);
383
384         return TPM_RC_SUCCESS;
385     }
```

8.5.3.16 ObjectCreateEventSequence()

This function creates an event sequence object.

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

```
386     TPM_RC
387     ObjectCreateEventSequence(
388         TPM2B_AUTH           *auth,              // IN: authValue
389         TPMI_DH_OBJECT       *newHandle          // OUT: sequence object handle
390     )
391     {
392         HASH_OBJECT          *hashObject;
393         UINT32                count;
394         TPM_ALG_ID            hash;
395
396         // Try to allocate a slot for new object
397         if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
398             return TPM_RC_OBJECT_MEMORY;
399
400         // Set the event sequence attribute
401         hashObject->attributes.eventSeq = SET;
402
403
404         // Initialize hash states for each implemented PCR algorithms
405         for(count = 0; (hash = CryptGetHashAlgByIndex(count)) != TPM_ALG_NULL; count++)
406         {
```

```
407         // If this is a _TPM_Init or _TPM_HashStart, the sequence object will
408         // not leave the TPM so it doesn't need the sequence handling
409         if(auth == NULL)
410             CryptStartHash(hash, &hashObject->state.hashState[count]);
411         else
412             CryptStartHashSequence(hash, &hashObject->state.hashState[count]);
413     }
414     return TPM_RC_SUCCESS;
415 }
```

8.5.3.17 ObjectTerminateEvent()

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

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

8.5.3.18 ObjectContextLoad()

This function loads an object from a saved object context.

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

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

```
456
457     return TPM_RC_SUCCESS;
458 }
```

8.5.3.19 ObjectFlush()

This function frees an object slot.

This function requires that the object is loaded.

```
459 void
460 ObjectFlush(
461     TPMI_DH_OBJECT       handle              // IN: handle to be freed
462 )
463 {
464     UINT32      index = handle - TRANSIENT_FIRST;
465     pAssert(ObjectIsPresent(handle));
466
467     // Mark the handle slot as unoccupied
468     s_objects[index].occupied = FALSE;
469
470     // With no attributes
471     MemorySet((BYTE*)&(s_objects[index].object.entity.attributes),
472               0, sizeof(OBJECT_ATTRIBUTES));
473     return;
474 }
```

8.5.3.20 ObjectFlushHierarchy()

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

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

```
506         }
507
508         return;
509
510     }
```

8.5.3.21 ObjectLoadEvict()

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

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

Error Returns	Meaning
TPM_RC_REFERENCE_H0	The persistent object does not exist
TPM_RC_OBJECT_MEMORY	no object slot
TPM_RC_HANDLE	the handle points to an existing persistent object that was created was created by a disabled hierarchy

```
511     TPM_RC
512     ObjectLoadEvict(
513         TPM_HANDLE          *handle,        // IN:OUT: evict object handle.  If
514                                             //         success, it will be replace by
515                                             //         the loaded object handle
516         TPM_CC               commandCode    // IN: the command being processed
517     )
518     {
519         TPM_RC          result;
520         TPM_HANDLE      evictHandle = *handle;   // Save the evict handle
521         OBJECT          *object;
522
523         // If this is an index that references a persistent object created by
524         // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
525         if(*handle >=  PLATFORM_PERSISTENT)
526         {
527             // belongs to platform
528             if(g_phEnable == CLEAR)
529                 return TPM_RC_HANDLE;
530         }
531         // belongs to owner
532         else if(gc.shEnable == CLEAR)
533             return TPM_RC_HANDLE;
534
535         // Try to allocate a slot for an object
536         if(!ObjectAllocateSlot(handle, &object))
537             return TPM_RC_OBJECT_MEMORY;
538
539         // Copy persistent object to transient object slot.  A TPM_RC_HANDLE
540         // may be returned at this point. This will mark the slot as containing
541         // a transient object so that it will be flushed at the end of the
542         // command
543         result = NvGetEvictObject(evictHandle, object);
544
545         // Bail out if this failed
546         if(result != TPM_RC_SUCCESS)
547             return result;
548
549         // check the object to see if it is in the endorsement hierarchy
550         // if it is and this is not a TPM2_EvictControl() command, indicate
551         // that the hierarchy is disabled.
552         // If the associated hierarchy is disabled, make it look like the
553         // handle is not defined
554         if(    ObjectDataGetHierarchy(object) == TPM_RH_ENDORSEMENT
```

```
555             && gc.ehEnable == CLEAR
556             && commandCode != TPM_CC_EvictControl
557            )
558             return TPM_RC_HIERARCHY;
559
560     return result;
561 }
```

8.5.3.22 ObjectComputeName()

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

```
562 void
563 ObjectComputeName(
564     TPMT_PUBLIC         *publicArea,        // IN: public area of an object
565     TPM2B_NAME          *name               // OUT: name of the object
566 )
567 {
568     TPM2B_PUBLIC         marshalBuffer;
569     BYTE                *buffer;            // auxiliary marshal buffer pointer
570     HASH_STATE           hashState;         // hash state
571
572     // if the nameAlg is NULL then there is no name.
573     if(publicArea->nameAlg == TPM_ALG_NULL)
574     {
575         name->t.size = 0;
576         return;
577     }
578     // Start hash stack
579     name->t.size = CryptStartHash(publicArea->nameAlg, &hashState);
580
581     // Marshal the public area into its canonical form
582     buffer = marshalBuffer.b.buffer;
583
584     marshalBuffer.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
585
586     // Adding public area
587     CryptUpdateDigest2B(&hashState, &marshalBuffer.b);
588
589     // Complete hash leaving room for the name algorithm
590     CryptCompleteHash(&hashState, name->t.size, &name->t.name[2]);
591
592     // set the nameAlg
593     UINT16_TO_BYTE_ARRAY(publicArea->nameAlg, name->t.name);
594     name->t.size += 2;
595     return;
596 }
```

8.5.3.23 ObjectComputeQualifiedName()

This function computes the qualified name of an object.

```
597 void
598 ObjectComputeQualifiedName(
599     TPM2B_NAME          *parentQN,          // IN: parent's qualified name
600     TPM_ALG_ID           nameAlg,           // IN: name hash
601     TPM2B_NAME          *name,              // IN: name of the object
602     TPM2B_NAME          *qualifiedName      // OUT: qualified name of the object
603 )
604 {
605     HASH_STATE      hashState;   // hash state
606
607     //      QN_A = hash_A (QN of parent || NAME_A)
```

```
608
609         // Start hash
610         qualifiedName->t.size = CryptStartHash(nameAlg, &hashState);
611
612         // Add parent's qualified name
613         CryptUpdateDigest2B(&hashState, &parentQN->b);
614
615         // Add self name
616         CryptUpdateDigest2B(&hashState, &name->b);
617
618         // Complete hash leaving room for the name algorithm
619         CryptCompleteHash(&hashState, qualifiedName->t.size,
620                           &qualifiedName->t.name[2]);
621         UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
622         qualifiedName->t.size += 2;
623         return;
624     }
```

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

```
625     BOOL
626     ObjectDataIsStorage(
627         TPMT_PUBLIC         *publicArea         // IN: public area of the object
628     )
629     {
630         if(   CryptIsAsymAlgorithm(publicArea->type)           // must be asymmetric,
631            && publicArea->objectAttributes.restricted == SET   // restricted,
632            && publicArea->objectAttributes.decrypt == SET      // decryption key
633            && publicArea->objectAttributes.sign == CLEAR       // can not be sign key
634           )
635             return TRUE;
636         else
637             return FALSE;
638     }
```

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

```
639     BOOL
640     ObjectIsStorage(
641         TPMI_DH_OBJECT       handle             // IN: object handle
642     )
643     {
644         OBJECT           *object = ObjectGet(handle);
645         return ObjectDataIsStorage(&object->publicArea);
646     }
```

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

```
TPMI_YES_NO
ObjectCapGetLoaded(
    TPMI_DH_OBJECT       handle,            // IN: start handle
    UINT32               count,             // IN: count of returned handles
    TPML_HANDLE         *handleList         // OUT: list of handle
)
{
    TPMI_YES_NO          more = NO;
    UINT32               i;

    pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);

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

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

    // Iterate object slots to get loaded object handles
    for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
    {
        if(s_objects[i].occupied == TRUE)
        {
            // A valid transient object can not be the copy of a persistent object
            pAssert(s_objects[i].object.entity.attributes.evict == CLEAR);

            if(handleList->count < count)
            {
                // If we have not filled up the return list, add this object
                // handle to it
                handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
                handleList->count++;
            }
            else
            {
                // If the return list is full but we still have loaded object
                // available, report this and stop iterating
                more = YES;
                break;
            }
        }
    }

    return more;
}
```

8.5.3.27 ObjectCapGetTransientAvail()

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

```
UINT32
```

```
693  ObjectCapGetTransientAvail(void)
694  {
695      UINT32      i;
696      UINT32      num = 0;
697
698      // Iterate object slot to get the number of unoccupied slots
699      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
700      {
701          if(s_objects[i].occupied == FALSE) num++;
702      }
703
704      return num;
705  }
```

DRAFT

8.6 PCR.c

8.6.1 Introduction

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

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

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

8.6.2 Includes, Defines, and Data Definitions

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

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

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

8.6.3 Functions

8.6.3.1 PCRBelongsAuthGroup()

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

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

```
22  BOOL
23  PCRBelongsAuthGroup(
24      TPMI_DH_PCR      handle,        // IN: handle of PCR
```

```
25      UINT32          *groupIndex     // OUT: group index if PCR belongs a
26                                      //      group that allows authValue.  If PCR
27                                      //      does not belong to an auth group,
28                                      //      the value in this parameter is
29                                      //      invalid
30  )
31  {
32  #if NUM_AUTHVALUE_PCR_GROUP > 0
33      // Platform specification determines to which auth group a PCR belongs (if
34      // any). In this implementation, we assume there is only
35      // one auth group which contains PCR[20-22].  If the platform specification
36      // requires differently, the implementation should be changed accordingly
37      if(handle >= 20 && handle <= 22)
38      {
39          *groupIndex = 0;
40          return TRUE;
41      }
42
43  #endif
44      return FALSE;
45  }
```

8.6.3.2 PCRBelongsPolicyGroup()

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

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

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

8.6.3.3 PCRBelongsTCBGroup()

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

 Copyright © TCG 2006-2013

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

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

8.6.3.4 PCRPolicyIsAvailable()

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

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

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

8.6.3.5 PCRGetAuthValue()

This function is used to access the *authValue* of a PCR. If PCR does not belong to an *authValue* group, an Empty Auth will be returned.

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

```
108             auth->t.size = 0;
109         }
110
111         return;
112     }
```

8.6.3.6 PCRGetAuthPolicy()

This function is used to access the authorization policy of a PCR. It sets *policy* to the authorization policy and returns the hash algorithm for policy If the PCR does not allow a policy, TPM_ALG_NULL is returned.

```
113     TPMI_ALG_HASH
114     PCRGetAuthPolicy(
115         TPMI_DH_PCR      handle,         // IN: PCR handle
116         TPM2B_DIGEST    *policy          // OUT: policy of PCR
117     )
118     {
119         UINT32           groupIndex;
120
121         if(PCRBelongsPolicyGroup(handle, &groupIndex))
122         {
123             *policy = gp.pcrPolicies.policy[groupIndex];
124             return gp.pcrPolicies.hashAlg[groupIndex];
125         }
126         else
127         {
128             policy->t.size = 0;
129             return TPM_ALG_NULL;
130         }
131     }
```

8.6.3.7 PCRSimStart()

This function is used to initialize the policies when a TPM is manufactured. This function would only be called in a manufacturing environment or in a TPM simulator.

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

```
158             for(i = 0; i < PCR_SELECT_MAX; i++)
159                 gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
160                     = 0xFF;
161         }
162 
163         // Store the initial configuration to NV
164         NvWriteReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
165         NvWriteReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);
166 
167         return;
168     }
```

8.6.3.8 GetSavedPcrPointer()

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

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

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

 Copyright © TCG 2006-2013

8.6.3.9 IsPcrAllocated()

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

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

```
210  static BOOL
211  IsPcrAllocated (
212      UINT32               pcr,            // IN: The number of the PCR
213      TPMI_ALG_HASH        hashAlg         // IN: The PCR algorithm
214  )
215  {
216      UINT32               i;
217      BOOL                 allocated = FALSE;
218
219      if(pcr < IMPLEMENTATION_PCR)
220      {
221
222          for(i = 0; i < gp.pcrAllocated.count; i++)
223          {
224              if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
225              {
226                  if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr/8])
227                          & (1 << (pcr % 8))) != 0)
228                      allocated = TRUE;
229                  else
230                      allocated = FALSE;
231                  break;
232              }
233          }
234      }
235      return allocated;
236  }
```

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

```
237  static BYTE *
238  GetPcrPointer (
239      TPM_ALG_ID        alg,            // IN: algorithm for bank
240      UINT32            pcrNumber       // IN: PCR number
241  )
242  {
243      static BYTE     *pcr = NULL;
244
245      if(!IsPcrAllocated(pcrNumber, alg))
246          return NULL;
247
248      switch(alg)
249      {
250  #ifdef TPM_ALG_SHA1
251      case TPM_ALG_SHA1:
252          pcr = s_pcrs[pcrNumber].sha1Pcr;
```

```
253             break;
254     #endif
255     #ifdef TPM_ALG_SHA256
256         case TPM_ALG_SHA256:
257             pcr = s_pcrs[pcrNumber].sha256Pcr;
258             break;
259     #endif
260     #ifdef TPM_ALG_SHA384
261         case TPM_ALG_SHA384:
262             pcr = s_pcrs[pcrNumber].sha384Pcr;
263             break;
264     #endif
265     #ifdef TPM_ALG_SHA512
266         case TPM_ALG_SHA512:
267             pcr = s_pcrs[pcrNumber].sha512Pcr;
268             break;
269     #endif
270     #ifdef TPM_ALG_SM3_256
271         case TPM_ALG_SM3_256:
272             pcr = s_pcrs[pcrNumber].sm3_256Pcr;
273             break;
274     #endif
275         default:
276             pAssert(FALSE);
277             break;
278         }
279
280         return pcr;
281     }
```

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

```
282     static BOOL
283     IsPcrSelected (
284         UINT32                  pcr,            // IN: The number of the PCR
285         TPMS_PCR_SELECTION  *selection      // IN: The selection structure
286     )
287     {
288         BOOL                    selected = FALSE;
289         if(   pcr < IMPLEMENTATION_PCR
290            && ((selection->pcrSelect[pcr/8]) & (1 << (pcr % 8))) != 0)
291             selected = TRUE;
292
293         return selected;
294     }
```

8.6.3.12 FilterPcr()

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

```
295     static void
296     FilterPcr(
297         TPMS_PCR_SELECTION      *selection      // IN: input PCR selection
298     )
299     {
```

```
    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];
    }

    return;
}
```

8.6.3.13 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```
void
PCRStartup(
    STARTUP_TYPE                type            // IN: startup type
)
{
    UINT32              pcr, j;
    UINT32              saveIndex = 0;
    TPM2B_DIGEST        localityDigest = {0};
    BYTE                locality = (BYTE)_plat__LocalityGet();
    HASH_STATE          hashState;

    g_pcrReConfig = FALSE;

    if(type != SU_RESUME)
    {
        // PCR generation counter is cleared at TPM_RESET and TPM_RESTART
        gr.pcrCounter = 0;
    }
    else
        // any HCRTM data it tosed on Resume
        g_DrtmPreStartup = FALSE;


    // Initialize/Restore PCR values
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        BOOL        incrSaveIndex = FALSE;


```

```
359         // Iterate each hash algorithm bank
360         for(j = 0; j < gp.pcrAllocated.count; j++)
361         {
362             TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
363             BYTE            *pcrData = GetPcrPointer(hash, pcr);
364             UINT16           pcrSize = CryptGetHashDigestSize(hash);
365 
366             if(pcrData != NULL)
367             {
368                 if(type == SU_RESUME && s_initAttributes[pcr].stateSave == SET)
369                 {
370                     // Restore saved PCR value
371                     BYTE    *pcrSavedData;
372                     pcrSavedData = GetSavedPcrPointer(
373                                         gp.pcrAllocated.pcrSelections[j].hash,
374                                         saveIndex);
375                     MemoryCopy(pcrData, pcrSavedData, pcrSize, pcrSize);
376                     incrSaveIndex = TRUE;
377                 }
378                 else
379                     // PCR was not restored by state save
380                 {
381                     // If the reset locality of the PCR is 4, then
382                     // the reset value is all one's, otherwise it is
383                     // all zero.
384                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
385                         MemorySet(pcrData, 0xFF, pcrSize);
386                     else
387                     {
388                         // If this the HCRTM PCR and there was a HCRTM
389                         // Then create a digest of the correct length with
390                         // the LSO set to the locality
391                         if(pcr == HCRTM_PCR && g_DrtmPreStartup)
392                         {
393                             // Modify localityDigest so that it looks to be a
394                             // digest-sized value of all zeros but withe LSO
395                             // set to the locality of the Startup command.
396                             localityDigest.t.size = pcrSize;
397                             localityDigest.t.buffer[pcrSize - 1] = locality;
398 
399                             // Now compute PCRnew = H(locality || PCRold)
400                             CryptStartHash(hash, &hashState);
401                             CryptUpdateDigest(&hashState, pcrSize,
402                                 localityDigest.t.buffer);
403                             CryptUpdateDigest(&hashState, pcrSize, pcrData);
404                             CryptCompleteHash(&hashState, pcrSize, pcrData);
405 
406                             // Clean up the localityDigest buffer to prepare for
407                             // the next algoritm
408                             localityDigest.t.buffer[pcrSize - 1] = 0;
409                         }
410                         else
411                             MemorySet(pcrData, 0, pcrSize);
412 
413                     }
414                 }
415             }
416         }
417         g_DrtmPreStartup = FALSE;
418         if(incrSaveIndex == TRUE)
419             saveIndex++;
420     }
421 
422     // Reset authValues
423     if(type != SU_RESUME)
424     {
```

```
425             for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
426             {
427                 gc.pcrAuthValues.auth[j].t.size = 0;
428             }
429         }
430 
431     }
```

8.6.3.14 PCRStateSave()

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

```
432     void
433     PCRStateSave(
434         TPM_SU            type        // IN: startup type
435     )
436     {
437         UINT32            pcr, j;
438         UINT32            saveIndex = 0;
439 
440         // if state save CLEAR, nothing to be done.  Return here
441         if(type == TPM_SU_CLEAR) return;
442 
443         // Copy PCR values to the structure that should be saved to NV
444         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
445         {
446             BOOL        incrSaveIndex = FALSE;
447 
448             // Iterate each hash algorithm bank
449             for(j = 0; j < gp.pcrAllocated.count; j++)
450             {
451                 BYTE    *pcrData;
452                 UINT32  pcrSize;
453 
454                 pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
455 
456                 if(pcrData != NULL)
457                 {
458                     pcrSize
459                         = CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
460 
461                     if(s_initAttributes[pcr].stateSave == SET)
462                     {
463                         // Restore saved PCR value
464                         BYTE    *pcrSavedData;
465                         pcrSavedData
466                             = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
467                                                  saveIndex);
468                         MemoryCopy(pcrSavedData, pcrData, pcrSize, pcrSize);
469                         incrSaveIndex = TRUE;
470                     }
471                 }
472             }
473             if(incrSaveIndex == TRUE)
474                 saveIndex++;
475         }
476 
477         return;
478     }
```

8.6.3.15 PCRIsStateSaved()

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

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

```
479  BOOL
480  PCRIsStateSaved(
481      TPMI_DH_PCR         handle          // IN: PCR handle to be extended
482  )
483  {
484      UINT32              pcr = handle - PCR_FIRST;
485
486      if(s_initAttributes[pcr].stateSave == SET)
487          return TRUE;
488      else
489          return FALSE;
490  }
```

8.6.3.16 PCRIsResetAllowed()

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

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

```
491  BOOL
492  PCRIsResetAllowed(
493      TPMI_DH_PCR         handle              // IN: PCR handle to be extended
494  )
495  {
496      UINT8               commandLocality;
497      UINT8               localityBits = 1;
498      UINT32              pcr = handle - PCR_FIRST;
499
500      // Check for the locality
501      commandLocality = _plat__LocalityGet();
502      localityBits = localityBits << commandLocality;
503      if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
504          return FALSE;
505      else
506          return TRUE;
507
508  }
```

8.6.3.17 PCRChanged()

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

```
509  void
510  PCRChanged(
511      TPM_HANDLE      pcrHandle           // IN: the handle of the PCR that changed.
```

```
512         )
513     {
514         // For the reference implementation, the only change that does not cause
515         // increment is a change to a PCR in the TCB group.
516         if(!PCRBelongsTCBGroup(pcrHandle))
517             gr.pcrCounter++;
518     }
```

8.6.3.18 PCRIsExtendAllowed()

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

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

```
519     BOOL
520     PCRIsExtendAllowed(
521         TPMI_DH_PCR         handle              // IN: PCR handle to be extended
522     )
523     {
524         UINT8               commandLocality;
525         UINT8               localityBits = 1;
526         UINT32              pcr = handle - PCR_FIRST;
527
528         // Check for the locality
529         commandLocality = _plat__LocalityGet();
530         localityBits = localityBits << commandLocality;
531         if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
532             return FALSE;
533         else
534             return TRUE;
535
536     }
```

8.6.3.19 PCRExtend()

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

```
537     void
538     PCRExtend(
539         TPMI_DH_PCR          handle,     // IN: PCR handle to be extended
540         TPMI_ALG_HASH        hash,       // IN: hash algorithm of PCR
541         UINT32               size,       // IN: size of data to be extended
542         BYTE                *data        // IN: data to be extended
543     )
544     {
545         UINT32               pcr = handle - PCR_FIRST;
546         BYTE                *pcrData;
547         HASH_STATE           hashState;
548         UINT16               pcrSize;
549
550         pcrData = GetPcrPointer(hash, pcr);
551
552         // Extend PCR if it is allocated
553         if(pcrData != NULL)
554         {
555             pcrSize = CryptGetHashDigestSize(hash);
556             CryptStartHash(hash, &hashState);
557             CryptUpdateDigest(&hashState, pcrSize, pcrData);
```

```
558             CryptUpdateDigest(&hashState, size, data);
559             CryptCompleteHash(&hashState, pcrSize, pcrData);
560 
561             // If PCR does not belong to TCB group, increment PCR counter
562             if(!PCRBelongsTCBGroup(handle))
563                 gr.pcrCounter++;
564         }
565 
566         return;
567     }
```

8.6.3.20 PCRComputeCurrentDigest()

This function computes the digest of the selected PCR.

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

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

8.6.3.21 PCRRead()

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

```
614  void
615  PCRRead(
616      TPML_PCR_SELECTION      *selection,     // IN/OUT: PCR selection (filtered on
617                                              //         output)
618      TPML_DIGEST             *digest,        // OUT: digest
619      UINT32                  *pcrCounter     // OUT: the current value of PCR
620                                              //      generation number
621  )
622  {
623      TPMS_PCR_SELECTION      *select;
624      BYTE                    *pcrData;       // will point to a digest
625      UINT32                   pcr;
626      UINT32                   i;
627
628      digest->count = 0;
629
630      // Iterate through the list of PCR selection structures
631      for(i = 0; i < selection->count; i++)
632      {
633          // Point to the current selection
634          select = &selection->pcrSelections[i]; // Point to the current selection
635          FilterPcr(select);      // Clear out the bits for unimplemented PCR
636
637          // Iterate through the selection
638          for (pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
639          {
640              if(IsPcrSelected(pcr, select))        // Is this PCR selected
641              {
642                  // Check if number of digest exceed upper bound
643                  if(digest->count > 7)
644                  {
645                      // Clear rest of the current select bitmap
646                      while(   pcr < IMPLEMENTATION_PCR
647                               // do not round up!
648                            && (pcr / 8) < select->sizeofSelect)
649                      {
650                          // do not round up!
651                          select->pcrSelect[pcr/8] &= (BYTE) ~(1 << (pcr % 8));
652                          pcr++;
653                      }
654                      // Exit inner loop
655                      break;;
656                  }
657                  // Need the size of each digest
658                  digest->digests[digest->count].t.size =
659                      CryptGetHashDigestSize(selection->pcrSelections[i].hash);
660
661                  // Get pointer to the digest data for the bank
662                  pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
663                  pAssert(pcrData != NULL);
664                  // Add to the data to digest
665                  MemoryCopy(digest->digests[digest->count].t.buffer,
666                             pcrData,
667                             digest->digests[digest->count].t.size,
668                             digest->digests[digest->count].t.size);
669                  digest->count++;
670              }
671          }
672          // If we exit inner loop because we have exceed the output upper bound
673          if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
```

```
674             {
675                 // Clear rest of the selection
676                 while(i < selection->count)
677                 {
678                     MemorySet(selection->pcrSelections[i].pcrSelect, 0,
679                               selection->pcrSelections[i].sizeofSelect);
680                     i++;
681                 }
682                 // exit outer loop
683                 break;
684             }
685         }
686 
687         *pcrCounter = gr.pcrCounter;
688 
689         return;
690     }
```

8.6.3.22 PcrWrite()

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

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

8.6.3.23 PCRAllocate()

This function is used to change the PCR allocation.

Return Value	Meaning
YES	allocate success
NO	allocate fail

```
710     TPMI_YES_NO
711     PCRAllocate(
712         TPML_PCR_SELECTION      *allocate,          // IN: required allocation
713         UINT32                  *maxPCR,            // OUT: Maximum number of PCR
714         UINT32                  *sizeNeeded,        // OUT: required space
715         UINT32                  *sizeAvailable      // OUT: available space
716     )
717     {
718         UINT32                  i, j, k;
719         TPML_PCR_SELECTION      newAllocate;
720 
```

```
721         // Create the expected new PCR allocation based on the existing allocation
722         // and the new input:
723         //  1. if a PCR bank does not appear in the new allocation, the existing
724         //     allocation of this PCR bank will be preserved.
725         //  2. if a PCR bank appears multiple times in the new allocation, only the
726         //     last one will be in effect.
727         newAllocate = gp.pcrAllocated;
728         for(i = 0; i < allocate->count; i++)
729         {
730             for(j = 0; j < newAllocate.count; j++)
731             {
732                 // If hash matches, the new allocation covers the old allocation
733                 // for this particular bank.
734                 // The assumption is the initial PCR allocation (from manufacture)
735                 // has all the supported hash algorithms allocated.  So there must
736                 // be a match for any new bank allocation from the input.
737                 if(newAllocate.pcrSelections[j].hash ==
738                     allocate->pcrSelections[i].hash)
739                 {
740                     newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
741                     break;
742                 }
743             }
744             // The j loop must exit with a match.
745             pAssert(j < newAllocate.count);
746         }
747 
748         // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
749         *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
750         if(*maxPCR > IMPLEMENTATION_PCR)
751             *maxPCR = IMPLEMENTATION_PCR;
752 
753         // Compute required size for allocation
754         *sizeNeeded = 0;
755         for(i = 0; i < newAllocate.count; i++)
756         {
757             UINT32      digestSize
758                 = CryptGetHashDigestSize(newAllocate.pcrSelections[i].hash);
759             for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
760             {
761                 BYTE        mask = 1;
762                 for(k = 0; k < 8; k++)
763                 {
764                     if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
765                         *sizeNeeded += digestSize;
766                     mask = mask << 1;
767                 }
768             }
769         }
770 
771         // In this particular implementation, we always have enough space to
772         // allocate PCR.  Different implementation may return a sizeAvailable less
773         // than the sizeNeed.
774         *sizeAvailable = sizeof(s_pcrs);
775 
776         // Save the required allocation to NV.  Note that after NV is written, the
777         // PCR allocation in NV is no longer consistent with the RAM data
778         // gp.pcrAllocated.  The NV version reflect the allocate after next
779         // TPM_RESET, while the RAM version reflects the current allocation
780         NvWriteReserved(NV_PCR_ALLOCATED, &newAllocate);
781 
782         return YES;
783 
784     }
```

8.6.3.24 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.

```
785 void
786 PCRSetValue(
787     TPM_HANDLE        handle,             // IN: the handle of the PCR to set
788     INT8              initialValue        // IN: the value to set
789     )
790 {
791     int               i;
792     UINT32            pcr = handle - PCR_FIRST;
793     TPMI_ALG_HASH     hash;
794     UINT16            digestSize;
795     BYTE             *pcrData;
796 
797     // Iterate supported PCR bank algorithms to reset
798     for(i = 0; i < HASH_COUNT; i++)
799     {
800         hash = CryptGetHashAlgByIndex(i);
801         // Prevent runaway
802         if(hash == TPM_ALG_NULL)
803             break;
804 
805         // Get a pointer to the data
806         pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
807 
808         // If the PCR is allocated
809         if(pcrData != NULL)
810         {
811             // And the size of the digest
812             digestSize = CryptGetHashDigestSize(hash);
813 
814             // Set the LSO to the input value
815             pcrData[digestSize - 1] = initialValue;
816 
817             // Sign extend
818             if(initialValue >= 0)
819                 MemorySet(pcrData, 0, digestSize - 1);
820             else
821                 MemorySet(pcrData, -1, digestSize - 1);
822         }
823     }
824 }
```

8.6.3.25 PCRResetDynamics

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

```
825 void
826 PCRResetDynamics(void)
827 {
828     UINT32              pcr, i;
829 
830     // Initialize PCR values
831     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
832     {
833         // Iterate each hash algorithm bank
834         for(i = 0; i < gp.pcrAllocated.count; i++)
835         {
836             BYTE    *pcrData;
837             UINT32  pcrSize;
```

```
838
839             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
840
841             if(pcrData != NULL)
842             {
843                 pcrSize =
844                     CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
845
846                 // Reset PCR
847                 // Any PCR can be reset by locality 4 should be reset to 0
848                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
849                     MemorySet(pcrData, 0, pcrSize);
850             }
851         }
852     }
853     return;
854 }
```

8.6.3.26 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

```
855 TPMI_YES_NO
856 PCRCapGetAllocation(
857     UINT32                  count,              // IN: count of return
858     TPML_PCR_SELECTION      *pcrSelection       // OUT: PCR allocation list
859 )
860 {
861     if(count == 0)
862     {
863         pcrSelection->count = 0;
864         return YES;
865     }
866     else
867     {
868         *pcrSelection = gp.pcrAllocated;
869         return NO;
870     }
871 }
```

8.6.3.27 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```
872 static void
873 PCRSetSelectBit(
874     UINT32          pcr,                    // IN: PCR number
875     BYTE            *bitmap                 // OUT: bit map to be set
876 )
877 {
878     bitmap[pcr / 8] |= (1 << (pcr % 8));
879     return;
880 }
```

DRAFT

8.6.3.28 PCRGetProperty()

This function returns the selected PCR property.

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

```
881  static BOOL
882  PCRGetProperty(
883      TPM_PT_PCR                  property,
884      TPMS_TAGGED_PCR_SELECT      *select
885  )
886  {
887      UINT32              pcr;
888      UINT32              groupIndex;
889
890      select->tag = property;
891      // Always set the bitmap to be the size of all PCR
892      select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
893
894      // Initialize bitmap
895      MemorySet(select->pcrSelect, 0, select->sizeofSelect);
896
897      // Collecting properties
898      for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
899      {
900          switch(property)
901          {
902              case TPM_PT_PCR_SAVE:
903                  if(s_initAttributes[pcr].stateSave == SET)
904                      PCRSetSelectBit(pcr, select->pcrSelect);
905                  break;
906              case TPM_PT_PCR_EXTEND_L0:
907                  if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
908                      PCRSetSelectBit(pcr, select->pcrSelect);
909                  break;
910              case TPM_PT_PCR_RESET_L0:
911                  if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
912                      PCRSetSelectBit(pcr, select->pcrSelect);
913                  break;
914              case TPM_PT_PCR_EXTEND_L1:
915                  if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
916                      PCRSetSelectBit(pcr, select->pcrSelect);
917                  break;
918              case TPM_PT_PCR_RESET_L1:
919                  if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
920                      PCRSetSelectBit(pcr, select->pcrSelect);
921                  break;
922              case TPM_PT_PCR_EXTEND_L2:
923                  if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
924                      PCRSetSelectBit(pcr, select->pcrSelect);
925                  break;
926              case TPM_PT_PCR_RESET_L2:
927                  if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
928                      PCRSetSelectBit(pcr, select->pcrSelect);
929                  break;
930              case TPM_PT_PCR_EXTEND_L3:
931                  if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
932                      PCRSetSelectBit(pcr, select->pcrSelect);
933                  break;
934              case TPM_PT_PCR_RESET_L3:
935                  if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
```

```
936                     PCRSetSelectBit(pcr, select->pcrSelect);
937                 break;
938             case TPM_PT_PCR_EXTEND_L4:
939                 if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
940                     PCRSetSelectBit(pcr, select->pcrSelect);
941                 break;
942             case TPM_PT_PCR_RESET_L4:
943                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
944                     PCRSetSelectBit(pcr, select->pcrSelect);
945                 break;
946             case TPM_PT_PCR_DRTM_RESET:
947                 // DRTM reset PCRs are the PCR reset by locality 4
948                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
949                     PCRSetSelectBit(pcr, select->pcrSelect);
950                 break;
951 #if NUM_POLICY_PCR_GROUP > 0
952             case TPM_PT_PCR_POLICY:
953                 if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
954                     PCRSetSelectBit(pcr, select->pcrSelect);
955                 break;
956 #endif
957 #if NUM_AUTHVALUE_PCR_GROUP > 0
958             case TPM_PT_PCR_AUTH:
959                 if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
960                     PCRSetSelectBit(pcr, select->pcrSelect);
961                 break;
962 #endif
963 #if ENABLE_PCR_NO_INCREMENT == YES
964             case TPM_PT_PCR_NO_INCREMENT:
965                 if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
966                     PCRSetSelectBit(pcr, select->pcrSelect);
967                 break;
968 #endif
969             default:
970                 // If property is not supported, stop scanning PCR attributes
971                 // and return.
972                 return FALSE;
973                 break;
974         }
975     }
976     return TRUE;
977 }
```

8.6.3.29 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

```
978 TPMI_YES_NO
979 PCRCapGetProperties(
980     TPM_PT_PCR                  property,       // IN: the starting PCR property
981     UINT32                      count,          // IN: count of returned
982     // properties
983     TPML_TAGGED_PCR_PROPERTY    *select         // OUT: PCR select
984 )
985 {
986     TPMI_YES_NO     more = NO;
987     UINT32          i;
988
```

```
989         // Initialize output property list
990         select->count = 0;
991 
992         // The maximum count of properties we may return is MAX_PCR_PROPERTIES
993         if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
994 
995         // TPM_PT_PCR_FIRST is defined as 0 in spec.  It ensures that property
996         // value would never be less than TPM_PT_PCR_FIRST
997         pAssert(TPM_PT_PCR_FIRST == 0);
998 
999         // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last propety
1000        // implemented on the TPM.
1001        for(i = property; i <= TPM_PT_PCR_LAST; i++)
1002        {
1003            if(select->count < count)
1004            {
1005                // If we have not filled up the return list, add more properties to it
1006                if(PCRGetProperty(i, &select->pcrProperty[select->count]))
1007                    // only increment if the property is implemented
1008                select->count++;
1009            }
1010            else
1011            {
1012                // If the return list is full but we still have properties
1013                // available, report this and stop iterating.
1014                more = YES;
1015                break;
1016            }
1017        }
1018        return more;
1019    }
```

8.6.3.30 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

```
1020    TPMI_YES_NO
1021    PCRCapGetHandles(
1022        TPMI_DH_PCR             handle,         // IN: start handle
1023        UINT32                  count,          // IN: count of returned handles
1024        TPML_HANDLE             *handleList     // OUT: list of handle
1025    )
1026    {
1027        TPMI_YES_NO     more = NO;
1028        UINT32          i;
1029 
1030        pAssert(HandleGetType(handle) == TPM_HT_PCR);
1031 
1032        // Initialize output handle list
1033        handleList->count = 0;
1034 
1035        // The maximum count of handles we may return is MAX_CAP_HANDLES
1036        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1037 
1038        // Iterate PCR handle range
1039        for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1040        {
```

```
1041            if(handleList->count < count)
1042            {
1043                // If we have not filled up the return list, add this PCR
1044                // handle to it
1045                handleList->handle[handleList->count] = i + PCR_FIRST;
1046                handleList->count++;
1047            }
1048            else
1049            {
1050                // If the return list is full but we still have PCR handle
1051                // available, report this and stop iterating
1052                more = YES;
1053                break;
1054            }
1055        }
1056        return more;
1057    }
```

DRAFT

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

```
#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.

```
void
PhysicalPresencePreInstall_Init(void)
{
    // Clear all the PP commands
    MemorySet(&gp.ppList, 0,
              ((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7) / 8);

    // TPM_CC_PP_Commands always requires PP
    if(CommandIsImplemented(TPM_CC_PP_Commands))
        PhysicalPresenceCommandSet(TPM_CC_PP_Commands);

    // Write PP list to NV
    NvWriteReserved(NV_PP_LIST, &gp.ppList);

    return;
}
```

8.7.3.2 PhysicalPresenceCommandSet()

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

```
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);
```

```
37
38      return;
39  }
```

8.7.3.3 PhysicalPresenceCommandClear()

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

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

8.7.3.4 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

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

```
84  }
```

8.7.3.5 PhysicalPresenceCapGetCCList()

This function returns a list of commands that require PP confirmation. The list starts from the first implemented command that has a command code that the same or greater than *commandCode*.

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

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

8.8 Session.c

8.8.1 Introduction

The code in this file is used to manage the session context counter. The scheme implemented here is a "truncated counter". This scheme allows the TPM to not need TPM_SU_CLEAR for a very long period of time and still not have the context count for a session repeated.

The counter *(contextCounter*)in this implementation is a UINT64 but can be smaller. The "tracking array" *(contextArray*) only has 16-bits per context. The tracking array is the data that needs to be saved and restored across TPM_SU_STATE so that sessions are not lost when the system enters the sleep state. Also, when the TPM is active, the tracking array is kept in RAM making it important that the number of bytes for each entry be kept as small as possible.

The TPM prevents **collisions** of these truncated values by not allowing a *contextID* to be assigned if it would be the same as an existing value. Since the array holds 16 bits, after a context has been saved, an additional 2^16-1 contexts may be saved before the count would again match. The normal expectation is that the context will be flushed before its count value is needed again but it is always possible to have long-lived sessions.

The *contextID* is assigned when the context is saved (TPM2_ContextSave()). At that time, the TPM will compare the low-order 16 bits of *contextCounter* to the existing values in *contextArray* and if one matches, the TPM will return TPM_RC_CONTEXT_GAP (by construction, the entry that contains the matching value is the oldest context).

The expected remediation by the TRM is to load the oldest saved session context (the one found by the TMP), and save it. Since loading the oldest session also eliminates its *contextID* value from *contextArray*, there TPM will always be able to load and save the oldest existing context.

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

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

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

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

EXAMPLE: If the TPM allows 8 loaded sessions, then the slot numbers would be 1-8 and a *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

 Copyright © TCG 2006-2013

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

```
#define SESSION_C
#include "InternalRoutines.h"
#include "Platform.h"
#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.

```
static void
ContextIdSetOldest(void)
{
    CONTEXT_SLOT    lowBits;
    CONTEXT_SLOT    entry;
    CONTEXT_SLOT    smallest = ((CONTEXT_SLOT) ~0);
    UINT32  i;

    // Set oldestSaveContext to a value indicating none assigned
    s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;

    lowBits = (CONTEXT_SLOT)gr.contextCounter;
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        entry = gr.contextArray[i];

        // only look at entries that are saved contexts
        if(entry > MAX_LOADED_SESSIONS)
        {
            // Use a less than or equal in case the oldest
            // is brand new (= lowBits-1) and equal to our initial
            // value for smallest.
            if(((CONTEXT_SLOT) (entry - lowBits)) <= smallest)
            {
                smallest = (entry - lowBits);
                s_oldestSavedSession = i;
            }
        }
    }
    // When we finish, either the s_oldestSavedSession still has its initial
    // value, or it has the index of the oldest saved context.
}
```

8.8.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```
37  void
38  SessionStartup(
39      STARTUP_TYPE         type
40  )
41  {
42      UINT32               i;
43
44      // Initialize session slots.  At startup, all the in-memory session slots
45      // are cleared and marked as not occupied
46      for(i = 0; i < MAX_LOADED_SESSIONS; i++)
47          s_sessions[i].occupied = FALSE;   // session slot is not occupied
48
49      // The free session slots the number of maximum allowed loaded sessions
50      s_freeSessionSlots = MAX_LOADED_SESSIONS;
51
52      // Initialize context ID data.  On a ST_SAVE or hibernate sequence, it will
53      // scan the saved array of session context counts, and clear any entry that
54      // references a session that was in memory during the state save since that
55      // memory was not preserved over the ST_SAVE.
56      if(type == SU_RESUME || type == SU_RESTART)
57      {
58          // On ST_SAVE we preserve the contexts that were saved but not the ones
59          // in memory
60          for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
61          {
62              // If the array value is unused or references a loaded session then
63              // that loaded session context is lost and the array entry is
64              // reclaimed.
65              if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
66                  gr.contextArray[i] = 0;
67          }
68          // Find the oldest session in context ID data and set it in
69          // s_oldestSavedSession
70          ContextIdSetOldest();
71      }
72      else
73      {
74          // For STARTUP_CLEAR, clear out the contextArray
75          for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
76              gr.contextArray[i] = 0;
77
78          // reset the context counter
79          gr.contextCounter = MAX_LOADED_SESSIONS + 1;
80
81          // Initialize oldest saved session
82          s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
83      }
84      return;
85  }
```

8.8.5 Access Functions

8.8.5.1 SessionIsLoaded()

This function test a session handle references a loaded session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE: A PWAP authorization does not have a session.

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

```
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

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

8.8.5.4 SessionGet()

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

The function requires that the session is loaded.

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

 Copyright © TCG 2006-2013

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

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

```
207 }
```

8.8.6.2 SessionCreate()

This function does the detailed work for starting an authorization session. This is done in a support routine rather than in the action code because the session management may differ in implementations. This implementation uses a fixed memory allocation to hold sessions and a fixed allocation to hold the *contextID* for the saved contexts.

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	need to recycle sessions
TPM_RC_SESSION_HANDLE	active session space is full
TPM_RC_SESSION_MEMORY	loaded session space is full

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

```
255
256         // Initialize the session data
257         MemorySet(session, 0, sizeof(SESSION));
258
259         // Initialize internal session data
260         session->authHashAlg = authHash;
261         // Initialize session type
262         if(sessionType == TPM_SE_HMAC)
263         {
264             *sessionHandle += HMAC_SESSION_FIRST;
265
266         }
267         else
268         {
269             *sessionHandle += POLICY_SESSION_FIRST;
270
271             // For TPM_SE_POLICY or TPM_SE_TRIAL
272             session->attributes.isPolicy = SET;
273             if(sessionType == TPM_SE_TRIAL)
274                 session->attributes.isTrialPolicy = SET;
275
276             // Initialize policy session data
277             SessionInitPolicyData(session);
278         }
279         // Create initial session nonce
280         session->nonceTPM.t.size = nonceCaller->t.size;
281         CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
282
283         // Set up session parameter encryption algorithm
284         session->symmetric = *symmetric;
285
286         // If there is a bind object or a session secret, then need to compute
287         // a sessionKey.
288         if(bind != TPM_RH_NULL || seed->t.size != 0)
289         {
290             // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
291             //                       nonceCaller, bits)
292             // The HMAC key for generating the sessionSecret can be the concatenation
293             // of an authorization value and a seed value
294             TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
295             TPM2B_KEY            key;
296
297             UINT16               hashSize;      // The size of the hash used by the
298                                                 // session crated by this command
299             TPM2B_AUTH  entityAuth;             // The authValue of the entity
300                                                 // associated with HMAC session
301
302             // Get hash size, which is also the length of sessionKey
303             hashSize = CryptGetHashDigestSize(session->authHashAlg);
304
305             // Get authValue of associated entity
306             entityAuth.t.size = EntityGetAuthValue(bind, &entityAuth.t.buffer);
307
308             // Concatenate authValue and seed
309             pAssert(entityAuth.t.size + seed->t.size <= <K>sizeof(key.t.buffer));
310             MemoryCopy2B(&key.b, &entityAuth.b, sizeof(key.t.buffer));
311             MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));
312
313             session->sessionKey.t.size = hashSize;
314
315             // Compute the session key
316             KDFa(session->authHashAlg, &key.b, "ATH", &session->nonceTPM.b,
317                  &nonceCaller->b, hashSize * 8, session->sessionKey.t.buffer, NULL);
318         }
319
320         // Copy the name of the entity that the HMAC session is bound to
```

```
321         // Policy session is not bound to an entity
322         if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
323         {
324             session->attributes.isBound = SET;
325             SessionComputeBoundEntity(bind, &session->u1.boundEntity);
326         }
327         // If there is a bind object and it is subject to DA, then use of this session
328         // is subject to DA regardless of how it is used.
329         session->attributes.isDaBound =     (bind != TPM_RH_NULL)
330                                         && (IsDAExempted(bind) == FALSE);
331
332         // If the session is bound, then check to see if it is bound to lockoutAuth
333         session->attributes.isLockoutBound =     (session->attributes.isDaBound  == SET)
334                                               && (bind == TPM_RH_LOCKOUT);
335         return TPM_RC_SUCCESS;
336
337     }
```

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

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

```
369
370         // Set the contextID for the contextArray
371         gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
372
373         // Increment the counter
374         gr.contextCounter++;
375
376         // In the unlikely event that the 64-bit context counter rolls over...
377         if(gr.contextCounter == 0)
378         {
379             // back it up
380             gr.contextCounter--;
381             // return an error
382             return TPM_RC_TOO_MANY_CONTEXTS;
383         }
384         // if the low-order bits wrapped, need to advance the value to skip over
385         // the values used to indicate that a session is loaded
386         if(((CONTEXT_SLOT)gr.contextCounter) == 0)
387             gr.contextCounter += MAX_LOADED_SESSIONS + 1;
388
389         // If no other sessions are saved, this is now the oldest.
390         if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
391             s_oldestSavedSession = contextIndex;
392
393         // Mark the session slot as unoccupied
394         s_sessions[slotIndex].occupied = FALSE;
395
396         // and indicate that there is an additional open slot
397         s_freeSessionSlots++;
398
399         return TPM_RC_SUCCESS;
400     }
```

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

```
401     TPM_RC
402     SessionContextLoad(
403         SESSION                 *session,     // IN: session structure from saved
404         //     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         // Get the current contents of the array
476         slotIndex = gr.contextArray[contextIndex];
477
478         // Mark context array entry as available
479         gr.contextArray[contextIndex] = 0;
480
481         // Is this a saved session being flushed
482         if(slotIndex > MAX_LOADED_SESSIONS)
483         {
484             // Flushing the oldest session?
485             if(contextIndex == s_oldestSavedSession)
486                 // If so, find a new value for oldest.
487                 ContextIdSetOldest();
488         }
489         else
490         {
491             // Adjust slot index to point to session array index
492             slotIndex -= 1;
493
494             // Free session array index
495             s_sessions[slotIndex].occupied = FALSE;
496             s_freeSessionSlots++;
497         }
498
499         return;
500     }
```

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.

```
501     void
502     SessionComputeBoundEntity(
503         TPMI_DH_ENTITY      entityHandle,  // IN: handle of entity
504         TPM2B_NAME         *bind           // OUT: binding value
505     )
506     {
507         TPM2B_AUTH          auth;
508         INT16               overlap;
509
510         // Get name
511         bind->t.size = EntityGetName(entityHandle, &bind->t.name);
512
513     //    // The bound value of a reserved handle is the handle itself
514     //    if(bind->t.size == sizeof(TPM_HANDLE)) return;
515
516         // For all the other entities, concatenate the auth value to the name.
517         // Get a local copy of the auth value because some overlapping
518         // may be necessary.
519         auth.t.size = EntityGetAuthValue(entityHandle, &auth.t.buffer);
520         pAssert(auth.t.size <= <K>sizeof(TPMU_HA));
521
522         // Figure out if there will be any overlap
523         overlap = bind->t.size + auth.t.size - sizeof(bind->t.name);
524
525         // There is overlap if the combined sizes are greater than will fit
526         if(overlap > 0)
527         {
528             // The overlap area is at the end of the Name
```

```
529             BYTE    *result = &bind->t.name[bind->t.size - overlap];
530             int     i;
531
532             // XOR the auth value into the Name for the overlap area
533             for(i = 0; i < overlap; i++)
534                 result[i] ^= auth.t.buffer[i];
535         }
536         else
537         {
538             // There is no overlap
539             overlap = 0;
540         }
541         //copy the remainder of the authData to the end of the name
542         MemoryCopy(&bind->t.name[bind->t.size], &auth.t.buffer[overlap],
543                    auth.t.size - overlap, sizeof(bind->t.name) - bind->t.size);
544
545         // Increase the size of the bind data by the size of the auth - the overlap
546         bind->t.size += auth.t.size-overlap;
547
548         return;
549     }
```

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.

```
550     void
551     SessionInitPolicyData(
552         SESSION         *session        // IN: session handle
553     )
554     {
555         // Initialize start time
556         session->startTime = go.clock;
557
558         // Initialize policyDigest.  policyDigest is initialized with a string of 0 of
559         // session algorithm digest size. Since the policy already contains all zeros
560         // it is only necessary to set the size
561         session->u2.policyDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
562         return;
563     }
```

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.

```
564     void
565     SessionResetPolicyData(
566         SESSION         *session        // IN: the session to reset
567     )
568     {
569         session->commandCode = 0;       // No command
570
571         // No locality selected
572         MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
573
574         // The cpHash size to zero
575         session->u1.cpHash.b.size = 0;
576
577         // No timeout
578         session->timeOut = 0;
579
580         // Reset the pcrCounter
```

```
581         session->pcrCounter = 0;
582
583         // Reset the policy hash
584         MemorySet(&session->u2.policyDigest.t.buffer, 0,
585                   session->u2.policyDigest.t.size);
586
587         // Reset the session attributes
588         MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
589
590         // set the policy attribute
591         session->attributes.isPolicy = SET;
592     }
```

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

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

```
                }
                else
                {
                    // If the return list is full but we still have loaded object
                    // available, report this and stop iterating
                    more = YES;
                    break;
                }
            }
        }
    }

    return more;

}
```

8.8.6.10 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

```
TPMI_YES_NO
SessionCapGetSaved(
    TPMI_SH_HMAC            handle,         // IN: start handle
    UINT32                  count,          // IN: count of returned handles
    TPML_HANDLE             *handleList     // OUT: list of handle
)
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);

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

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

    // Iterate session context ID slots to get loaded session handles
    for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
    {
        // If session is active
        if(gr.contextArray[i] != 0)
        {
            // If session is saved
            if (gr.contextArray[i] > MAX_LOADED_SESSIONS)
            {
                if(handleList->count < count)
                {
                    // If we have not filled up the return list, add this
                    // session handle to it
                    handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
                    handleList->count++;
                }
                else
                {
                    // If the return list is full but we still have loaded object
```

```
685                     // available, report this and stop iterating
686                     more = YES;
687                     break;
688                 }
689             }
690         }
691     }
692
693     return more;
694
695 }
```

8.8.6.11 SessionCapGetLoadedNumber()

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

```
696 UINT32
697 SessionCapGetLoadedNumber(void)
698 {
699     return MAX_LOADED_SESSIONS - s_freeSessionSlots;
700 }
```

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.

```
701 UINT32
702 SessionCapGetLoadedAvail(void)
703 {
704     return s_freeSessionSlots;
705 }
```

8.8.6.13 SessionCapGetActiveNumber()

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

```
706 UINT32
707 SessionCapGetActiveNumber(void)
708 {
709     UINT32              i;
710     UINT32              num = 0;
711
712     // Iterate the context array to find the number of non-zero slots
713     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
714     {
715         if(gr.contextArray[i] != 0) num++;
716     }
717
718     return num;
719 }
```

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.

```
720 UINT32
721 SessionCapGetActiveAvail(void)
722 {
723     UINT32              i;
724     UINT32              num = 0;
725
726     // Iterate the context array to find the number of zero slots
727     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
728     {
729         if(gr.contextArray[i] == 0) num++;
730     }
731
732     return num;
733 }
```

DRAFT

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

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

8.9.3 Functions

8.9.3.1 TimePowerOn()

This function initialize time info at _TPM_Init().

```
void
TimePowerOn(void)
{
    TPM_SU          orderlyShutDown;

    // Read orderly data info from NV memory
    NvReadReserved(NV_ORDERLY_DATA, &go);

    // Read orderly shut down state flag
    NvReadReserved(NV_ORDERLY, &orderlyShutDown);

    // If the previous cycle is orderly shut down, the value of the safe bit
    // the same as previously saved.  Otherwise, it is not safe.
    if(orderlyShutDown == SHUTDOWN_NONE)
        go.clockSafe= NO;
    else
        go.clockSafe = YES;

    // Set the initial state of the DRBG
    CryptDrbgGetPutState(PUT_STATE);

    // Clear time since TPM power on
    g_time = 0;

    return;
}
```

8.9.3.2 TimeStartup()

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

```
void
TimeStartup(
    STARTUP_TYPE        type        // IN: start up type
)
{
    if(type == SU_RESUME)
    {
        // Resume sequence
        gr.restartCount++;
```

```
38      }
39      else
40      {
41          if(type == SU_RESTART)
42          {
43              // Hibernate sequence
44              gr.clearCount++;
45              gr.restartCount++;
46          }
47          else
48          {
49              // Reset sequence
50              // Increase resetCount
51              gp.resetCount++;
52
53              // Write resetCount to NV
54              NvWriteReserved(NV_RESET_COUNT, &gp.resetCount);
55              gp.totalResetCount++;
56
57              // We do not expect the total reset counter overflow during the life
58              // time of TPM.  if it ever happens, TPM will be put to failure mode
59              // and there is no way to recover it.
60              // The reason that there is no recovery is that we don't increment
61              // the NV totalResetCount when incrementing would make it 0. When the
62              // TPM starts up again, the old value of totalResetCount will be read
63              // and we will get right back to here with the increment failing.
64              if(gp.totalResetCount == 0)
65                  FAIL(FATAL_ERROR_INTERNAL);
66
67
68              // Write total reset counter to NV
69              NvWriteReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
70
71              // Reset restartCount
72              gr.restartCount = 0;
73          }
74      }
75
76      return;
77  }
```

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.

```
78  void
79  TimeUpdateToCurrent(void)
80  {
81      UINT64          oldClock;
82      UINT64          elapsed;
83  #define CLOCK_UPDATE_MASK  ((1ULL << NV_CLOCK_UPDATE_INTERVAL)- 1)
84
85      // Can't update time during the dark interval or when rate limiting.
86      if(NvIsAvailable() != TPM_RC_SUCCESS)
```

```
87          return;
88
89      // Save the old clock value
90      oldClock = go.clock;
91
92      // Update the time info to current
93      elapsed = _plat__ClockTimeElapsed();
94      go.clock += elapsed;
95      g_time += elapsed;
96
97      // Check to see if the update has caused a need for an nvClock update
98      // CLOCK_UPDATE_MASK is measured by second, while the value in go.clock is
99      // recorded by millisecond.  Align the clock value to second before the bit
100     // operations
101     if( ((go.clock/1000) | CLOCK_UPDATE_MASK)
102             > ((oldClock/1000) | CLOCK_UPDATE_MASK))
103     {
104         // Going to update the time state so the safe flag
105         // should be set
106         go.clockSafe = YES;
107
108         // Get the DRBG state before updating orderly data
109         CryptDrbgGetPutState(GET_STATE);
110
111         NvWriteReserved(NV_ORDERLY_DATA, &go);
112     }
113
114     // Call self healing logic for dictionary attack parameters
115     DASelfHeal();
116
117     return;
118 }
```

8.9.3.4 TimeSetAdjustRate()

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

```
119 void
120 TimeSetAdjustRate(
121     TPM_CLOCK_ADJUST            adjust          // IN: adjust constant
122 )
123 {
124     switch(adjust)
125     {
126         case TPM_CLOCK_COARSE_SLOWER:
127             _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
128             break;
129         case TPM_CLOCK_COARSE_FASTER:
130             _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
131             break;
132         case TPM_CLOCK_MEDIUM_SLOWER:
133             _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
134             break;
135         case TPM_CLOCK_MEDIUM_FASTER:
136             _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
137             break;
138         case TPM_CLOCK_FINE_SLOWER:
139             _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
140             break;
141         case TPM_CLOCK_FINE_FASTER:
142             _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
143             break;
144         case TPM_CLOCK_NO_CHANGE:
145             break;
```

```
146             default:
147                 pAssert(FALSE);
148                 break;
149         }
150 
151         return;
152     }
```

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

```
153 TPM_RC
154 TimeGetRange(
155     UINT16                offset,         // IN: offset in TPMS_TIME_INFO
156     UINT16                size,           // IN: size of data
157     TIME_INFO            *dataBuffer      // OUT: result buffer
158 )
159 {
160     TPMS_TIME_INFO        timeInfo;
161     UINT16                infoSize;
162     BYTE                  infoData[sizeof(TPMS_TIME_INFO)];
163     BYTE                  *buffer;
164 
165     // Fill TPMS_TIME_INFO structure
166     timeInfo.time = g_time;
167     TimeFillInfo(&timeInfo.clockInfo);
168 
169     // Marshal TPMS_TIME_INFO to canonical form
170     buffer = infoData;
171     infoSize = TPMS_TIME_INFO_Marshal(&timeInfo, &buffer, NULL);
172 
173     // Check if the input range is valid
174     if(offset + size > infoSize) return TPM_RC_RANGE;
175 
176     // Copy info data to output buffer
177     MemoryCopy(dataBuffer, infoData + offset, size, sizeof(TIME_INFO));
178 
179     return TPM_RC_SUCCESS;
180 }
```

8.9.3.6 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```
181 void
182 TimeFillInfo(
183     TPMS_CLOCK_INFO       *clockInfo
184 )
185 {
186     clockInfo->clock = go.clock;
187     clockInfo->resetCount = gp.resetCount;
188     clockInfo->restartCount = gr.restartCount;
189 
190     // If NV is not available, clock stopped advancing and the value reported is
191     // not "safe".
192     if(NvIsAvailable() == TPM_RC_SUCCESS)
```

```
193             clockInfo->safe = go.clockSafe;
194         else
195             clockInfo->safe = NO;
196 
197         return;
198     }
```

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
```

DRAFT

```
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) count = MAX_CAP_ALGS;
126
127       // Compute how many algorithms are defined in s_algorithms array.
128       algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
129
130       // Scan the implemented algorithm list to see if there is a match to 'algID'.
131       for(i = 0; i < algNum; i++)
132       {
133           // If algID is less than the starting algorithm ID, skip it
134           if(s_algorithms[i].algID < algID)
135               continue;
136           if(algList->count < count)
137           {
138               // If we have not filled up the return list, add more algorithms
139               // to it
140               algList->algProperties[algList->count].alg = s_algorithms[i].algID;
141               algList->algProperties[algList->count].algProperties =
142                   s_algorithms[i].attributes;
143               algList->count++;
144           }
145           else
146           {
147               // If the return list is full but we still have algorithms
148               // available, report this and stop scanning.
149               more = YES;
150               break;
151           }
152
153       }
154
155       return more;
156
157   }
```

DRAFT

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

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

9.2.3 BitIsSet()

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

Return Value	Meaning
TRUE	bit is set
FALSE	bit is not set

```
BOOL
BitIsSet(
    unsigned int         bitNum,     // IN: number of the bit in 'bArray'
    BYTE                *bArray,     // IN: array containing the bits
    unsigned int         arraySize   // IN: size in bytes of 'bArray'
)
{
    pAssert(arraySize > (bitNum >> 3));
    return((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
}
```

9.2.4 BitSet()

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

```
void
BitSet(
    unsigned int         bitNum,     // IN: number of the bit in 'bArray'
    BYTE                *bArray,     // IN: array containing the bits
    unsigned int         arraySize   // IN: size in bytes of 'bArray'
)
{
    pAssert(arraySize > bitNum/8);
    bArray[bitNum >> 3] |= (1 << (bitNum & 7));
}
```

9.2.5 BitClear()

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

```
void
BitClear(
    unsigned int         bitNum,     // IN: number of the bit in 'bArray'.
    BYTE                *bArray,     // IN: array containing the bits
```

```
26      unsigned int         arraySize   // IN: size in bytes of 'bArray'
27  )
28  {
29      pAssert(arraySize > bitNum/8);
30      bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
31  }
```

DRAFT

9.3 CommandCodeAttributes.c

9.3.1 Introduction

This file contains the functions for testing various command properties.

9.3.2 Includes and Defines

```
#include     "Tpm.h"
#include     "InternalRoutines.h"
#define     NOT_IMPLEMENTED      0
#define     PP_COMMMAND          (1 << 0)
#define     DECRYPT_4            (1 << 1)
#define     DECRYPT_2            (1 << 2)
#define     HANDLE_1_DUP         (1 << 3)
#define     NO_SESSIONS          (1 << 4)
#define     HANDLE_1_USER        (1 << 5)
#define     ENCRYPT_4            (1 << 6)
#define     IS_IMPLEMENTED       (1 << 7)
#define     ENCRYPT_2            (1 << 8)
#define     HANDLE_1_ADMIN       (1 << 9)
#define     RESPONSE_HANDLE      (1 << 10)
#define     NV_COMMAND           (1 << 11)
#define     HANDLE_2_USER        (1 << 12)
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 inculuded in the specification so that their is no ambiguity about the table containging the information being the normative definition.

```
#include     ".\CommandCodeAttributesTables.c"
```

9.3.3 Functions

9.3.3.1 CommandAuthRole()

This function returns the authorization role required of a handle based on the command code and handle number.

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)
    )
{
    AUTH_ROLE         retVal = AUTH_NONE;
    ATTRIBUTE_TYPE    properties = s_commandAttributes[commandCode - TPM_CC_FIRST];
```

```
27
28      if(handleIndex == 0)
29      {
30          if(0 != (properties & HANDLE_1_USER))
31              retVal = AUTH_USER;
32          else if(0 != (properties & HANDLE_1_ADMIN))
33              retVal = AUTH_ADMIN;
34          else if(0 != (properties & HANDLE_1_DUP))
35              retVal = AUTH_DUP;
36      }
37      else if((handleIndex == 1) && (0 != (properties & HANDLE_2_USER)))
38          retVal = AUTH_USER;
39
40      return retVal;
41  }
```

9.3.3.2 CommandIsImplemented()

This function indicates if a command is implemnted.

Return Value	Meaning
TRUE	if the command is implemented
FALSE	if the commandis not implemented

```
42  BOOL
43  CommandIsImplemented(
44      TPM_CC        commandCode          // IN: command code
45      )
46  {
47      BOOL          implemented;
48
49
50      implemented = (commandCode >= TPM_CC_FIRST) && (commandCode <= TPM_CC_LAST);
51      commandCode -= TPM_CC_FIRST;
52      implemented = implemented
53                    && (    ((s_commandAttributes[commandCode] & IS_IMPLEMENTED))
54                        != 0);
55
56      return implemented;
57  }
```

9.3.3.3 CommandGetAttribute()

This function will return a TPMA_CC structure for the given command code. This function must not be called unless it is known that *commandCode* is for an implemented command. This is used by the GetCapabilities() command.

```
58  TPMA_CC
59  CommandGetAttribute(
60      TPM_CC       commandCode          // IN: command code
61      )
62  {
63      UINT32        size = sizeof(s_ccAttr) / sizeof(s_ccAttr[0]);
64      UINT32        i;
65      TPMA_CC       retVal = {0};
66
67      for(i = 0; i < size; i++)
68      {
69          if(s_ccAttr[i].commandIndex == (UINT16) commandCode)
70          {
71              retVal = s_ccAttr[i];
```

```
72                  break;
73              }
74          }
75          pAssert((UINT16)retVal.commandIndex != 0);
76          return retVal;
77      }
```

9.3.3.4 DecryptSize()

This function returns the size of the decrypt size field. If command parameter decryption is not allowed, the function returns 0.

Return Value	Meaning
0	parameter decryption is not allowed
2	parameter decryptoin is allowed and buffer ia a TPM2B
4	parameter decryption is allowed and buffer is a TPM4B

```
78      int
79      DecryptSize (
80          TPM_CC        commandCode      // IN: commandCode
81          )
82      {
83          // This code is written so that there can be full test coverage even though
84          // there is currently no command that uses a 4 byte size field int he buffer
85          int           retVal = 0;
86
87          commandCode -= TPM_CC_FIRST;
88          if((s_commandAttributes[commandCode] & DECRYPT_2) != 0)
89              retVal = 2;
90          else if((s_commandAttributes[commandCode] & DECRYPT_4) != 0)
91              retVal = 4;
92          return retVal;
93      }
```

9.3.3.5 EncryptSize()

This function returns the size of the encrypt size field. If response parameter decryption is not allowed, the function returns 0.

Return Value	Meaning
0	parameter encryption is not allowed
2	parameter encryptoin is allowed and buffer ia a TPM2B
4	parameter encryption is allowed and buffer is a TPM4B

```
94      int
95      EncryptSize (
96          TPM_CC       commandCode      // IN: commandCode
97          )
98      {
99          // This code is written so that there can be full test coverage even though
100         // there is currently no command that uses a 4 byte size field int he buffer
101         int           retVal = 0;
102
103         commandCode -= TPM_CC_FIRST;
104         if((s_commandAttributes[commandCode] & ENCRYPT_2) != 0)
105             retVal = 2;
106         else if((s_commandAttributes[commandCode] & ENCRYPT_4) != 0)
```

```
107             retVal = 4;
108         return retVal;
109     }
```

9.3.3.6 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

```
110     BOOL
111     IsSessionAllowed(
112         TPM_CC      commandCode     // IN: the command to be checked
113         )
114     {
115         BOOL         retVal;
116
117         commandCode -= TPM_CC_FIRST;
118
119         retVal = (0 == (s_commandAttributes[commandCode] & NO_SESSIONS));
120         return retVal;
121     }
```

9.3.3.7 IsHandleInResponse()

This function indicates if the response has a handle.

This function must not be called if the command is not known to be implemented.

Return Value	Meaning
TRUE	response has a handle
FALSE	response does not have a handle

```
122     BOOL
123     IsHandleInResponse(
124         TPM_CC      commandCode     // IN: the command to be checked
125         )
126     {
127         BOOL         retVal;
128
129         commandCode -= TPM_CC_FIRST;
130
131         retVal = (0 != (s_commandAttributes[commandCode] & RESPONSE_HANDLE));
132         return retVal;
133     }
```

9.4 Commands.c

9.4.1 Description

This file contains the function used by TPM2_GetCapability() to build the list of command code attributes.

9.4.2 Includes

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

9.4.3 CommandCapGetCCList()

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

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

```
TPMI_YES_NO
CommandCapGetCCList(
    TPM_CC            commandCode,       // IN: start command code
    UINT32            count,             // IN: maximum count for number of
    //     entries in 'commandList'
    TPML_CCA         *commandList        // OUT: list of TPMA_CC
)
{
    TPMI_YES_NO       more = NO;
    UINT32            i;

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

    // The maximum count of commands that may be return is MAX_CAP_CC.
    if(count > MAX_CAP_CC) count = MAX_CAP_CC;

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

    // Collect command attributes
    for(i = commandCode; i <= TPM_CC_LAST; i++)
    {
        if(CommandIsImplemented(i))
        {
            if(commandList->count < count)
            {
                // If the list is not full, add the attributes for this command.
                commandList->commandAttributes[commandList->count]
                    = CommandGetAttribute(i);
                commandList->count++;
            }
            else
            {
                // If the list is full but there are more commands to report,
                // indicate this and return.
                more = YES;
                break;
            }
        }
```

```
42          }
43
44          return more;
45
46      }
```

DRAFT

9.5 DRTM.c

9.5.1 Description

This file contains functions that simulate the DRTM events.

9.5.2 Includes

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

9.5.2.1 Signal_Hash_Start()

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

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

9.5.2.2 Signal_Hash_Data()

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

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

9.5.2.3 Signal_Hash_End()

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

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

 Copyright © TCG 2006-2013

9.6 Entity.c

9.6.1 Description

The functions in this file are used for accessing properties for handles of various types. Functions in other files require handles of a specific type but the functions in this file allow use of any handle type.

9.6.2 Includes

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

9.6.3 Functions

9.6.3.1 EntityGetLoadStatus()

This function will indicate if the entity associated with a handle is present in TPM memory. If the handle is a persistent object handle, and the object exists, the persistent object is moved from NV memory into a RAM object slot and the persistent handle is replaced with the transient object handle for the slot.

Error Returns	Meaning
TPM_RC_HANDLE	handle type does not match
TPM_RC_REFERENCE_H0	entity is not present
TPM_RC_HIERARCHY	entity belongs to a disabled hierarchy
TPM_RC_OBJECT_MEMORY	handle is an evict object but there is no space to load it to RAM

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

```
                    break;
                default:
                    // should never see any other permanent handle here
                    pAssert(FALSE);
                    break;
            }
            break;
        case TPM_HT_TRANSIENT:
            // For a transient object, check if the handle is associated
            // with a loaded object.
            if(!ObjectIsPresent(*handle))
                result = TPM_RC_REFERENCE_H0;
            break;
        case TPM_HT_PERSISTENT:
            // Persistent object
            // Copy the persistent object to RAM and replace the handle with the
            // handle of the assigned slot.  A TPM_RC_OBJECT_MEMORY,
            // TPM_RC_HIERARCHY or TPM_RC_HANDLE error may be returned by
            // ObjectLoadEvict()
            result = ObjectLoadEvict(handle, commandCode);
            break;
        case TPM_HT_HMAC_SESSION:
            // For an HMAC session, see if the session is loaded
            // and if the session in the session slot is actually
            // an HMAC session.
            if(SessionIsLoaded(*handle))
            {
                SESSION             *session;
                session = SessionGet(*handle);
                // Check if the session is a HMAC session
                if(session->attributes.isPolicy == SET)
                    result = TPM_RC_HANDLE;
            }
            else
                result = TPM_RC_REFERENCE_H0;
            break;
        case TPM_HT_POLICY_SESSION:
            // For a policy session, see if the session is loaded
            // and if the session in the session slot is actually
            // a policy session.
            if(SessionIsLoaded(*handle))
            {
                SESSION             *session;
                session = SessionGet(*handle);
                // Check if the session is a policy session
                if(session->attributes.isPolicy == CLEAR)
                    result = TPM_RC_HANDLE;
            }
            else
                result = TPM_RC_REFERENCE_H0;
            break;
        case TPM_HT_NV_INDEX:
            // For an NV Index, use the platform-specific routine
            // to search the IN Index space.
            result = NvIndexIsAccessible(*handle);
            break;
        case TPM_HT_PCR:
            // Any PCR handle that is unmarshaled successfully referenced
            // a PCR that is defined.
            break;
        default:
            // Any other handle type is a defect in the unmarshaling code.
            pAssert(FALSE);
            break;
    }
    return result;
```

```
100 }
```

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*.

```
101 UINT16
102 EntityGetAuthValue(
103     TPMI_DH_ENTITY   handle,          // IN: handle of entity
104     AUTH_VALUE      *auth             // OUT: authValue of the entity
105 )
106 {
107     TPM2B_AUTH       authValue = {0};
108
109     switch(HandleGetType(handle))
110     {
111         case TPM_HT_PERMANENT:
112             switch(handle)
113             {
114                 case TPM_RH_OWNER:
115                     // ownerAuth for TPM_RH_OWNER
116                     authValue = gp.ownerAuth;
117                     break;
118                 case TPM_RH_ENDORSEMENT:
119                     // endorsementAuth for TPM_RH_ENDORSEMENT
120                     authValue = gp.endorsementAuth;
121                     break;
122                 case TPM_RH_PLATFORM:
123                     // platformAuth for TPM_RH_PLATFORM
124                     authValue = gc.platformAuth;
125                     break;
126                 case TPM_RH_LOCKOUT:
127                     // lockoutAuth for TPM_RH_LOCKOUT
128                     authValue = gp.lockoutAuth;
129                     break;
130                 case TPM_RH_NULL:
131                     // nullAuth for TPM_RH_NULL. Return 0 directly here
132                     return 0;
133                     break;
134                 default:
135                     // If any other permanent handle is present it is
136                     // a code defect.
137                     pAssert(FALSE);
138                     break;
139             }
140             break;
141         case TPM_HT_TRANSIENT:
142             // authValue for an object
143             // A persistent object would have been copied into RAM
144             // and would have an transient object handle here.
145             {
146                 OBJECT          *object;
147                 object = ObjectGet(handle);
148                 // special handling if this is a sequence object
149                 if(ObjectIsSequence(object))
150                 {
151                     authValue = ((HASH_OBJECT *)object)->auth;
```

```
152                 }
153                 else
154                 {
155                     // Auth value is available only when the private portion of
156                     // the object is loaded.  The check should be made before
157                     // this function is called
158                     pAssert(object->attributes.publicOnly == CLEAR);
159                     authValue = object->sensitive.authValue;
160                 }
161             }
162             break;
163         case TPM_HT_NV_INDEX:
164             // authValue for an NV index
165             {
166                 NV_INDEX        nvIndex;
167                 NvGetIndexInfo(handle, &nvIndex);
168                 authValue = nvIndex.authValue;
169             }
170             break;
171         case TPM_HT_PCR:
172             // authValue for PCR
173             PCRGetAuthValue(handle, &authValue);
174             break;
175         default:
176             // If any other handle type is present here, then there is a defect
177             // in the unmarshaling code.
178             pAssert(FALSE);
179             break;
180     }
181
182     // Copy the authValue
183     pAssert(authValue.t.size <= <K>sizeof(authValue.t.buffer));
184     MemoryCopy(auth, authValue.t.buffer, authValue.t.size, sizeof(TPMU_HA));
185
186     return authValue.t.size;
187 }
```

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.

```
188 TPMI_ALG_HASH
189 EntityGetAuthPolicy(
190     TPMI_DH_ENTITY  handle,         // IN: handle of entity
191     TPM2B_DIGEST   *authPolicy      // OUT: authPolicy of the entity
192 )
193 {
194     TPMI_ALG_HASH       hashAlg = TPM_ALG_NULL;
195
196     switch(HandleGetType(handle))
197     {
198         case TPM_HT_PERMANENT:
199             switch(handle)
200             {
201                 case TPM_RH_OWNER:
202                     // ownerPolicy for TPM_RH_OWNER
203                     *authPolicy = gp.ownerPolicy;
```

```
                    hashAlg = gp.ownerAlg;
                    break;
                case TPM_RH_ENDORSEMENT:
                    // endorsementPolicy for TPM_RH_ENDORSEMENT
                    *authPolicy = gp.endorsementPolicy;
                    hashAlg = gp.endorsementAlg;
                    break;
                case TPM_RH_PLATFORM:
                    // platformPolicy for TPM_RH_PLATFORM
                    *authPolicy = gc.platformPolicy;
                    hashAlg = gc.platformAlg;
                    break;
                default:
                    // If any other permanent handle is present it is
                    // a code defect.
                    pAssert(FALSE);
                    break;
            }
            break;
        case TPM_HT_TRANSIENT:
            // authPolicy for an object
            {
                OBJECT *object = ObjectGet(handle);
                *authPolicy = object->publicArea.authPolicy;
                hashAlg = object->publicArea.nameAlg;
            }
            break;
        case TPM_HT_NV_INDEX:
            // authPolicy for a NV index
            {
                NV_INDEX        nvIndex;
                NvGetIndexInfo(handle, &nvIndex);
                *authPolicy = nvIndex.publicArea.authPolicy;
                hashAlg = nvIndex.publicArea.nameAlg;
            }
            break;
        case TPM_HT_PCR:
            // authPolicy for a PCR
            hashAlg = PCRGetAuthPolicy(handle, authPolicy);
            break;
        default:
            // If any other handle type is present it is a code defect.
            pAssert(FALSE);
            break;
    }
    return hashAlg;
}
```

9.6.3.4 EntityGetName()

This function returns the Name associated with a handle. It will set *name* to the Name and return the size of the Name string.

```
UINT16
EntityGetName(
    TPMI_DH_ENTITY  handle,     // IN: handle of entity
    NAME           *name        // OUT: name of entity
)
{
    UINT16          nameSize;

    switch(HandleGetType(handle))
    {
        case TPM_HT_TRANSIENT:
```

```
262             // Name for an object
263             nameSize = ObjectGetName(handle, name);
264             break;
265         case TPM_HT_NV_INDEX:
266             // Name for a NV index
267             nameSize = NvGetName(handle, name);
268             break;
269         default:
270             // For all other types, the handle is the Name
271             nameSize = TPM_HANDLE_Marshal(&handle, (BYTE **)&name, NULL);
272             break;
273     }
274     return nameSize;
275 }
```

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.

```
276 TPMI_RH_HIERARCHY
277 EntityGetHierarchy(
278     TPMI_DH_ENTITY  handle      // IN :handle of entity
279 )
280 {
281     TPMI_RH_HIERARCHY       hierarcy = TPM_RH_NULL;
282 
283     switch(HandleGetType(handle))
284     {
285         case TPM_HT_PERMANENT:
286             // hierarchy for a permanent handle
287             switch(handle)
288             {
289                 case TPM_RH_PLATFORM:
290                 case TPM_RH_ENDORSEMENT:
291                 case TPM_RH_NULL:
292                     hierarcy = handle;
293                     break;
294                 // all other permanent handles are associated with the owner
295                 // hierarcchy. (should only be TPM_RH_OWNER and TMP_RH_LOCKOUT)
296                 default:
297                     hierarcy = TPM_RH_OWNER;
298                     break;
299             }
300             break;
301         case TPM_HT_NV_INDEX:
302             // hierarchy for NV index
303             {
304                 NV_INDEX        nvIndex;
305                 NvGetIndexInfo(handle, &nvIndex);
306                 // If only the platform can delete the index, then it is
307                 // considered to be in the platform hierarchy, otherwise it
308                 // is in the owner hierarchy.
309                 if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
310                     hierarcy = TPM_RH_PLATFORM;
311                 else
312                     hierarcy = TPM_RH_OWNER;
313             }
```

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

DRAFT

9.7 Global.c

9.7.1 Description

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

9.7.2 Includes and Defines

```
#define GLOBAL_C
#include "InternalRoutines.h"
```

9.7.3 Global Data Values

These values are visible across multiple modules.

```
BOOL                    g_phEnable;
const UINT16            g_rcIndex[15] = {TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,
                                        TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,
                                        TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,
                                        TPM_RC_D, TPM_RC_E, TPM_RC_F
                                       };
TPM_HANDLE              g_exclusiveAuditSession;
UINT64                  g_time;
BOOL                    g_pcrReConfig;
TPMI_DH_OBJECT          g_DRTMHandle;
BOOL                    g_DrtmPreStartup;
BOOL                    g_clearOrderly;
TPM_SU                  g_prevOrderlyState;
BOOL                    g_updateNV;
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

```
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 = UNDEFINED_INDEX;
UINT32                  s_decryptSessionIndex = UNDEFINED_INDEX;
UINT32                  s_auditSessionIndex = UNDEFINED_INDEX;
TPM2B_DIGEST            s_cpHashForAudit;
UINT32                  s_sessionNum;
BOOL                    s_DAPendingOnNV = FALSE;
#ifdef TPM_CC_GetCommandAuditDigest
TPM2B_DIGEST            s_cpHashForCommandAudit;
#endif
```

9.7.4.2 DA.c

```
UINT64                  s_selfHealTimer = 0;
UINT64                  s_lockoutTimer = 0;
```

9.7.4.3 NV.c

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

9.7.4.4 Object.c

```
47  OBJECT_SLOT         s_objects[MAX_LOADED_OBJECTS];
```

9.7.4.5 PCR.c

```
48  PCR                 s_pcrs[IMPLEMENTATION_PCR];
```

9.7.4.6 Session.c

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

9.7.4.7 Manufacture.c

```
52  BOOL                s_manufactured = FALSE;
```

9.7.4.8 Power.c

```
53  BOOL                s_initialized = FALSE;
```

9.7.4.9 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.

```
54  UINT32   s_actionInputBuffer[1024];          // action input buffer
55  UINT32   s_actionOutputBuffer[1024];         // action output buffer
56  BYTE     s_responseBuffer[MAX_RESPONSE_SIZE];// response buffer
```

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 funciton 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 <= TPM_RH_LAST)
        return inHandle;
    return 0;
}
```

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

```
TPMI_YES_NO
PermanentCapGetHandles(
    TPM_HANDLE              handle,         // IN: start handle
```

```
23      UINT32                  count,          // IN: count of returned handles
24      TPML_HANDLE             *handleList     // OUT: list of handle
25  )
26  {
27      TPMI_YES_NO     more = NO;
28      UINT32          i;
29
30      pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
31
32      // Initialize output handle list
33      handleList->count = 0;
34
35      // The maximum count of handles we may return is MAX_CAP_HANDLES
36      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
37
38      // Iterate permanent handle range
39      for(i = NextPermanentHandle(handle);
40              i != 0; i = NextPermanentHandle(i+1))
41      {
42          if(handleList->count < count)
43          {
44              // If we have not filled up the return list, add this permanent
45              // handle to it
46              handleList->handle[handleList->count] = i;
47              handleList->count++;
48          }
49          else
50          {
51              // If the return list is full but we still have permanent handle
52              // available, report this and stop iterating
53              more = YES;
54              break;
55          }
56      }
57      return more;
58  }
```

DRAFT

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"
```

9.10.3 Functions

9.10.3.1 TPM_Manufacture()

This function initializes the TPM values in preparation for the TPM's first use. This function will fail if previously called. The TPM can be remanufactured by calling TPM_Teardown() first and then calling this function again.

Return Value	Meaning
0	success
1	manufacturing process previously performed

```
int
TPM_Manufacture(void)
{
    TPM_SU          orderlyShutdown;
    UINT64          totalResetCount = 0;

    // If TPM has been manufactured, return indication.
    if(s_manufactured)
        return 1;

    // initialize crypto units
    //CryptInitUnits();

    // initialize NV
    NvInit();

#ifdef _DRBG_STATE_SAVE
    // Initialize the drbg. This needs to come before the install
    // of the hierarchies
    if(!_cpri__Startup())               // Have to start the crypto units first
        FAIL(FATAL_ERROR_INTERNAL);
    _cpri__DrbgGetPutState(PUT_STATE, 0, NULL);
#endif

    // default configuration for PCR
    PCRSimStart();

    // initialize pre-installed hierarchy data
    // This should happen after NV is initialized because hierarchy data is
    // stored in NV.
    HierarchyPreInstall_Init();

    // initialize dictionary attack parameters
    DAPreInstall_Init();
```

 Copyright © TCG 2006-2013

```
37
38      // initialize PP list
39      PhysicalPresencePreInstall_Init();
40
41      // initialize command audit list
42      CommandAuditPreInstall_Init();
43
44      // first start up is required to be Startup(CLEAR)
45      orderlyShutdown = TPM_SU_CLEAR;
46      NvWriteReserved(NV_ORDERLY, &orderlyShutdown);
47
48      // initialize the firmware version
49      gp.firmwareV1 = FIRMWARE_V1;
50  #ifdef FIRMWARE_V2
51      gp.firmwareV2 = FIRMWARE_V2;
52  #else
53      gp.firmwareV2 = 0;
54  #endif
55      NvWriteReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
56      NvWriteReserved(NV_FIRMWARE_V2, &gp.firmwareV2);
57
58      // initialize the total reset counter to 0
59      NvWriteReserved(NV_TOTAL_RESET_COUNT, &totalResetCount);
60
61      // initialize the clock stuff
62      go.clock = 0;
63      go.clockSafe = YES;
64
65  #ifdef _DRBG_STATE_SAVE
66      // initialize the current DRBG state in NV
67
68      _cpri__DrbgGetPutState(GET_STATE, sizeof(go.drbgState), (BYTE *)&go.drbgState);
69  #endif
70
71      NvWriteReserved(NV_ORDERLY_DATA, &go);
72
73      // Commit NV writes.  Manufacture process is an artificial process existing
74      // only in simulator environment and it is not defined in the specification
75      // that what should be the expected behavior if the NV write fails at this
76      // point.  Therefore, it is assumed the NV write here is always success and
77      // no return code of this function is checked.
78      NvCommit();
79
80      s_manufactured = TRUE;
81
82      return 0;
83  }
```

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

```
84  int
```

```
85  TPM_TearDown(void)
86  {
87      // if TPM has not been manufactured, return indication
88      if(!s_manufactured)
89          return 1;
90
91      // stop crypt units
92      CryptStopUnits();
93
94      s_manufactured = FALSE;
95      return 0;
96  }
```

DRAFT

9.11 Marshal.c

9.11.1 Introduction

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

The marshaling and unmarshaling code and function prototypes are not listed, as the code is repetitive, long, and not very useful to read. Examples of the a few unmarshaling routines are provided. Most of the others are similar.

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

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets ("<>") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

9.11.2 Unmarshal and Marshal a Value

In part 2, a TPMI_DI_OBJECT is defined by this table:

Table xxx — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```
1   TPM_RC
2   TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
3                            bool flag)
4   {
5       TPM_RC    result;
6       result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       if (*target == TPM_RH_NULL) {
10          if(flag)
11              return TPM_RC_SUCCESS;
12          else
13              return TPM_RC_VALUE;
14      }
15      if((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
16          if((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST))
17              return TPM_RC_VALUE;
18      return TPM_RC_SUCCESS;
19  }
```

and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```
1   UINT16
2   TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
```

```
{
    return UINT32_Marshal((UINT32 *)source, buffer, size);
}
```

9.11.3 Unmarshal and Marshal a Union

In part 2, a TPMU_PUBLIC_PARMS union is defined by:

Table 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:

```
UINT16
```

```
TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
                          UINT32 selector)
{
    switch(selector) {
#ifdef TPM_ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Marshal(
                        (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
#endif
#ifdef TPM_ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Marshal(
                        (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
#endif
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Marshal(
                         (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
#endif
#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Marshal(
                          (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
#endif
    }
    assert(1);
    return 0;
}
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

9.11.4 Unmarshal and Marshal a Structure

In part 2, the TPMT_PUBLiC structure is defined by:

Table 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  }
```

 Copyright © TCG 2006-2013

The marshaling code for the TPMT_PUBLIC structure is:

```
1   UINT16
2   TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
3   {
4       UINT16    result = 0;
5       result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
6                                      (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
7       result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
8                                     (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
9   ;
10      result = (UINT16)(result + TPMA_OBJECT_Marshal(
11                               (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
12
13      result = (UINT16)(result + TPM2B_DIGEST_Marshal(
14                                  (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));
15
16      result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
17                               (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
18                                                                  (UINT32)source->type));
19
20      result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
21                                     (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
22                                                                  (UINT32)source->type));
23
24      return result;
25  }
```

9.11.5 Unmarshal and Marshal an Array

In part 2, the TPML_DIGEST is defined by:

Table xxx — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```
1   TPM_RC
2   TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8
9       if( (target->count < 2))     // This check is triggered by the {2:} notation
10                                   // on 'count'
11          return TPM_RC_SIZE;
12
13      if((target->count) > 8)      // This check is triggered by the {:8} notation
14                                   // on 'digests'.
15          return TPM_RC_SIZE;
16
17      result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
18                                            buffer, size, (INT32)(target->count));
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      return TPM_RC_SUCCESS;
23  }
```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```
1   TPM_RC
2   TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                                INT32 count)
4   {
5       TPM_RC    result;
6       INT32 i;
7       for(i = 0; i < count; i++) {
8           result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9           if(result != TPM_RC_SUCCESS)
10              return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
14
```

 Copyright © TCG 2006-2013

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
1   UINT16
2   TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
3   {
4       UINT16    result = 0;
5       result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
6                                                                                 size));
7       result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
8                                       (TPM2B_DIGEST *)(source->digests), buffer, size,
9                                       (INT32)(source->count)));
10
11      return result;
12  }
```

The marshaling code for the array is:

```
1   TPM_RC
2   TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                                INT32 count)
4   {
5       TPM_RC    result;
6       INT32 i;
7       for(i = 0; i < count; i++) {
8           result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9           if(result != TPM_RC_SUCCESS)
10              return result;
11      }
12      return TPM_RC_SUCCESS;
13  }
```

9.11.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 9.11.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

Table xxx — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

```
TPM_RC
TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    // if size equal to 0, the rest of the structure is a zero buffer.  Stop
 processing
    if(target->t.size == 0)
        return TPM_RC_SUCCESS;

    if((target->t.size) > 1024)     // This check is triggered by the {:1024} notation
                                    // on 'buffer'
        return TPM_RC_SIZE;

    result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
                                  (INT32)(target->t.size));
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```

Which use these structure definitions:

```
typedef struct {
    UINT16        size;
    BYTE          buffer[1];
} TPM2B;

typedef struct {
    UINT16        size;
    BYTE          buffer[1024];
} EVENT_2B;

typedef union {
    EVENT_2B      t;    // The type-specific union member
    TPM2B         b;    // The generic union member
} TPM2B_EVENT;
```

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

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 funciton 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.

```
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  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.

```
62  INT16
63  MemoryCopy2B(
64      TPM2B         *dest,      // OUT: receiving TPM2B
65      const TPM2B   *source,    // IN: source TPM2B
66      UINT16         dSize      // IN: size of the receiving buffer
67  )
68  {
69      dest->size = source->size;
70      MemoryMove(dest->buffer, source->buffer, dest->size, dSize);
71      return dest->size;
72  }
```

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)*).

```
73  void
74  MemoryConcat2B(
75      TPM2B   *aInOut,    // IN/OUT: destination 2B
76      TPM2B   *bIn,       // IN: second 2B
77      UINT16   aSize      // IN: The size of aInOut.buffer
78                          //     (max values for aInOut.size)
79  )
80  {
81      MemoryMove(&aInOut->buffer[aInOut->size],
82                 bIn->buffer,
83                 bIn->size,
84                 aSize - aInOut->size);
85      aInOut->size = aInOut->size + bIn->size;
86      return;
87  }
```

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

```
88  BOOL
89  Memory2BEqual(
90      const TPM2B       *aIn,     // IN: compare value
91      const TPM2B       *bIn      // IN: compare value
92  )
93  {
94      if(aIn->size != bIn->size)
95          return FALSE;
96
97      return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
98  }
```

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 possiblity that the caller will inadvertentl run over the end of the buffer.

```
 99  void
100  MemorySet(
101      void            *destination,       // OUT: memory destination
102      char             value,             // IN: fill value
103      UINT32           size               // IN: number of octets to fill
104  )
105  {
106      char *p = (char *)destination;
107      while (size--)
108          *p++ = value;
109      return;
110  }
```

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.

```
111  BYTE *
112  MemoryGetActionInputBuffer(
113      UINT32      size        // Size, in bytes, required for the input unmarshaling
114  )
115  {
116      BYTE        *buf = NULL;
117
118      if(size > 0)
119      {
120          // In this implementation, a static buffer is set aside for action output.
121          // Other implementations may apply additional optimization based on command
122          // code or other factors.
123          UINT32      *p = s_actionInputBuffer;
124          buf = (BYTE *)p;
125          pAssert(size < <K>sizeof(s_actionInputBuffer));
126
127          // size of an element in the buffer
128  #define SZ      sizeof(s_actionInputBuffer[0])
129
130          for(size = (size + SZ - 1) / SZ; size > 0; size--)
131              *p++ = 0;
132  #undef SZ
133      }
134      return buf;
135  }
```

9.12.3.9 MemoryGetActionOutputBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```
136  void *
137  MemoryGetActionOutputBuffer(
138      TPM_CC      command                 // Command that requires the buffer
139  )
140  {
141      // In this implementation, a static buffer is set aside for action output.
142      // Other implementations may apply additional optimization based on the command
143      // code or other factors.
144      command = 0;        // Unreferenced parameter
145      return s_actionOutputBuffer;
```

```
146 }
```

9.12.3.10 MemoryGetResponseBuffer()

This function returns the address into which the command response is marshaled from values in the action output buffer.

```
147 BYTE *
148 MemoryGetResponseBuffer(
149     TPM_CC      command                 // Command that requires the buffer
150 )
151 {
152     // In this implementation, a static buffer is set aside for responses.
153     // Other implementation may apply additional optimization based on the command
154     // code or other factors.
155     command = 0;        // Unreferenced parameter
156     return s_responseBuffer;
157 }
```

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.

```
158 UINT16
159 MemoryRemoveTrailingZeros (
160     TPM2B_AUTH      *auth        // IN/OUT: value to adjust
161 )
162 {
163     BYTE        *a = &auth->t.buffer[auth->t.size-1];
164     for(; auth->t.size > 0; auth->t.size--)
165     {
166         if(*a--)
167             break;
168     }
169     return auth->t.size;
170 }
```

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

```
#define POWER_C
#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);
```

```
38              break;
39          case TPM_PT_VENDOR_STRING_2:
40              // second four characters of the vendor ID string
41  #ifdef VENDOR_STRING_2
42              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
43  #else
44              *value = 0;
45  #endif
46              break;
47          case TPM_PT_VENDOR_STRING_3:
48              // third four characters of the vendor ID string
49  #ifdef VENDOR_STRING_3
50              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
51  #else
52              *value = 0;
53  #endif
54              break;
55          case TPM_PT_VENDOR_STRING_4:
56              // fourth four characters of the vendor ID string
57  #ifdef VENDOR_STRING_4
58              *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
59  #else
60              *value = 0;
61  #endif
62              break;
63          case TPM_PT_VENDOR_TPM_TYPE:
64              // vendor-defined value indicating the TPM model
65              *value = 1;
66              break;
67          case TPM_PT_FIRMWARE_VERSION_1:
68              // more significant 32-bits of a vendor-specific value
69              *value = gp.firmwareV1;
70              break;
71          case TPM_PT_FIRMWARE_VERSION_2:
72              // less significant 32-bits of a vendor-specific value
73              *value = gp.firmwareV2;
74              break;
75          case TPM_PT_INPUT_BUFFER:
76              // maximum size of TPM2B_MAX_BUFFER
77              *value = MAX_DIGEST_BUFFER;
78              break;
79          case TPM_PT_HR_TRANSIENT_MIN:
80              // minimum number of transient objects that can be held in TPM
81              // RAM
82              *value = MAX_LOADED_OBJECTS;
83              break;
84          case TPM_PT_HR_PERSISTENT_MIN:
85              // minimum number of persistent objects that can be held in
86              // TPM NV memory
87              // In this implementation, there is no minimum number of
88              // persistent objects.
89              *value = MIN_EVICT_OBJECTS;
90              break;
91          case TPM_PT_HR_LOADED_MIN:
92              // minimum number of authorization sessions that can be held in
93              // TPM RAM
94              *value = MAX_LOADED_SESSIONS;
95              break;
96          case TPM_PT_ACTIVE_SESSIONS_MAX:
97              // number of authorization sessions that may be active at a time
98              *value = MAX_ACTIVE_SESSIONS;
99              break;
100         case TPM_PT_PCR_COUNT:
101             // number of PCR implemented
102             *value = IMPLEMENTATION_PCR;
103             break;
```

```
104             case TPM_PT_PCR_SELECT_MIN:
105                 // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
106                 *value = PCR_SELECT_MIN;
107                 break;
108             case TPM_PT_CONTEXT_GAP_MAX:
109                 // maximum allowed difference (unsigned) between the contextID
110                 // values of two saved session contexts
111                 *value = (1 << (<K>sizeof(CONTEXT_SLOT) * 8)) - 1;
112                 break;
113             case TPM_PT_NV_COUNTERS_MAX:
114                 // maximum number of NV indexes that are allowed to have the
115                 // TPMA_NV_COUNTER attribute SET
116                 // In this implementation, there is no limitation on the number
117                 // of counters, except for the size of the NV Index memory.
118                 *value = 0;
119                 break;
120             case TPM_PT_NV_INDEX_MAX:
121                 // maximum size of an NV index data area
122                 *value = MAX_NV_INDEX_SIZE;
123                 break;
124             case TPM_PT_MEMORY:
125                 // a TPMA_MEMORY indicating the memory management method for the TPM
126                 {
127                     TPMA_MEMORY         attributes = {0};
128                     attributes.sharedNV = SET;
129                     attributes.objectCopiedToRam = SET;
130 
131                     // Note: Different compilers may require a different method to cast
132                     // a bit field structure to a UINT32.
133                     *value = * (UINT32 *) &attributes;
134                     break;
135                 }
136             case TPM_PT_CLOCK_UPDATE:
137                 // interval, in seconds, between updates to the copy of
138                 // TPMS_TIME_INFO .clock in NV
139                 *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
140                 break;
141             case TPM_PT_CONTEXT_HASH:
142                 // algorithm used for the integrity hash on saved contexts and
143                 // for digesting the fuData of TPM2_FirmwareRead()
144                 *value = CONTEXT_INTEGRITY_HASH_ALG;
145                 break;
146             case TPM_PT_CONTEXT_SYM:
147                 // algorithm used for encryption of saved contexts
148                 *value = CONTEXT_ENCRYPT_ALG;
149                 break;
150             case TPM_PT_CONTEXT_SYM_SIZE:
151                 // size of the key used for encryption of saved contexts
152                 *value = CONTEXT_ENCRYPT_KEY_BITS;
153                 break;
154             case TPM_PT_ORDERLY_COUNT:
155                 // maximum difference between the volatile and non-volatile
156                 // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
157                 *value = MAX_ORDERLY_COUNT;
158                 break;
159             case TPM_PT_MAX_COMMAND_SIZE:
160                 // maximum value for 'commandSize'
161                 *value = MAX_COMMAND_SIZE;
162                 break;
163             case TPM_PT_MAX_RESPONSE_SIZE:
164                 // maximum value for 'responseSize'
165                 *value = MAX_RESPONSE_SIZE;
166                 break;
167             case TPM_PT_MAX_DIGEST:
168                 // maximum size of a digest that can be produced by the TPM
169                 *value = sizeof(TPMU_HA);
```

```
170             break;
171         case TPM_PT_MAX_OBJECT_CONTEXT:
172             // maximum size of a TPMS_CONTEXT that will be returned by
173             // TPM2_ContextSave for object context
174             *value = 0;
175 
176             // adding sequence, saved handle and hierarchy
177             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
178                       sizeof(TPMI_RH_HIERARCHY);
179             // add size field in TPM2B_CONTEXT
180             *value += sizeof(UINT16);
181 
182             // add integrity hash size
183             *value += sizeof(UINT16) +
184                       CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
185 
186             // Add fingerprint size, which is the same as sequence size
187             *value += sizeof(UINT64);
188 
189             // Add OBJECT structure size
190             *value += sizeof(OBJECT);
191             break;
192         case TPM_PT_MAX_SESSION_CONTEXT:
193             // the maximum size of a TPMS_CONTEXT that will be returned by
194             // TPM2_ContextSave for object context
195             *value = 0;
196 
197             // adding sequence, saved handle and hierarchy
198             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
199                       sizeof(TPMI_RH_HIERARCHY);
200             // Add size field in TPM2B_CONTEXT
201             *value += sizeof(UINT16);
202 
203             // Add integrity hash size
204             *value += sizeof(UINT16) +
205                       CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
206             // Add fingerprint size, which is the same as sequence size
207             *value += sizeof(UINT64);
208 
209             // Add SESSION structure size
210             *value += sizeof(SESSION);
211             break;
212         case TPM_PT_PS_FAMILY_INDICATOR:
213             // platform specific values for the TPM_PT_PS parameters from
214             // the relevant platform-specific specification
215             // In this reference implementation, all of these values are 0.
216             *value = 0;
217             break;
218         case TPM_PT_PS_LEVEL:
219             // level of the platform-specific specification
220             *value = 0;
221             break;
222         case TPM_PT_PS_REVISION:
223             // specification Revision times 100 for the platform-specific
224             // specification
225             *value = 0;
226             break;
227         case TPM_PT_PS_DAY_OF_YEAR:
228             // platform-specific specification day of year using TCG calendar
229             *value = 0;
230             break;
231         case TPM_PT_PS_YEAR:
232             // platform-specific specification year using the CE
233             *value = 0;
234             break;
235         case TPM_PT_SPLIT_MAX:
```

DRAFT

```
236             // number of split signing operations supported by the TPM
237             *value = 0;
238     #ifdef TPM_ALG_ECDAA
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                             //     properties
```

```
419         TPML_TAGGED_TPM_PROPERTY    *propertyList   // OUT: property list
420     )
421     {
422         TPMI_YES_NO     more = NO;
423         UINT32          i;
424
425         // initialize output property list
426         propertyList->count = 0;
427
428         // maximum count of properties we may return is MAX_PCR_PROPERTIES
429         if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;
430
431         // If property is less than PT_FIXED, start from PT_FIXED.
432         if(property < PT_FIXED) property = PT_FIXED;
433
434         // Scan through the TPM properties of the requested group.
435         // The size of TPM property group is PT_GROUP * 2 for fix and
436         // variable groups.
437         for(i = property; i <= PT_FIXED + PT_GROUP * 2; i++)
438         {
439             UINT32          value;
440             if(TPMPropertyIsDefined((TPM_PT) i, &value))
441             {
442                 if(propertyList->count < count)
443                 {
444
445                     // If the list is not full, add this property
446                     propertyList->tpmProperty[propertyList->count].property =
447                         (TPM_PT) i;
448                     propertyList->tpmProperty[propertyList->count].value = value;
449                     propertyList->count++;
450                 }
451                 else
452                 {
453                     // If the return list is full but there are more properties
454                     // available, set the indication and exit the loop.
455                     more = YES;
456                     break;
457                 }
458             }
459         }
460         return more;
461     }
```

DRAFT

Cryptographic Functions

9.15 Introduction

The files in this section provide cryptographic support for the other functions in the TPM and the interface to the Crypto Engine.

9.16 CryptUtil.c

9.16.1 Introduction

This module contains the interfaces to the CryptoEngine() and provides miscellaneous cryptographic functions in support of the TPM.

9.16.2 Includes

```
1   #include     "TPM_Types.h"
2   #include     "CryptoEngine.h"     // types shared by CryptUtil and CryptoEngine.
3                                     // Includes the function prototypes for the
4                                     // CryptoEngine functions.
5   #include     "Global.h"
6   #include     "InternalRoutines.h"
7   #include     "MemoryLib_fp.h"
```

9.16.3 TranslateCryptErrors()

This function converts errors from the cryptographic library into TPM_RC_VALUES.

Error Returns	Meaning
TPM_RC_VALUE	CRYPT_FAIL
TPM_RC_NO_RESULT	CRYPT_NO_RESULT
TPM_RC_SCHEME	CRYPT_SCHEME
TPM_RC_VALUE	CRYPT_PARAMETER
TPM_RC_SIZE	CRYPT_UNDERFLOW
TPM_RC_ECC_POINT	CRYPT_POINT
TPM_RC_CANCELLED	CRYPT_CANCEL

```
8   static TPM_RC
9   TranslateCryptErrors (
10      CRYPT_RESULT         retVal              // IN: crypt error to evaluate
11  )
12  {
13      switch (retVal)
14      {
15      case CRYPT_SUCCESS:
16          return TPM_RC_SUCCESS;
17      case CRYPT_FAIL:
18          return TPM_RC_VALUE;
19      case CRYPT_NO_RESULT:
20          return TPM_RC_NO_RESULT;
21      case CRYPT_SCHEME:
22          return TPM_RC_SCHEME;
23      case CRYPT_PARAMETER:
```

```
24          return TPM_RC_VALUE;
25      case CRYPT_UNDERFLOW:
26          return TPM_RC_SIZE;
27      case CRYPT_POINT:
28          return TPM_RC_ECC_POINT;
29      case CRYPT_CANCEL:
30          return TPM_RC_CANCELED;
31      default: // Other unknown warnings
32          return TPM_RC_FAILURE;
33      }
34  }
```

9.16.4 Random Number Generation Functions

```
35  #ifdef TPM_ALG_NULL //%
36  #ifdef _DRBG_STATE_SAVE //%
```

9.16.4.1 CryptDrbgGetPutState()

Read or write the current state from the DRBG in the *cryptoEngine*.

```
37  void
38  CryptDrbgGetPutState(
39      GET_PUT             direction             // IN: Get from or put to DRBG
40  )
41  {
42      _cpri__DrbgGetPutState(direction,
43                             sizeof(go.drbgState),
44                             (BYTE *)&go.drbgState);
45  }
46  #else   //% 00
47  //%#define CryptDrbgGetPutState(ignored)  // If not doing state save, turn this
48  //%                                       // into a null macro
49  #endif  //%
```

9.16.4.2 CryptStirRandom()

Stir random entropy

```
50  void
51  CryptStirRandom(
52      UINT32               entropySize,       // IN: size of entropy buffer
53      BYTE                *buffer             // IN: entropy buffer
54  )
55  {
56      // RNG self testing code may be inserted here
57
58      // Call crypto engine random number stirring function
59      _cpri__StirRandom(entropySize, buffer);
60
61      return;
62  }
```

9.16.4.3 CryptGenerateRandom()

This is the interface to _cpri__GenerateRandom().

```
63  UINT16
64  CryptGenerateRandom(
65      UINT16               randomSize,        // IN: size of random number
```

```
66      BYTE                *buffer             // OUT: buffer of random number
67  )
68  {
69      // Call crypto engine random number generation
70      return _cpri__GenerateRandom(randomSize, buffer);
71  }
72  #endif //TPM_ALG_NULL //%
```

9.16.5 Hash/HMAC Functions

9.16.5.1 CryptGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```
73  #ifdef TPM_ALG_KEYEDHASH        //% 1
74  TPM_ALG_ID
75  CryptGetContextAlg(
76      void                *state              // IN: the context to check
77  )
78  {
79      HASH_STATE  *context = (HASH_STATE *)state;
80      return _cpri__GetContextAlg(&context->state);
81  }
```

9.16.5.2 CryptStartHash()

This function starts a hash and return the size, in bytes, of the digest.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
82  UINT16
83  CryptStartHash(
84      TPMI_ALG_HASH        hashAlg,           // IN: hash algorithm
85      HASH_STATE          *hashState          // OUT: the state of hash stack. It
86                                              //      will be used in hash update
87                                              //      and completion
88  )
89  {
90      CRYPT_RESULT       retVal = 0;
91
92      pAssert(hashState != NULL);
93      hashState->type = HASH_STATE_EMPTY;
94
95      // Call crypto engine start hash function
96      if((retVal = _cpri__StartHash(hashAlg, FALSE, &hashState->state)) > 0)
97          hashState->type = HASH_STATE_HASH;
98
99      return retVal;
100 }
```

9.16.5.3 CryptStartHashSequence()

Start a hash stack for a sequence object and return the size, in bytes, of the digest. This call uses the form of the hash state that requires context save and restored.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
101   UINT16
102   CryptStartHashSequence(
103       TPMI_ALG_HASH        hashAlg,           // IN: hash algorithm
104       HASH_STATE          *hashState          // OUT: the state of hash stack. It
105                                               //      will be used in hash update
106                                               //      and completion
107   )
108   {
109       CRYPT_RESULT   retVal = 0;
110
111       pAssert(hashState != NULL);
112       hashState->type = HASH_STATE_EMPTY;
113
114       // Call crypto engine start hash function
115       if((retVal = _cpri__StartHash(hashAlg, TRUE, &hashState->state)) > 0)
116           hashState->type = HASH_STATE_HASH;
117
118       return retVal;
119
120   }
```

9.16.5.4 CryptStartHMAC()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
121   UINT16
122   CryptStartHMAC(
123       TPMI_ALG_HASH        hashAlg,           // IN: hash algorithm
124       UINT16               keySize,           // IN: the size of HMAC key in bytes
125       BYTE                *key,               // IN: HMAC key
126       HMAC_STATE          *hmacState          // OUT: the state of HMAC stack. It
127                                               //      will be used in HMAC update
128                                               //      and completion
129   )
130   {
131       HASH_STATE      *hashState = (HASH_STATE *)hmacState;
132       CRYPT_RESULT    retVal;
133
134       hashState->type = HASH_STATE_EMPTY;
135
136       if((retVal =  _cpri__StartHMAC(hashAlg, FALSE, &hashState->state, keySize, key,
137                                      &hmacState->hmacKey.b)) > 0)
138           hashState->type = HASH_STATE_HMAC;
139
140       return retVal;
141   }
```

9.16.5.5 CryptStartHMACSequence()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

This call is used to start a sequence HMAC that spans multiple TPM commands.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
142  UINT16
143  CryptStartHMACSequence(
144      TPMI_ALG_HASH         hashAlg,           // IN: hash algorithm
145      UINT16                keySize,           // IN: the size of HMAC key in bytes
146      BYTE                 *key,               // IN: HMAC key
147      HMAC_STATE           *hmacState          // OUT: the state of HMAC stack. It
148                                               //      will be used in HMAC update
149                                               //      and completion
150  )
151  {
152      HASH_STATE       *hashState = (HASH_STATE *)hmacState;
153      CRYPT_RESULT     retVal;
154
155      hashState->type = HASH_STATE_EMPTY;
156
157      if((retVal =  _cpri__StartHMAC(hashAlg, TRUE, &hashState->state,
158                                     keySize, key, &hmacState->hmacKey.b)) > 0)
159          hashState->type = HASH_STATE_HMAC;
160
161      return retVal;
162  }
```

9.16.5.6 CryptStartHMAC2B()

This function starts an HMAC and returns the size of the digest that will be produced.

This function is provided to support the most common use of starting an HMAC with a TPM2B key.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
163  UINT16
164  CryptStartHMAC2B(
165      TPMI_ALG_HASH         hashAlg,           // IN: hash algorithm
166      TPM2B                *key,               // IN: HMAC key
167      HMAC_STATE           *hmacState          // OUT: the state of HMAC stack. It
168                                               //      will be used in HMAC update
169                                               //      and completion
170  )
171  {
172      return CryptStartHMAC(hashAlg, key->size, key->buffer, hmacState);
173  }
```

9.16.5.7 CryptStartHMACSequence2B()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

This function is provided to support the most common use of starting an HMAC with a TPM2B key.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
174  UINT16
175  CryptStartHMACSequence2B(
176      TPMI_ALG_HASH         hashAlg,          // IN: hash algorithm
177      TPM2B                *key,              // IN: HMAC key
178      HMAC_STATE           *hmacState         // OUT: the state of HMAC stack. It
179                                              //      will be used in HMAC update
180                                              //      and completion
181  )
182  {
183      return CryptStartHMACSequence(hashAlg, key->size, key->buffer, hmacState);
184  }
```

9.16.5.8 CryptUpdateDigest()

This function updates a digest (hash or HMAC) with an array of octets.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```
185  void
186  CryptUpdateDigest(
187      void                 *digestState,      // IN: the state of hash stack
188      UINT32                dataSize,         // IN: the size of data
189      BYTE                 *data              // IN: data to be hashed
190  )
191  {
192      HASH_STATE      *hashState = (HASH_STATE *)digestState;
193
194      pAssert(digestState != NULL);
195
196      if(hashState->type != HASH_STATE_EMPTY && data != NULL && dataSize != 0)
197      {
198          // Call crypto engine update hash function
199          _cpri__UpdateHash(&hashState->state, dataSize, data);
200      }
201      return;
202  }
```

9.16.5.9 CryptUpdateDigest2B()

This function updates a digest (hash or HMAC) with a TPM2B.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```
203  void
204  CryptUpdateDigest2B(
205      void                 *digestState,      // IN: the digest state
```

```
206         TPM2B               *bIn                // IN: 2B containing the data
207     )
208     {
209         // Only compute the digest if a pointer to the 2B is provided.
210         // In CryptUpdateDigest(), if size is zero or buffer is NULL, then no change
211         // to the digest occurs. This function should not provide a buffer if bIn is
212         // not provided.
213         if(bIn != NULL)
214             CryptUpdateDigest(digestState, bIn->size, bIn->buffer);
215         return;
216     }
```

9.16.5.10 CryptUpdateDigestInt()

This function is used to include an integer value to a hash stack. The function marshals the integer into its canonical form before calling CryptUpdateHash().

```
217     void
218     CryptUpdateDigestInt(
219         void                *state,             // IN: the state of hash stack
220         UINT32               intSize,           // IN: the size of 'intValue' in bytes
221         void                *intValue           // IN: integer value to be hashed
222     )
223     {
224
225     #if BIG_ENDIAN_TPM == YES
226         pAssert(    intValue != NULL && (intSize == 1 || intSize == 2
227                 ||  intSize == 4 || intSize == 8));
228         CryptUpdateHash(state, inSize, (BYTE *)intValue);
229     #else
230
231         BYTE        marshalBuffer[8];
232         // Point to the big end of an little-endian value
233         BYTE        *p = &((BYTE *)intValue)[intSize - 1];
234         // Point to the big end of an big-endian value
235         BYTE        *q = marshalBuffer;
236
237         pAssert(intValue != NULL);
238         switch (intSize)
239         {
240         case 8:
241             *q++ = *p--;
242             *q++ = *p--;
243             *q++ = *p--;
244             *q++ = *p--;
245         case 4:
246             *q++ = *p--;
247             *q++ = *p--;
248         case 2:
249             *q++ = *p--;
250         case 1:
251             *q = *p;
252             // Call update the hash
253             CryptUpdateDigest(state, intSize, marshalBuffer);
254             break;
255         default:
256             FAIL(0);
257         }
258
259     #endif
260         return;
261     }
```

9.16.5.11 CryptCompleteHash()

This function completes a hash sequence and returns the digest.

This function can be called to complete either an HMAC or hash sequence. The state type determines if the context type is a hash or HMAC. If an HMAC, then the call is forwarded to CryptCompleteHash().

If **digestSize** is smaller than the digest size of hash/HMAC algorithm, the most significant bytes of required size will be returned

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
262  UINT16
263  CryptCompleteHash(
264      void                *state,             // IN: the state of hash stack
265      UINT16               digestSize,        // IN: size of digest buffer
266      BYTE                *digest             // OUT: hash digest
267  )
268  {
269      HASH_STATE      *hashState = (HASH_STATE *)state;     // local value
270
271      // If the session type is HMAC, then could forward this to
272      // the HMAC processing and not cause an error. However, if no
273      // function calls this routine to forward it, then we can't get
274      // test coverage. The decision is to assert if this is called with
275      // the type == HMAC and fix anything that makes the wrong call.
276      pAssert(hashState->type == HASH_STATE_HASH);
277
278      // Set the state to empty so that it doesn't get used again
279      hashState->type = HASH_STATE_EMPTY;
280
281      // Call crypto engine complete hash function
282      return     _cpri__CompleteHash(&hashState->state, digestSize, digest);
283  }
```

9.16.5.12 CryptCompleteHash2B()

This function is the same as CypteCompleteHash() but the digest is placed in a TPM2B. This is the most common use and this is provided for specification clarity. 'digst. 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'

```
284  UINT16
285  CryptCompleteHash2B(
286      void                *state,             // IN: the state of hash stack
287      TPM2B               *digest             // IN: the size of the buffer
288                                              // Out: requested number of bytes
289  )
290  {
291      UINT16              retVal = 0;
292
293      if(digest != NULL)
294          retVal = CryptCompleteHash(state, digest->size, digest->buffer);
295
296      return retVal;
297  }
```

9.16.5.13 CryptHashBlock()

Hash a block of data and return the results. If the digest is larger than *retSize*, it is truncated and with the least significant octets dropped.

Return Value	Meaning
>=0	the number of bytes placed in *ret*

```
298  UINT16
299  CryptHashBlock(
300      TPM_ALG_ID           algId,              // IN: the hash algorithm to use
301      UINT16               blockSize,          // IN: size of the data block
302      BYTE                *block,              // IN: address of the block to hash
303      UINT16               retSize,            // IN: size of the return buffer
304      BYTE                *ret                 // OUT: address of the buffer
305  )
306  {
307      return _cpri__HashBlock(algId, blockSize, block, retSize, ret);
308  }
```

9.16.5.14 CryptCompleteHMAC()

This function completes a HMAC sequence and returns the digest. If *digestSize* is smaller than the digest size of the HMAC algorithm, the most significant bytes of required size will be returned.

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
309  UINT16
310  CryptCompleteHMAC(
311      HMAC_STATE          *hmacState,          // IN: the state of HMAC stack
312      UINT32               digestSize,         // IN: size of digest buffer
313      BYTE                *digest              // OUT: HMAC digest
314  )
315  {
316      HASH_STATE      *hashState;
317
318      pAssert(hmacState != NULL);
319      hashState = &hmacState->hashState;
320
321      pAssert(hashState->type == HASH_STATE_HMAC);
322
323      hashState->type = HASH_STATE_EMPTY;
324
325      return _cpri__CompleteHMAC(&hashState->state, &hmacState->hmacKey.b,
326                                 digestSize, digest);
327
328  }
```

9.16.5.15 CryptCompleteHMAC2B()

This function is the same as CryptCompleteHMAC() but the HMAC result is returned in a TPM2B which is the most common use.

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
329  UINT16
```

```
330 CryptCompleteHMAC2B(
331     HMAC_STATE          *hmacState,         // IN: the state of HMAC stack
332     TPM2B               *digest             // OUT: HMAC
333 )
334 {
335     UINT16               retVal = 0;
336     if(digest != NULL)
337         retVal = CryptCompleteHMAC(hmacState, digest->size, digest->buffer);
338     return retVal;
339 }
```

9.16.5.16 CryptHashStateImportExport()

This function is used to prepare a hash state context for export or to import it into the internal format. It is used by TPM2_ContextSave() and TPM2_ContextLoad() via SequenceDataImportExport(). This is just a passthrough function to the crypto library.

```
340 void
341 CryptHashStateImportExport(
342     HASH_STATE          *internalFmt,     // IN: state to export
343     HASH_STATE          *externalFmt,     // OUT: exported state
344     IMPORT_EXPORT        direction
345     )
346 {
347     _cpri__ImportExportHashState(&internalFmt->state,
348                                  (EXPORT_HASH_STATE *)&externalFmt->state,
349                                  direction);
350 }
```

9.16.5.17 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

```
351 UINT16
352 CryptGetHashDigestSize(
353     TPM_ALG_ID          hashAlg             // IN: hash algorithm
354 )
355 {
356     return _cpri__GetDigestSize(hashAlg);
357 }
```

9.16.5.18 CryptGetHashBlockSize()

Get the digest size in byte of a hash algorithm.

Return Value	Meaning
0	block size for TPM_ALG_NULL
> 0	block size

```
358 UINT16
359 CryptGetHashBlockSize(
360     TPM_ALG_ID          hash                // IN: hash algorithm to look up
361 )
```

```
362 {
363     return _cpri__GetHashBlockSize(hash);
364 }
```

9.16.5.19 CryptGetHashAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and an *index* value of 2 will return the last implemented hash. All other index values will return TPM_ALG_NULL.

Return Value	Meaning
TPM_ALG_xxx()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
365 TPM_ALG_ID
366 CryptGetHashAlgByIndex(
367     UINT32      index       // IN: the index
368 )
369 {
370     return _cpri__GetHashAlgByIndex(index);
371 }
```

9.16.5.20 CryptSignHMAC()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

Error Returns	Meaning
none	…

```
372 static TPM_RC
373 CryptSignHMAC(
374     OBJECT              *signKey,           // IN: HMAC key sign the hash
375     TPMT_SIG_SCHEME     *scheme,            // IN: signing scheme
376     TPM2B_DIGEST        *hashData,          // IN: hash to be signed
377     TPMT_SIGNATURE      *signature          // OUT: signature
378 )
379 {
380     HMAC_STATE       hmacState;
381     UINT32           digestSize;
382
383     // HMAC algorithm self testing code may be inserted here
384
385     digestSize = CryptStartHMAC2B(scheme->details.hmac.hashAlg,
386                                   &signKey->sensitive.sensitive.bits.b,
387                                   &hmacState);
388
389     // The hash algorithm must be a valid one.
390     pAssert(digestSize > 0);
391
392     CryptUpdateDigest2B(&hmacState, &hashData->b);
393
394     CryptCompleteHMAC(&hmacState, digestSize,
395                       (BYTE *) &signature->signature.hmac.digest);
396
397     // Set HMAC algorithm
398     signature->signature.hmac.hashAlg = scheme->details.hmac.hashAlg;
399
400     return TPM_RC_SUCCESS;
```

```
401 }
```

9.16.5.21 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

```
402 static TPM_RC
403 CryptHMACVerifySignature(
404     OBJECT          *signKey,           // IN: HMAC key signed the hash
405     TPM2B_DIGEST    *hashData,          // IN: digest being verified
406     TPMT_SIGNATURE  *signature          // IN: signature to be verified
407 )
408 {
409     HMAC_STATE          hmacState;
410     TPM2B_DIGEST        digestToCompare;
411
412     digestToCompare.t.size = CryptStartHMAC2B(signature->signature.hmac.hashAlg,
413                              &signKey->sensitive.sensitive.bits.b, &hmacState);
414
415     CryptUpdateDigest2B(&hmacState, &hashData->b);
416
417     CryptCompleteHMAC2B(&hmacState, &digestToCompare.b);
418
419     // Compare digest
420     if(MemoryEqual(digestToCompare.t.buffer,
421                    (BYTE *) &signature->signature.hmac.digest,
422                    digestToCompare.t.size))
423         return TPM_RC_SUCCESS;
424     else
425         return TPM_RC_SIGNATURE;
426
427 }
```

9.16.5.22 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Error Returns	Meaning
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```
428 static TPM_RC
429 CryptGenerateKeyedHash(
430     TPMT_PUBLIC             *publicArea,        // IN/OUT: the public area template
431                                                 //         for the new key.
432     TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN:  sensitive creation data
433     TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
434     TPM_ALG_ID               kdfHashAlg,        // IN: algorithm for the KDF
435     TPM2B_SEED              *seed,              // IN: the seed
436     TPM2B_NAME              *name               // IN: name of the object
437 )
438 {
439     TPMT_KEYEDHASH_SCHEME   *scheme;
440     TPM_ALG_ID               hashAlg;
441     UINT16                   hashBlockSize;
442
443     scheme = &publicArea->parameters.keyedHashDetail.scheme;
444
```

```
445         pAssert(publicArea->type == TPM_ALG_KEYEDHASH);
446
447         // Pick the limiting hash algorithm
448         if(scheme->scheme == TPM_ALG_NULL)
449             hashAlg = publicArea->nameAlg;
450         else if(scheme->scheme == TPM_ALG_XOR)
451             hashAlg = scheme->details.xor.hashAlg;
452         else
453             hashAlg = scheme->details.hmac.hashAlg;
454         hashBlockSize =  CryptGetHashBlockSize(hashAlg);
455
456         // if this is a signing or a decryption key, then then the limit
457         // for the data size is the block size of the hash. This limit
458         // is set because larger values have lower entropy because of the
459         // HMAC function.
460         if(     publicArea->objectAttributes.sensitiveDataOrigin == CLEAR
461            && (   publicArea->objectAttributes.decrypt
462                || publicArea->objectAttributes.sign)
463            && sensitiveCreate->data.t.size > hashBlockSize)
464
465             return TPM_RC_SIZE;
466
467         if(publicArea->objectAttributes.sensitiveDataOrigin == SET)
468         {
469             // Created block cannot be larger than the structure allows.
470             if(hashBlockSize > MAX_SYM_DATA)
471                 hashBlockSize = MAX_SYM_DATA;
472
473             // Create new keyedHash object
474             sensitive->sensitive.bits.t.size = hashBlockSize;
475
476             CryptKDFa(kdfHashAlg,
477                       &seed->b,
478                       "sensitive",  //This string is a vendor-
479                       //specific information
480                       &name->b,              // computed from the public template
481                       NULL,            // 32-bit ENDIAN counter.
482                       sensitive->sensitive.bits.t.size * 8,
483                       sensitive->sensitive.bits.t.buffer, NULL);
484         }
485         else
486         {
487             // Copy input data to sensitive area
488             MemoryCopy2B(&sensitive->sensitive.any.b, &sensitiveCreate->data.b,
489                          sizeof(sensitive->sensitive.any.t.buffer));
490         }
491
492         // Compute obfuscation.  Parent handle is not available and not needed for
493         // symmetric object at this point.  TPM_RH_UNASSIGNED is passed at the
494         // place of parent handle
495         CryptComputeSymValue(TPM_RH_UNASSIGNED, publicArea, sensitive, seed,
496                              kdfHashAlg, name);
497
498         CryptComputeSymmetricUnique(publicArea->nameAlg,
499                                     sensitive,
500                                     &publicArea->unique.keyedHash);
501         return TPM_RC_SUCCESS;
502     }
```

9.16.5.23 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.

```
503 //%#define CryptKDFa(hashAlg, key, label, contextU, contextV,    \
504 //%                  sizeInBits, keyStream, counterInOut)       \
505 //%        _cpri__KDFa(                                         \
506 //%                    ((TPM_ALG_ID)hashAlg),                   \
507 //%                    ((TPM2B *)key),                          \
508 //%                    ((const char *)label),                   \
509 //%                    ((TPM2B *)contextU),                     \
510 //%                    ((TPM2B *)contextV),                     \
511 //%                    ((UINT32)sizeInBits),                    \
512 //%                    ((BYTE *)keyStream),                     \
513 //%                    ((UINT32 *)counterInOut),                \
514 //%                    ((BOOL) FALSE)                           \
515 //%                   )
516 //%
```

9.16.5.24 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*.

```
517 //%#define CryptKDFaOnce(hashAlg, key, label, contextU, contextV,    \
518 //%                      sizeInBits, keyStream, counterInOut)       \
519 //%        _cpri__KDFa(                                             \
520 //%                    ((TPM_ALG_ID)hashAlg),                       \
521 //%                    ((TPM2B *)key),                              \
522 //%                    ((const char *)label),                       \
523 //%                    ((TPM2B *)contextU),                         \
524 //%                    ((TPM2B *)contextV),                         \
525 //%                    ((UINT32)sizeInBits),                        \
526 //%                    ((BYTE *)keyStream),                         \
527 //%                    ((UINT32 *)counterInOut),                    \
528 //%                    ((BOOL) TRUE)                                \
529 //%                   )
530 //%
```

9.16.5.25 KDFa()

This function is used by functions outside of CryptUtil() to access _cpri_KDFa().

```
531 void
532 KDFa(
533     TPM_ALG_ID           hash,              // IN: hash algorithm used in HMAC
534     TPM2B               *key,               // IN: HMAC key
535     const char          *label,             // IN: a null-terminated label for KDF
536     TPM2B               *contextU,          // IN: context U
537     TPM2B               *contextV,          // IN: context V
538     UINT32               sizeInBits,        // IN: size of generated key in bits
539     BYTE                *keyStream,         // OUT: key buffer
540     UINT32              *counterInOut       // IN/OUT: caller may provide the
541                                             //         iteration counter for
542                                             //         incremental operations to
543                                             //         avoid large intermediate
544                                             //         buffers.
545 )
546 {
547     CryptKDFa(hash, key, label, contextU, contextV, sizeInBits,
548               keyStream, counterInOut);
```

```
549 }
```

9.16.5.26 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().

```
550 //%#define CryptKDFe(hashAlg, Z, label, partyUInfo, partyVInfo,          \
551 //%                  sizeInBits, keyStream)                             \
552 //% _cpri__KDFe(                                                        \
553 //%             ((TPM_ALG_ID)hashAlg),                                  \
554 //%             ((TPM2B *)Z),                                           \
555 //%             ((const char *)label),                                  \
556 //%             ((TPM2B *)partyUInfo),                                  \
557 //%             ((TPM2B *)partyVInfo),                                  \
558 //%             ((UINT32)sizeInBits),                                   \
559 //%             ((BYTE *)keyStream)                                     \
560 //%            )
561 //%
562 #endif //TPM_ALG_KEYEDHASH    //% 1
```

9.16.6 RSA Functions

9.16.6.1 BuildRSA()

Function to set the cryptographic elements of an RSA key into a structure to simplify the interface to _cpri__ RSA function. This can/should be eliminated by building this structure into the object structure.

```
563 #ifdef TPM_ALG_RSA          //% 2
564 static void
565 BuildRSA(
566     OBJECT      *rsaKey,
567     RSA_KEY     *key
568 )
569 {
570     key->exponent = rsaKey->publicArea.parameters.rsaDetail.exponent;
571     if(key->exponent == 0)
572         key->exponent = RSA_DEFAULT_PUBLIC_EXPONENT;
573     key->publicKey = &rsaKey->publicArea.unique.rsa.b;
574
575     if(rsaKey->attributes.publicOnly || rsaKey->privateExponent.t.size == 0)
576         key->privateKey = NULL;
577     else
578         key->privateKey = &(rsaKey->privateExponent.b);
579 }
```

9.16.6.2 CryptTestKeyRSA()

This function provides the interface to _cpri__TestKeyRSA(). If both *p* and *q* are provided, *n* will be set to *p*q*.

If only *p* is provided, *q* is computed by *q* = *n*/*p*. If *n* 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

```
580 TPM_RC
581 CryptTestKeyRSA(
582     TPM2B               *d,                 // OUT: receives the private exponent
583     UINT32               e,                 // IN: public exponent
584     TPM2B               *n,                 // IN/OUT: public modulus
585     TPM2B               *p,                 // IN: a first prime
586     TPM2B               *q                  // IN: an optional second prime
587 )
588 {
589     CRYPT_RESULT    retVal;
590
591     pAssert(d != NULL && n != NULL && p != NULL);
592     // Set the exponent
593     if(e == 0)
594         e = RSA_DEFAULT_PUBLIC_EXPONENT;
595     // CRYPT_PARAMETER
596     retVal = _cpri__TestKeyRSA(d, e, n, p, q);
597     if(retVal == CRYPT_SUCCESS)
598         return TPM_RC_SUCCESS;
599     else
600         return TPM_RC_BINDING;  // convert CRYPT_PARAMETER
601 }
```

9.16.6.3 CryptGenerateKeyRSA()

This function is called to generate an RSA key from a provided seed. It calls _cpri__GenerateKeyRSA() to perform the computations.

Error Returns	Meaning
TPM_RC_RANGE	the exponent value is not supported
TPM_RC_CANCELLED	key generation has been cancelled
TPM_RC_VALUE	exponent is not prime or is less than 3; or could not find a prime using the provided parameters

```
602 static TPM_RC
603 CryptGenerateKeyRSA(
604     TPMT_PUBLIC         *publicArea,        // IN/OUT: The public area template for
605                                             //      the new key. The public key
606                                             //      area will be replaced by the
607                                             //      product of two primes found by
608                                             //      this function
609     TPMT_SENSITIVE      *sensitive,         // OUT: the sensitive area will be
610                                             //      updated to contain the first
611                                             //      prime and the symmetric
612                                             //      encryption key
613     TPM_ALG_ID           hashAlg,           // IN: the hash algorithm for the KDF
614     TPM2B_SEED          *seed,              // IN: Seed for the creation
615     TPM2B_NAME          *name,              // IN: Object name
616     UINT32              *counter            // OUT: last iteration of the counter
617 )
618 {
619     CRYPT_RESULT    retVal;
620     UINT32          exponent = publicArea->parameters.rsaDetail.exponent;
621
622     // In this implementation, only the default exponent is allowed
623     if(exponent != 0 && exponent != RSA_DEFAULT_PUBLIC_EXPONENT)
624         return TPM_RC_RANGE;
```

```
625         exponent = RSA_DEFAULT_PUBLIC_EXPONENT;
626
627         *counter = 0;
628
629         // _cpri_GenerateKeyRSA can return CRYPT_CANCEL or CRYPT_FAIL
630         retVal = _cpri__GenerateKeyRSA(&publicArea->unique.rsa.b,
631                                        &sensitive->sensitive.rsa.b,
632                                        publicArea->parameters.rsaDetail.keyBits,
633                                        exponent,
634                                        hashAlg,
635                                        &seed->b,
636                                        "RSA key by vendor",
637                                        &name->b,
638                                        counter);
639
640         // CRYPT_CANCEL -> TPM_RC_CANCELLED; CRYPT_FAIL -> TPM_RC_VALUE
641         return TranslateCryptErrors(retVal);
642
643     }
```

9.16.6.4 CryptLoadPrivateRSA()

This function is called to generate the private exponent of an RSA key. It uses CryptTestKeyRSA().

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of *rsaKey* are not matched

```
644     TPM_RC
645     CryptLoadPrivateRSA(
646         OBJECT      *rsaKey     // IN: the RSA key object
647     )
648     {
649         TPM_RC           result;
650         TPMT_PUBLIC     *publicArea = &rsaKey->publicArea;
651         TPMT_SENSITIVE  *sensitive = &rsaKey->sensitive;
652
653         // Load key by computing the private exponent
654         // TPM_RC_BINDING
655         result = CryptTestKeyRSA(&(rsaKey->privateExponent.b),
656                                  publicArea->parameters.rsaDetail.exponent,
657                                  &(publicArea->unique.rsa.b),
658                                  &(sensitive->sensitive.rsa.b),
659                                  NULL);
660         if(result == TPM_RC_SUCCESS)
661             rsaKey->attributes.privateExp = SET;
662
663         return result;
664     }
```

9.16.6.5 CryptSelectRSAScheme()

This function is used by TPM2_RSA_Decrypt() and TPM2_RSA_Encrypt(). It sets up the rules to select a scheme between input and object default. This function assume the RSA object is loaded. If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both the object and *scheme* are not TPM_ALG_NULL, then if the schemes are the same, the input scheme will be chosen. if the scheme are not compatible, a NULL pointer will be returned.

The return pointer may point to a TPM_ALG_NULL scheme.

```
665     TPMT_RSA_DECRYPT*
```

```
666 CryptSelectRSAScheme(
667     TPMI_DH_OBJECT       rsaHandle,          // IN: handle of sign key
668     TPMT_RSA_DECRYPT    *scheme              // IN: a sign or decrypt scheme
669 )
670 {
671     OBJECT              *rsaObject;
672     TPMT_ASYM_SCHEME    *keyScheme;
673     TPMT_RSA_DECRYPT    *retVal = NULL;
674
675     // Get sign object pointer
676     rsaObject = ObjectGet(rsaHandle);
677     keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
678
679     // if the default scheme of the object is TPM_ALG_NULL, then select the
680     // input scheme
681     if(keyScheme->scheme == TPM_ALG_NULL)
682     {
683         retVal = scheme;
684     }
685     // if the object scheme is not TPM_ALG_NULL and the input scheme is
686     // TPM_ALG_NULL, then select the default scheme of the object.
687     else if(scheme->scheme == TPM_ALG_NULL)
688     {
689         // if input scheme is NULL
690         retVal = (TPMT_RSA_DECRYPT *)keyScheme;
691     }
692     // get here if both the object scheme and the input scheme are
693     // not TPM_ALG_NULL. Need to insure that they are the same.
694     // IMPLEMENTATION NOTE: This could cause problems if future versions have
695     // schemes that have more values than just a hash algorithm. A new function
696     // (IsSchemeSame()) might be needed then.
697     else if(   keyScheme->scheme == scheme->scheme
698             && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
699     {
700         retVal = scheme;
701     }
702     // two different, incompatible schemes specified wo will return NULL
703     return retVal;
704 }
```

9.16.6.6 CryptDecryptRSA()

This function is the interface to _cpri__DecryptRSA(). It handles the return codes from that function and converts them from CRYPT_RESULT to TPM_RC values. 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.

```
705 TPM_RC
706 CryptDecryptRSA(
707     UINT16              *dataOutSize,       // OUT: size of plain text in byte
708     BYTE                *dataOut,           // OUT: plain text
709     OBJECT              *rsaKey,            // IN: internal RSA key
710     TPMT_RSA_DECRYPT    *scheme,            // IN: selects the padding scheme
711     UINT16               cipherInSize,      // IN: size of cipher text  in byte
712     BYTE                *cipherIn,          // IN: cipher text
713     const char          *label              // IN: a label, when needed
```

```
714 )
715 {
716     RSA_KEY         key;
717     CRYPT_RESULT    retVal = CRYPT_SUCCESS;
718     UINT32          dSize;                  // Place to put temporary value for the
719                                             // returned data size
720     TPMI_ALG_HASH   hashAlg = TPM_ALG_NULL; // hash algorithm in the selected
721                                             // padding scheme
722     TPM_RC          result = TPM_RC_SUCCESS;
723     // pointer checks
724     pAssert(   (dataOutSize != NULL) && (dataOut != NULL)
725             && (rsaKey != NULL) && (cipherIn != NULL));
726
727     // The public type is a RSA decrypt key
728     pAssert(   rsaKey->publicArea.type == TPM_ALG_RSA
729             && rsaKey->publicArea.objectAttributes.decrypt == SET);
730
731     // Must have the private portion loaded.  This check is made before this
732     // function is called.
733     pAssert(rsaKey->attributes.publicOnly == CLEAR);
734
735     // decryption requires that the private modulus be present
736     if(rsaKey->attributes.privateExp == CLEAR)
737     {
738
739         // Load key by computing the private exponent
740         // CryptLoadPrivateRSA may return TPM_RC_BINDING
741         result = CryptLoadPrivateRSA(rsaKey);
742     }
743
744     // the input buffer must be the size of the key
745     if(result == TPM_RC_SUCCESS) {
746         if(cipherInSize != rsaKey->publicArea.unique.rsa.t.size)
747             result = TPM_RC_SIZE;
748         else
749         {
750             BuildRSA(rsaKey, &key);
751
752             // Initialize the dOutSize parameter
753             dSize = *dataOutSize;
754
755             // For OAEP scheme, initialize the hash algorithm for padding
756             if(scheme->scheme == TPM_ALG_OAEP)
757                 hashAlg = scheme->details.oaep.hashAlg;
758
759             // _cpri__DecryptRSA may return CRYPT_PARAMETER CRYPT_FAIL CRYPT_SCHEME
760             retVal = _cpri__DecryptRSA(&dSize, dataOut, &key, scheme->scheme,
761                                        cipherInSize, cipherIn, hashAlg, label);
762
763             // Scheme must have been validated when the key was loaded/imported
764             pAssert(retVal != CRYPT_SCHEME);
765
766             // Set the return size
767             pAssert(dSize <= UINT16_MAX);
768             *dataOutSize = (UINT16)dSize;
769
770             // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_FAIL -> TPM_RC_VALUE
771             result = TranslateCryptErrors(retVal);
772         }
773     }
774     return result;
775 }
```

9.16.6.7 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

```
776  TPM_RC
777  CryptEncryptRSA(
778      UINT16              *cipherOutSize,     // OUT: size of cipher text in byte
779      BYTE                *cipherOut,         // OUT: cipher text
780      OBJECT              *rsaKey,            // IN: internal RSA key
781      TPMT_RSA_DECRYPT    *scheme,            // IN: selects the padding scheme
782      UINT16               dataInSize,        // IN: size of plain text in byte
783      BYTE                *dataIn,            // IN: plain text
784      const char          *label              // IN: an optional label
785  )
786  {
787      RSA_KEY              key;
788      CRYPT_RESULT         retVal;
789      UINT32               cOutSize;                  // Conversion variable
790      TPMI_ALG_HASH        hashAlg = TPM_ALG_NULL;    // hash algorithm in selected
791                                                      // padding scheme
792
793      // must have a pointer to a key and some data to encrypt
794      pAssert(rsaKey != NULL && dataIn != NULL);
795
796      // The public type is a RSA decryption key
797      pAssert(   rsaKey->publicArea.type == TPM_ALG_RSA
798              && rsaKey->publicArea.objectAttributes.decrypt == SET);
799
800      // If the cipher buffer must be provided and it must be large enough
801      // for the result
802      pAssert(   cipherOut != NULL
803              && cipherOutSize != NULL
804              && *cipherOutSize >= rsaKey->publicArea.unique.rsa.t.size);
805
806      // Only need the public key and exponent for encryption
807      BuildRSA(rsaKey, &key);
808
809      // Copy the size to the conversion buffer
810      cOutSize = *cipherOutSize;
811
812      // For OAEP scheme, initialize the hash algorithm for padding
813      if(scheme->scheme == TPM_ALG_OAEP)
814          hashAlg = scheme->details.oaep.hashAlg;
815
816      // Encrypt the data
817      // _cpri__EncryptRSA may return CRYPT_PARAMETER or CRYPT_SCHEME
818      retVal = _cpri__EncryptRSA(&cOutSize,cipherOut, &key, scheme->scheme,
819                                 dataInSize, dataIn, hashAlg, label);
820
821      pAssert (cOutSize <= UINT16_MAX);
822      *cipherOutSize = (UINT16)cOutSize;
823      // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_SCHEME -> TPM_RC_SCHEME
824      return TranslateCryptErrors(retVal);
825  }
```

9.16.6.8 CryptSignRSA()

This function is used to sign a digest with an RSA signing key.

Error Returns	Meaning
TPM_RC_BINDING	public and private part of *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*

```
826 static TPM_RC
827 CryptSignRSA(
828     OBJECT              *signKey,           // IN: RSA key signs the hash
829     TPMT_SIG_SCHEME     *scheme,            // IN: sign scheme
830     TPM2B_DIGEST        *hashData,          // IN: hash to be signed
831     TPMT_SIGNATURE      *sig                // OUT: signature
832 )
833 {
834     UINT32               signSize;
835     RSA_KEY              key;
836     CRYPT_RESULT         retVal;
837     TPM_RC               result = TPM_RC_SUCCESS;
838
839     pAssert(    (signKey != NULL) && (scheme != NULL)
840                 && (hashData != NULL) && (sig != NULL));
841
842
843     // assume that the key has private part loaded and that it is a signing key.
844     pAssert(   (signKey->attributes.publicOnly == CLEAR)
845             && (signKey->publicArea.objectAttributes.sign == SET));
846
847     // check if the private exponent has been computed
848     if(signKey->attributes.privateExp == CLEAR)
849     {
850         // need to compute the private exponent
851         TPM_RC      result;
852         // May return TPM_RC_BINDING
853         result = CryptLoadPrivateRSA(signKey);
854     }
855
856     if(result == TPM_RC_SUCCESS)
857     {
858         BuildRSA(signKey, &key);
859
860         // initialize the common signature values
861         sig->sigAlg = scheme->scheme;
862         sig->signature.any.hashAlg = scheme->details.any.hashAlg;
863
864         // _crypi__SignRSA can return CRYPT_SCHEME and CRYPT_PARAMETER
865         retVal = _cpri__SignRSA(&signSize,
866                                 sig->signature.rsassa.sig.t.buffer,
867                                 &key,
868                                 sig->sigAlg,
869                                 sig->signature.any.hashAlg,
870                                 hashData->t.size, hashData->t.buffer);
871         pAssert(signSize <= UINT16_MAX);
872         sig->signature.rsassa.sig.t.size = (UINT16)signSize;
873
874         // CRYPT_SCHEME -> TPM_RC_SCHEME; CRYPT_PARAMTER -> TPM_RC_VALUE
875         result = TranslateCryptErrors(retVal);
876     }
877     return result;
```

```
878 }
```

9.16.6.9 CryptRSAVerifySignature()

This function is used to verify signature signed by a RSA key.

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not genuine
TPM_RC_SCHEME	signature scheme not supported

```
879 static TPM_RC
880 CryptRSAVerifySignature(
881     OBJECT              *signKey,           // IN: RSA key signed the hash
882     TPM2B_DIGEST        *hashData,          // IN: hash being signed
883     TPMT_SIGNATURE      *sig                // IN: signature to be verified
884 )
885 {
886     RSA_KEY              key;
887     CRYPT_RESULT         retVal;
888     TPM_RC               result;
889
890     // Validate parameter assumptions
891     pAssert((signKey != NULL) && (hashData != NULL) && (sig != NULL));
892
893     // This is a public-key-only operation
894     BuildRSA(signKey, &key);
895
896     // Call crypto engine to verify signature
897     // _cpri_ValidateSignaturRSA may return CRYPT_FAIL or CRYPT_SCHEME
898     retVal = _cpri__ValidateSignatureRSA(&key,sig->sigAlg,
899                                          sig->signature.any.hashAlg,
900                                          hashData->t.size,
901                                          hashData->t.buffer,
902                                          sig->signature.rsassa.sig.t.size,
903                                          sig->signature.rsassa.sig.t.buffer,
904                                          0);
905     // _cpri__ValidateSignatureRSA can return CRYPT_SUCCESS, CRYPT_FAIL, or
906     // CRYPT_SCHEME. Translate CRYPT_FAIL to TPM_RC_SIGNATURE
907     if(retVal == CRYPT_FAIL)
908         result = TPM_RC_SIGNATURE;
909     else
910         // CRYPT_SCHEME -> TPM_RC_SCHEME
911         result = TranslateCryptErrors(retVal);
912
913     return result;
914 }
915 #endif //TPM_ALG_RSA      //% 2
```

9.16.7 ECC Functions

9.16.7.1 CryptEccGetCurveDataPointer()

This function returns a pointer to an ECC_CURVE_VALUES structure that contains the parameters for the key size and schemes for a given curve.

```
916 #ifdef TPM_ALG_ECC //% 3
917 static const ECC_CURVE    *
918 CryptEccGetCurveDataPointer(
919     TPM_ECC_CURVE        curveID            // IN: id of the curve
920 )
```

```
921 {
922     return _cpri__EccGetParametersByCurveId(curveID);
923 }
```

9.16.7.2 CryptEccGetKeySizeInBits()

This function returns the size in bits of the key associated with a curve.

```
924 UINT16
925 CryptEccGetKeySizeInBits(
926     TPM_ECC_CURVE            curveID     // IN: id of the curve
927 )
928 {
929     const ECC_CURVE         *curve = CryptEccGetCurveDataPointer(curveID);
930     UINT16                   keySizeInBits = 0;
931 
932     if(curve != NULL)
933         keySizeInBits = curve->keySizeBits;
934 
935     return keySizeInBits;
936 }
```

9.16.7.3 CryptEccGetKeySizeBytes()

This macro returns the size of the ECC key in bytes. It uses CryptEccGetKeySizeInBits(). The next lines will be placed in CyrptUtil_fp.h with the //% removed

```
937 //% #define CryptEccGetKeySizeInBytes(curve)            \
938 //%             ((CryptEccGetKeySizeInBits(curve)+7)/8)
```

9.16.7.4 CryptEccGetParameter()

This function returns a pointer to an ECC curve parameter. The parameter is selected by a single character designator from the set of {pnabxyh}.

```
939 const TPM2B *
940 CryptEccGetParameter(
941     char                 p,                 // IN: the parameter selector
942     TPM_ECC_CURVE        curveId              // IN: the curve id
943 )
944 {
945     const ECC_CURVE     *curve = _cpri__EccGetParametersByCurveId(curveId);
946     const TPM2B         *parameter = NULL;
947 
948     if(curve != NULL)
949     {
950         switch (p)
951         {
952         case 'p':
953             parameter = curve->curveData->p;
954             break;
955         case 'n':
956             parameter =  curve->curveData->n;
957             break;
958         case 'a':
959             parameter =  curve->curveData->a;
960             break;
961         case 'b':
962             parameter =  curve->curveData->b;
963             break;
964         case 'x':
```

```
965             parameter =  curve->curveData->x;
966             break;
967         case 'y':
968             parameter =  curve->curveData->y;
969             break;
970         case 'h':
971             parameter =  curve->curveData->h;
972             break;
973         default:
974             break;
975         }
976     }
977     return parameter;
978 }
```

9.16.7.5 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```
979 const TPMT_ECC_SCHEME *
980 CryptGetCurveSignScheme(
981     TPM_ECC_CURVE        curveId             // IN: The curve selector
982     )
983 {
984     const ECC_CURVE         *curve = _cpri__EccGetParametersByCurveId(curveId);
985     const TPMT_ECC_SCHEME   *scheme = NULL;
986
987     if(curve != NULL)
988         scheme =  &(curve->sign);
989     return scheme;
990 }
```

9.16.7.6 CryptEccIsPointOnCurve()

This function will validate that an ECC point is on the curve of given *curveID*.

Return Value	Meaning
TRUE	if the point is on curve
FALSE	if the point is not on curve

```
991 BOOL
992 CryptEccIsPointOnCurve(
993     TPM_ECC_CURVE        curveID,            // IN: ECC curve ID
994     TPMS_ECC_POINT      *Q                   // IN: ECC point
995 )
996 {
997     // ECC algorithm self testing code may be inserted here
998
999     // Call crypto engine function to check if a ECC public point is on the
1000    // given curve
1001    if(_cpri__EccIsPointOnCurve(curveID, Q))
1002        return TRUE;
1003    else
1004        return FALSE;
1005 }
```

9.16.7.7 CryptNewEccKey()

This function creates a random ECC key that is not derived from other parameters as is a Primary Key.

```
1006    TPM_RC
1007    CryptNewEccKey(
1008        TPM_ECC_CURVE         curveID,          // IN: ECC curve
1009        TPMS_ECC_POINT       *publicPoint,      // OUT: public point
1010        TPM2B_ECC_PARAMETER *sensitive          // OUT: private area
1011    )
1012    {
1013        TPM_RC                result = TPM_RC_SUCCESS;
1014        // _cpri__GetEphemeralECC may return CRYPT_PARAMETER
1015        if(_cpri__GetEphemeralEcc(publicPoint, sensitive, curveID) != CRYPT_SUCCESS)
1016            // Something is wrong with the key.
1017            result = TPM_RC_KEY;
1018
1019        return result;
1020    }
```

9.16.7.8 CryptEccPointMultiply()

This function is used to perform a point multiply *R* = [*d*]*Q*. If *Q* is not provided, the multiplication is performed using the generator point of the curve.

Error Returns	Meaning
TPM_RC_ECC_POINT	invalid optional ECC point *pIn*
TPM_RC_NO_RESULT	multiplication resulted in a point at infinity

```
1021    TPM_RC
1022    CryptEccPointMultiply(
1023        TPMS_ECC_POINT       *pOut,             // OUT: output point
1024        TPM_ECC_CURVE         curveId,          // IN: curve selector
1025        TPM2B_ECC_PARAMETER *dIn,               // IN: public scalar
1026        TPMS_ECC_POINT       *pIn               // IN: optional point
1027    )
1028    {
1029        TPM2B_ECC_PARAMETER     *n = NULL;
1030        CRYPT_RESULT             retVal;
1031
1032        pAssert(pOut != NULL && dIn != NULL);
1033
1034        if(pIn != NULL)
1035        {
1036            n = dIn;
1037            dIn = NULL;
1038        }
1039
1040        // _cpri__EccPointMultiply may return CRYPT_POINT or CRYPT_NO_RESULT
1041        retVal = _cpri__EccPointMultiply(pOut, curveId, dIn, pIn, n);
1042
1043        // CRYPT_POINT->TPM_RC_ECC_POINT and CRYPT_NO_RESULT->TPM_RC_NO_RESULT
1044        return TranslateCryptErrors(retVal);
1045    }
```

9.16.7.9 CryptGenerateKeyECC()

This function generates an ECC key from a seed value.

The method here may not work for objects that have an order *(G)* that with a different size than a private key.

Error Returns	Meaning
TPM_RC_VALUE	hash algorithm is not supported

```
1046  static TPM_RC
1047  CryptGenerateKeyECC(
1048      TPMT_PUBLIC         *publicArea,        // IN/OUT: The public area template
1049                                              //         for the new key.
1050      TPMT_SENSITIVE      *sensitive,         // IN/OUT: the sensitive area
1051      TPM_ALG_ID           hashAlg,           // IN: algorithm for the KDF
1052      TPM2B_SEED          *seed,              // IN: the seed value
1053      TPM2B_NAME          *name,              // IN: the name of the object
1054      UINT32              *counter            // OUT: the iteration counter
1055  )
1056  {
1057      CRYPT_RESULT         retVal;
1058
1059      *counter = 0;
1060
1061      // _cpri__GenerateKeyEcc only has one error return (CRYPT_PARAMETER) which means
1062      // that the hash algorithm is not supported. This should not be possible
1063      retVal = _cpri__GenerateKeyEcc(&publicArea->unique.ecc,
1064                                     &sensitive->sensitive.ecc,
1065                                     publicArea->parameters.eccDetail.curveID,
1066                                     hashAlg, &seed->b, "ECC key by vendor",
1067                                     &name->b, counter);
1068      // This will only be useful if _cpri__GenerateKeyEcc return CRYPT_CANCEL
1069      return TranslateCryptErrors(retVal);
1070  }
```

9.16.7.10 CryptSignECC()

This function is used for ECC signing operations. If the signing scheme is a split scheme, and the signing operation is successful, the commit value is retired.

Error Returns	Meaning
TPM_RC_SCHEME	unsupported *scheme*
TPM_RC_VALUE	invalid commit status (in case of a split scheme) or failed to generate r value.

```
1071  static TPM_RC
1072  CryptSignECC(
1073      OBJECT              *signKey,           // IN: ECC key to sign the hash
1074      TPMT_SIG_SCHEME     *scheme,            // IN: sign scheme
1075      TPM2B_DIGEST        *hashData,          // IN: hash to be signed
1076      TPMT_SIGNATURE      *signature          // OUT: signature
1077  )
1078  {
1079      TPM2B_ECC_PARAMETER      r;
1080      TPM2B_ECC_PARAMETER     *pr = NULL;
1081      CRYPT_RESULT             retVal;
1082
1083      if(CryptIsSplitSign(scheme->scheme))
1084      {
1085          // When this code was written, the only split scheme was ECDAA
1086          // (which can also be used for U-Prove).
1087          if(!CryptGenerateR(&r,
1088                             &scheme->details.ecdaa.count,
1089                             signKey->publicArea.parameters.eccDetail.curveID,
1090                             &signKey->name))
1091              return TPM_RC_VALUE;
```

```
1092            pr = &r;
1093        }
1094        // Call crypto engine function to sign
1095        // _cpri__SignEcc may return CRYPT_SCHEME
1096        retVal = _cpri__SignEcc(&signature->signature.ecdsa.signatureR,
1097                                &signature->signature.ecdsa.signatureS,
1098                                scheme->scheme,
1099                                scheme->details.any.hashAlg,
1100                                signKey->publicArea.parameters.eccDetail.curveID,
1101                                &signKey->sensitive.sensitive.ecc,
1102                                &hashData->b,
1103                                pr
1104                                );
1105        if(CryptIsSplitSign(scheme->scheme) && retVal == CRYPT_SUCCESS)
1106            CryptEndCommit(scheme->details.ecdaa.count);
1107        // CRYPT_SCHEME->TPM_RC_SCHEME
1108        return TranslateCryptErrors(retVal);
1109    }
```

9.16.7.11 CryptECCVerifySignature()

This function is used to verify a signature created with an ECC key.

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not valid
TPM_RC_SCHEME	the signing scheme or *hashAlg* is not supported

```
1110    static TPM_RC
1111    CryptECCVerifySignature(
1112        OBJECT              *signKey,           // IN: ECC key signed the hash
1113        TPM2B_DIGEST        *hashData,          // IN: hash being signed
1114        TPMT_SIGNATURE      *signature          // IN: signature to be verified
1115    )
1116    {
1117        CRYPT_RESULT        retVal;
1118        // This implementation uses the fact that all the defined ECC signing
1119        // schemes have the hash as the first parameter.
1120        // _cpriValidateSignatureEcc may return CRYPT_FAIL or CRYP_SCHEME
1121        retVal = _cpri__ValidateSignatureEcc(&signature->signature.ecdsa.signatureR,
1122                                             &signature->signature.ecdsa.signatureS,
1123                                             signature->sigAlg,
1124                                             signature->signature.any.hashAlg,
1125                                             signKey->publicArea.parameters.eccDetail.curveID,
1126                                             &signKey->publicArea.unique.ecc,
1127                                             &hashData->b);
1128        if(retVal == CRYPT_FAIL)
1129            return TPM_RC_SIGNATURE;
1130        // CRYPT_SCHEME->TPM_RC_SCHEME
1131        return TranslateCryptErrors(retVal);
1132    }
```

9.16.7.12 CryptGenerateR()

This function computes the commit random value for a split signing scheme.

If *c* is NULL, it indicates that *r* is being generated for TPM2_Commit(). If *c* is not NULL, the TPM will validate that the gr.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

```
1133  BOOL
1134  CryptGenerateR(
1135      TPM2B_ECC_PARAMETER *r,                 // OUT: the generated random value
1136      UINT16              *c,                 // IN/OUT: count value.
1137      TPMI_ECC_CURVE       curveID,           // IN: the curve for the value
1138      TPM2B_NAME          *name               // IN: optional name of a key to
1139                                              //     associate with 'r'
1140  )
1141  {
1142      // This holds the marshaled g_commitCounter.
1143      TPM2B_TYPE(8B, 8);
1144      TPM2B_8B                cntr = {8,{0}};
1145
1146      UINT32                   iterations;
1147      const TPM2B             *n;
1148      UINT64                   currentCount = gr.commitCounter;
1149
1150      n =  CryptEccGetParameter('n', curveID);
1151      pAssert(r != NULL && n != NULL);
1152
1153      // If this is the commit phase, use the current value of the commit counter
1154      if(c != NULL)
1155      {
1156
1157          UINT16      t1;
1158          // if the array bit is not set, can't use the value.
1159          if(!BitIsSet((*c & COMMIT_INDEX_MASK), gr.commitArray,
1160                       sizeof(gr.commitArray)))
1161              return FALSE;
1162
1163          // If it is the sign phase, figure out what the counter value was
1164          // when the commitment was made.
1165          //
1166          // When gr.commitArray has less than 64K bits, the extra
1167          // bits of 'c' are used as a check to make sure that the
1168          // signing operation is not using an out of range count value
1169          t1 = (UINT16)currentCount;
1170
1171          // If the lower bits of c are greater or equal to the lower bits of t1
1172          // then the upper bits of t1 must be one more than the upper bits
1173          // of c
1174          if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
1175              // Since the counter is behind, reduce the current count
1176              currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
1177
1178          t1 = (UINT16)currentCount;
1179          if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
1180              return FALSE;
1181          // set the counter to the value that was
1182          // present when the commitment was made
1183          currentCount = (currentCount & 0xffffffffffff0000) | *c;
1184
1185      }
1186      // Marshal the count value to a TPM2B buffer for the KDF
1187      cntr.t.size = sizeof(currentCount);
1188      UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
1189
1190      // Now can do the KDF to create the random value for the signing operation
1191      // During the creation process, we may generate an r that does not meet the
```

```
1192        // requirements of the random value.
1193        // want to generate a new r.
1194
1195        r->t.size = n->size;
1196
1197        // Arbitrary upper limit on the number of times that we can look for
1198        // a suitable random value.  The normally number of tries will be 1.
1199        for(iterations = 1; iterations < 1000000;)
1200        {
1201            BYTE    *pr = &r->b.buffer[0];
1202            int     i;
1203            CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, "ECDAA Commit",
1204                      name, &cntr.b, n->size * 8, r->t.buffer, &iterations);
1205
1206            // random value must be less than the prime
1207            if(CryptCompare(r->b.size, r->b.buffer, n->size, n->buffer) >= 0)
1208                continue;
1209
1210            // in this implementation it is required that at least bit
1211            // in the upper half of the number be set
1212            for(i = n->size/2; i > 0; i--)
1213                if(*pr++ != 0)
1214                    return TRUE;
1215        }
1216        return FALSE;
1217    }
```

9.16.7.13 CryptCommit()

This function is called when the count value is committed. The gr.c*ommitArray* value associated with the current count value is SET and *g_commitCounter* is incremented. The low-order 16 bits of old value of the counter is returned.

```
1218    UINT16
1219    CryptCommit(
1220        void
1221    )
1222    {
1223        UINT16      oldCount = (UINT16)gr.commitCounter;
1224        gr.commitCounter++;
1225        BitSet(oldCount & COMMIT_INDEX_MASK, gr.commitArray, sizeof(gr.commitArray));
1226        return oldCount;
1227    }
```

9.16.7.14 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the gr.c*ommitArray* bit associated with the count value so that it can't be used again.

```
1228    void
1229    CryptEndCommit(
1230        UINT16              c               // IN: the counter value of the commitment
1231    )
1232    {
1233        BitClear((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
1234    }
```

9.16.7.15 CryptCommitCompute()

This function performs the computations for the TPM2_Commit() command. This could be a macro.

Error Returns	Meaning
TPM_RC_NO_RESULT	*K*, *L*, or *E* is the point at infinity
TPM_RC_CANCELLED	command was cancelled

```
1235  TPM_RC
1236  CryptCommitCompute(
1237      TPMS_ECC_POINT      *K,                 // OUT: [d]B
1238      TPMS_ECC_POINT      *L,                 // OUT: [r]B
1239      TPMS_ECC_POINT      *E,                 // OUT: [r]M
1240      TPM_ECC_CURVE        curveID,           // IN: The curve for the computations
1241      TPMS_ECC_POINT      *M,                 // IN: M (P1)
1242      TPMS_ECC_POINT      *B,                 // IN: B (x2, y2)
1243      TPM2B_ECC_PARAMETER *d,                 // IN: the private scalar
1244      TPM2B_ECC_PARAMETER *r                  // IN: the computed r value
1245  )
1246  {
1247      // CRYPT_NO_RESULT->TPM_RC_NO_RESULT CRYPT_CANCEL->TPM_RC_CANCELLED
1248      return TranslateCryptErrors(
1249                  _cpri__EccCommitCompute(K, L , E, curveID, M, B, d, r));
1250  }
```

9.16.7.16 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve

Return Value	Meaning
TRUE	Get parameters success
FALSE	Unsupported ECC curve ID

```
1251  BOOL
1252  CryptEccGetParameters(
1253      TPM_ECC_CURVE                curveId,     // IN: ECC curve ID
1254      TPMS_ALGORITHM_DETAIL_ECC   *parameters // OUT: ECC parameters
1255  )
1256  {
1257      const ECC_CURVE             *curve = _cpri__EccGetParametersByCurveId(curveId);
1258      const ECC_CURVE_DATA        *data;
1259      BOOL                         found = curve != NULL;
1260
1261      if(found)
1262      {
1263
1264          data = curve->curveData;
1265
1266          parameters->curveID = curve->curveId;
1267
1268          // Key size in bit
1269          parameters->keySize = curve->keySizeBits;
1270
1271          // KDF
1272          parameters->kdf = curve->kdf;
1273
1274          // Sign
1275          parameters->sign = curve->sign;
1276
1277          // Copy p value
1278          MemoryCopy2B(&parameters->p.b, data->p, sizeof(parameters->p.t.buffer));
1279
1280          // Copy a value
1281          MemoryCopy2B(&parameters->a.b, data->a, sizeof(parameters->a.t.buffer));
```

```
1282
1283            // Copy b value
1284            MemoryCopy2B(&parameters->b.b, data->b, sizeof(parameters->b.t.buffer));
1285
1286            // Copy Gx value
1287            MemoryCopy2B(&parameters->gX.b, data->x, sizeof(parameters->gX.t.buffer));
1288
1289            // Copy Gy value
1290            MemoryCopy2B(&parameters->gY.b, data->y, sizeof(parameters->gY.t.buffer));
1291
1292            // Copy n value
1293            MemoryCopy2B(&parameters->n.b, data->n, sizeof(parameters->n.t.buffer));
1294
1295            // Copy h value
1296            MemoryCopy2B(&parameters->h.b, data->h, sizeof(parameters->h.t.buffer));
1297        }
1298        return found;
1299    }
1300    #if CC_ZGen_2Phase == YES
```

CryptEcc2PhaseKeyExchange() This is the interface to the key exchange funciton.

```
1301    TPM_RC
1302    CryptEcc2PhaseKeyExchange(
1303        TPMS_ECC_POINT          *outZ1,         // OUT: the computed point
1304        TPMS_ECC_POINT          *outZ2,         // OUT: optional second point
1305        TPM_ALG_ID               scheme,        // IN: the key exchange scheme
1306        TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
1307        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1308        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1309        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1310        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1311        )
1312    {
1313        return (TranslateCryptErrors(_cpri__C_2_2_KeyExchange(outZ1,
1314                                                              outZ2,
1315                                                              scheme,
1316                                                              curveId,
1317                                                              dsA,
1318                                                              deA,
1319                                                              QsB,
1320                                                              QeB)));
1321    }
1322    #endif //  CC_ZGen_2Phase
1323    #endif //TPM_ALG_ECC  //% 3
```

9.16.7.17 CryptIsSchemeAnonymous()

This function is used to test a scheme to see if it is an anonymous scheme The only anonymous scheme is ECDAA. ECDAA can be used to do things like U-Prove.

```
1324    BOOL
1325    CryptIsSchemeAnonymous(
1326        TPM_ALG_ID           scheme             // IN: the scheme algorithm to test
1327    )
1328    {
1329    #ifdef TPM_ALG_ECDAA
1330            return (scheme == TPM_ALG_ECDAA);
1331    #else
1332            return  0;
1333    #endif
1334    }
```

9.16.8 Symmetric Functions

9.16.8.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```
1335  void
1336  ParmDecryptSym(
1337      TPM_ALG_ID           symAlg,           // IN: the symmetric algorithm
1338      TPM_ALG_ID           hash,             // IN: hash algorithm for KDFa
1339      UINT16               keySizeInBits,    // IN: key key size in bits
1340      TPM2B               *key,              // IN: KDF HMAC key
1341      TPM2B               *nonceCaller,      // IN: nonce caller
1342      TPM2B               *nonceTpm,         // IN: nonce TPM
1343      UINT32               dataSize,         // IN: size of parameter buffer
1344      BYTE                *data              // OUT: buffer to be decrypted
1345  )
1346  {
1347      // KDF output buffer
1348      // It contains parameters for the CFB encryption
1349      // From MSB to LSB, they are the key and iv
1350      BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
1351      // Symmetric key size in byte
1352      UINT16           keySize = (keySizeInBits + 7) / 8;
1353      TPM2B_IV         iv;
1354
1355      iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
1356      // If there is decryption to do...
1357      if(iv.t.size > 0)
1358      {
1359          // Generate key and iv
1360          CryptKDFa(hash, key, "CFB", nonceCaller, nonceTpm,
1361                    keySizeInBits + (iv.t.size * 8), symParmString, NULL);
1362          MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size,
1363                     sizeof(iv.t.buffer));
1364
1365          CryptSymmetricDecrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB,
1366                                symParmString, &iv, dataSize, data);
1367      }
1368      return;
1369  }
```

9.16.8.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
1370  void
1371  ParmEncryptSym(
1372      TPM_ALG_ID           symAlg,           // IN: symmetric algorithm
1373      TPM_ALG_ID           hash,             // IN: hash algorithm for KDFa
1374      UINT16               keySizeInBits,    // IN: AES key size in bits
1375      TPM2B               *key,              // IN: KDF HMAC key
1376      TPM2B               *nonceCaller,      // IN: nonce caller
1377      TPM2B               *nonceTpm,         // IN: nonce TPM
1378      UINT32               dataSize,         // IN: size of parameter buffer
1379      BYTE                *data              // OUT: buffer to be encrypted
1380  )
1381  {
1382      // KDF output buffer
1383      // It contains parameters for the CFB encryption
1384      BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
1385
1386      // Symmetric key size in bytes
```

```
1387        UINT16          keySize = (keySizeInBits + 7) / 8;
1388
1389        TPM2B_IV        iv;
1390
1391        iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
1392        // See if there is any encryption to do
1393        if(iv.t.size > 0)
1394        {
1395            // Generate key and iv
1396            CryptKDFa(hash, key, "CFB", nonceTpm, nonceCaller,
1397                      keySizeInBits + (iv.t.size * 8), symParmString, NULL);
1398
1399            MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size,
1400                       sizeof(iv.t.buffer));
1401
1402            CryptSymmetricEncrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB,
1403                                  symParmString, &iv, dataSize, data);
1404        }
1405        return;
1406    }
```

9.16.8.3 CryptGenerateKeySymmetric()

This function derives a symmetric cipher key from the provided seed.

Error Returns	Meaning
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area

```
1407    static TPM_RC
1408    CryptGenerateKeySymmetric(
1409        TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
1410                                                    //         for the new key.
1411        TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN:  sensitive creation data
1412        TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
1413        TPM_ALG_ID               hashAlg,           // IN: hash algorithm for the KDF
1414        TPM2B_SEED              *seed,              // IN: seed used in creation
1415        TPM2B_NAME              *name               // IN: name of the object
1416    )
1417    {
1418        // If this is not a new key, then the provided key data must be the right size
1419        if(     publicArea->objectAttributes.sensitiveDataOrigin == CLEAR
1420           &&       (sensitiveCreate->data.t.size * 8)
1421                !=  publicArea->parameters.symDetail.sym.keyBits.sym
1422          )
1423            return TPM_RC_KEY_SIZE;
1424
1425        // Make sure that the key size is OK.
1426        // This implementation only supports symmetric key sizes that are
1427        // multiples of 8
1428        if(publicArea->parameters.symDetail.sym.keyBits.sym % 8 != 0)
1429            return TPM_RC_KEY_SIZE;
1430
1431        if(publicArea->objectAttributes.sensitiveDataOrigin == SET)
1432        {
1433            // Create new symmetric key
1434            sensitive->sensitive.sym.t.size =
1435                (publicArea->parameters.symDetail.sym.keyBits.sym + 7)/8;
1436
1437            CryptKDFa(hashAlg, &seed->b, "sensitive", &name->b,
1438                      NULL, publicArea->parameters.symDetail.sym.keyBits.sym,
1439                      sensitive->sensitive.sym.t.buffer, NULL);
1440        }
```

```
1441        else
1442        {
1443            // Copy input symmetric key to sensitive area if the size is right
1444            MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
1445                         sizeof(sensitive->sensitive.sym.t.buffer));
1446        }
1447
1448        // Compute obfuscation.  Parent handle is not available and not needed for
1449        // symmetric object at this point.  TPM_RH_UNASSIGNED is passed at the
1450        // place of parent handle
1451        CryptComputeSymValue(TPM_RH_UNASSIGNED, publicArea, sensitive, seed,
1452                             hashAlg, name);
1453
1454        // Create unique area in public
1455        CryptComputeSymmetricUnique(publicArea->nameAlg,
1456                                    sensitive, &publicArea->unique.sym);
1457
1458        return TPM_RC_SUCCESS;
1459    }
```

9.16.8.4 CryptXORObfuscation()

This function implements XOR obfuscation. It should not be called if the hash algorithm is not implemented. The only return value from this function is TPM_RC_SUCCESS.

```
1460    #ifdef TPM_ALG_KEYEDHASH //% 5
1461    void
1462    CryptXORObfuscation(
1463        TPM_ALG_ID           hash,              // IN: hash algorithm for KDF
1464        TPM2B               *key,               // IN: KDF key
1465        TPM2B               *contextU,          // IN: contextU
1466        TPM2B               *contextV,          // IN: contextV
1467        UINT32               dataSize,          // IN: size of data buffer
1468        BYTE                *data               // IN/OUT: data to be XORed in place
1469    )
1470    {
1471        BYTE             mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
1472        BYTE            *pm;
1473        UINT32           i;
1474        UINT32           counter = 0;
1475        UINT16           hLen = CryptGetHashDigestSize(hash);
1476        UINT32           requestSize = dataSize * 8;
1477        INT32            remainBytes = (INT32) dataSize;
1478
1479        pAssert((key != NULL) && (data != NULL) && (hLen != 0));
1480
1481        // Call KDFa to generate XOR mask
1482        for(; remainBytes > 0; remainBytes -= hLen)
1483        {
1484            // Make a call to KDFa to get next iteration
1485            CryptKDFaOnce(hash, key, "XOR", contextU, contextV,
1486                          requestSize, mask, &counter);
1487
1488            // XOR next piece of the data
1489            pm = mask;
1490            for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
1491                *data++ ^= *pm++;
1492        }
1493        return;
1494    }
1495    #endif //TPM_ALG_KEYED_HASH //%5
```

9.16.9 Initialization and shut down

9.16.9.1 CryptInitUnits()

This function is called when the TPM receives a _TPM_Init() indication. After function returns, the hash algorithms should be available.

NOTE: The hash algorithms do not have to be tested, they just need to be available. They have to be tested before the TPM can accept HMAC authorization or return any result that relies on a hash algorithm.

```
1496  void
1497  CryptInitUnits(void)
1498  {
1499      // Call crypto engine unit initialization
1500      // We assume crypt engine initialization should always succeed.  Otherwise,
1501      // TPM should go to failure mode.
1502
1503      // This is used to make sure that the correct version of CryptoEngine
1504      // has been linked
1505      if(_cpri__InitCryptoUnits())
1506          FAIL(FATAL_ERROR_INTERNAL);
1507      return;
1508  }
```

9.16.9.2 CryptStopUnits()

This function is only used in a simulated environment. There should be no reason to shut down the cryptography on an actual TPM other than loss of power. After receiving TPM2_Startup(), the TPM should be able to accept commands until it loses power and, unless the TPM is in Failure Mode, the cryptographic algorithms should be available.

```
1509  void
1510  CryptStopUnits(void)
1511  {
1512      // Call crypto engine unit stopping
1513      _cpri__StopCryptoUnits();
1514
1515      return;
1516  }
```

9.16.9.3 CryptUtilStartup()

This function is called by TPM2_Startup() to initialize the functions in this crypto library and in the provided CryptoEngine(). In this implementation, the only initialization required in this library is initialization of the Commit nonce on TPM Reset.

This function returns false if some problem prevents the functions from starting correctly. The TPM should go into failure mode.

```
1517  BOOL
1518  CryptUtilStartup(
1519      STARTUP_TYPE         type               // IN: the startup type
1520  )
1521  {
1522      // Make sure that the crypto library functions are ready.
1523      // NOTE: need to initialize the crypto before loading
1524      // the RND state may trigger a self-test which
1525      // uses the
1526      if( !_cpri__Startup())
1527          return FALSE;
```

```
1528
1529        // Initialize the state of the RNG.
1530        CryptDrbgGetPutState(PUT_STATE);
1531
1532
1533        if(type == SU_RESET)
1534        {
1535    #ifdef TPM_ALG_ECDAA
1536
1537            // Get a new  random commit nonce
1538            gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
1539            _cpri__GenerateRandom(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
1540            // Reset the counter and commit array
1541            gr.commitCounter = 0;
1542            MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
1543    #endif // TPM_ALG_ECDAA
1544        }
1545
1546        // If the shutdown was orderly, then the values recovered from NV will
1547        // be OK to use. If the shutdown was not orderly, then a TPM Reset was required
1548        // and we would have initialized in the code above.
1549
1550        return TRUE;
1551    }
```

9.16.10 Algorithm-Independent Functions

9.16.10.1 Introduction

These functions are used generically when a function of a general type (e. g. , symmetric encryption) is required. The functions will modify the parameters as required to interface to the indicated algorithms.

9.16.10.2 CryptIsAsymAlgorithm()

This function indicates if an algorithm is an asymmetric algorithm.

Return Value	Meaning
TRUE	if it is an asymmetric algorithm
FALSE	if it is not an asymmetric algorithm

```
1552    BOOL
1553    CryptIsAsymAlgorithm(
1554        TPM_ALG_ID          algID               // IN: algorithm ID
1555    )
1556    {
1557        return (
1558    #ifdef TPM_ALG_RSA
1559                algID ==  TPM_ALG_RSA
1560    #endif
1561    #if defined TPM_ALG_RSA && defined TPM_ALG_ECC
1562                ||
1563    #endif
1564    #ifdef TPM_ALG_ECC
1565                algID == TPM_ALG_ECC
1566    #endif
1567              );
1568    }
```

9.16.10.3 CryptGetSymmetricBlockSize()

This function returns the size in octets of the symmetric encryption block used by an algorithm and key size combination.

```
1569  INT16
1570  CryptGetSymmetricBlockSize(
1571      TPMI_ALG_SYM          algorithm,          // IN: symmetric algorithm
1572      UINT16                keySize             // IN: key size in bit
1573  )
1574  {
1575      return _cpri__GetSymmetricBlockSize(algorithm, keySize);
1576  }
```

9.16.10.4 CryptSymmetricEncrypt()

This function does in-place encryption of a buffer using the indicated symmetric algorithm, key, IV, and mode. If the symmetric algorithm and mode are not defined, the TPM will fail.

```
1577  void
1578  CryptSymmetricEncrypt(
1579      BYTE                 *encrypted,          // OUT: the encrypted data
1580      TPM_ALG_ID            algorithm,          // IN: algorithm for encryption
1581      UINT16                keySizeInBits,      // IN: key size in bits
1582      TPMI_ALG_SYM_MODE     mode,               // IN: symmetric encryption mode
1583      BYTE                 *key,                // IN: encryption key
1584      TPM2B_IV             *ivIn,               // IN/OUT: Input IV and output chaining
1585                                                //         value for the next block
1586      UINT32                dataSize,           // IN: data size in byte
1587      BYTE*                 data                // IN/OUT: data buffer
1588  )
1589  {
1590      BYTE                 *iv = NULL;
1591      BYTE                  defaultIV[sizeof(TPMT_HA)];
1592
1593      pAssert(   ((mode == TPM_ALG_ECB) && (ivIn->t.size == 0))
1594              || (mode != TPM_ALG_ECB));
1595      if(
1596  #ifdef TPM_ALG_AES
1597            algorithm == TPM_ALG_AES
1598  #endif
1599  #if defined TPM_ALG_AES && defined TPM_ALG_SM4
1600         ||
1601  #endif
1602  #ifdef  TPM_ALG_SM4
1603            algorithm == TPM_ALG_SM4
1604  #endif
1605      )
1606      {
1607          // Both SM4 and AES have block size of 128 bits
1608          // If the iv is not provided, create a default of 0
1609          if(ivIn == NULL)
1610          {
1611              // Initialize the default IV
1612              iv = defaultIV;
1613              MemorySet(defaultIV, 0, 16);
1614          }
1615          else
1616          {
1617              // A provided IV has to be the right size
1618              pAssert(mode == TPM_ALG_ECB || ivIn->t.size == 16);
1619              iv = &(ivIn->t.buffer[0]);
1620          }
```

```
1621        }
1622        switch(algorithm)
1623        {
1624    #ifdef TPM_ALG_AES
1625            case TPM_ALG_AES:
1626            {
1627                switch (mode)
1628                {
1629                    case TPM_ALG_CTR:
1630                        _cpri__AESEncryptCTR(encrypted, keySizeInBits, key, iv,
1631                                             dataSize, data);
1632                        break;
1633                    case TPM_ALG_OFB:
1634                        _cpri__AESEncryptOFB(encrypted, keySizeInBits, key, iv,
1635                                             dataSize, data);
1636                        break;
1637                    case TPM_ALG_CBC:
1638                        _cpri__AESEncryptCBC(encrypted, keySizeInBits, key, iv,
1639                                             dataSize, data);
1640                        break;
1641                    case TPM_ALG_CFB:
1642                        _cpri__AESEncryptCFB(encrypted, keySizeInBits, key, iv,
1643                                             dataSize, data);
1644                        break;
1645                    case TPM_ALG_ECB:
1646                        _cpri__AESEncryptECB(encrypted, keySizeInBits, key,
1647                                             dataSize, data);
1648                        break;
1649                    default:
1650                        pAssert(0);
1651                }
1652            }
1653            break;
1654    #endif
1655    #ifdef TPM_ALG_SM4
1656            case TPM_ALG_SM4:
1657            {
1658                switch (mode)
1659                {
1660                    case TPM_ALG_CTR:
1661                        _cpri__SM4EncryptCTR(encrypted, keySizeInBits, key, iv,
1662                                             dataSize, data);
1663                        break;
1664                    case TPM_ALG_OFB:
1665                        _cpri__SM4EncryptOFB(encrypted, keySizeInBits, key, iv,
1666                                             dataSize, data);
1667                        break;
1668                    case TPM_ALG_CBC:
1669                        _cpri__SM4EncryptCBC(encrypted, keySizeInBits, key, iv,
1670                                             dataSize, data);
1671                        break;
1672
1673                    case TPM_ALG_CFB:
1674                        _cpri__SM4EncryptCFB(encrypted, keySizeInBits, key, iv,
1675                                             dataSize, data);
1676                        break;
1677                    case TPM_ALG_ECB:
1678                        _cpri__SM4EncryptECB(encrypted, keySizeInBits, key,
1679                                             dataSize, data);
1680                        break;
1681                    default:
1682                        pAssert(0);
1683                }
1684            }
1685            break;
1686
```

```
#endif
        default:
            pAssert(FALSE);
            break;
    }

    return;

}
```

9.16.10.5 CryptSymmetricDecrypt()

This function does in-place decryption of a buffer using the indicated symmetric algorithm, key, IV, and mode. If the symmetric algorithm and mode are not defined, the TPM will fail.

```
void
CryptSymmetricDecrypt(
    BYTE                *decrypted,
    TPM_ALG_ID           algorithm,         // IN: algorithm for encryption
    UINT16               keySizeInBits,     // IN: key size in bits
    TPMI_ALG_SYM_MODE    mode,              // IN: symmetric encryption mode
    BYTE                *key,               // IN: encryption key
    TPM2B_IV            *ivIn,              // IN/OUT: IV for next block
    UINT32               dataSize,          // IN: data size in byte
    BYTE*                data               // IN/OUT: data buffer
)
{
    BYTE                *iv = NULL;
    BYTE                 defaultIV[sizeof(TPMT_HA)];

    if(
#ifdef TPM_ALG_AES
          algorithm == TPM_ALG_AES
#endif
#if defined TPM_ALG_AES && defined TPM_ALG_SM4
       ||
#endif
#ifdef  TPM_ALG_SM4
          algorithm == TPM_ALG_SM4
#endif
      )
    {
        // Both SM4 and AES have block size of 128 bits
        // If the iv is not provided, create a default of 0
        if(ivIn == NULL)
        {
            // Initialize the default IV
            iv = defaultIV;
            MemorySet(defaultIV, 0, 16);
        }
        else
        {
            // A provided IV has to be the right size
            pAssert(mode == TPM_ALG_ECB || ivIn->t.size == 16);
            iv = &(ivIn->t.buffer[0]);
        }
    }


    switch(algorithm)
    {
#ifdef TPM_ALG_AES
    case TPM_ALG_AES:
    {
```

```
1745            switch (mode)
1746            {
1747                case TPM_ALG_CTR:
1748                    _cpri__AESDecryptCTR(decrypted, keySizeInBits, key, iv,
1749                                         dataSize, data);
1750                    break;
1751                case TPM_ALG_OFB:
1752                    _cpri__AESDecryptOFB(decrypted, keySizeInBits, key, iv,
1753                                         dataSize, data);
1754                    break;
1755                case TPM_ALG_CBC:
1756                    _cpri__AESDecryptCBC(decrypted, keySizeInBits, key, iv,
1757                                         dataSize, data);
1758                    break;
1759                case TPM_ALG_CFB:
1760                    _cpri__AESDecryptCFB(decrypted, keySizeInBits, key, iv,
1761                                         dataSize, data);
1762                    break;
1763                case TPM_ALG_ECB:
1764                    _cpri__AESDecryptECB(decrypted, keySizeInBits, key,
1765                                         dataSize, data);
1766                    break;
1767                default:
1768                    pAssert(0);
1769            }
1770            break;
1771        }
1772    #endif //TPM_ALG_AES
1773    #ifdef TPM_ALG_SM4
1774        case TPM_ALG_SM4 :
1775            switch (mode)
1776            {
1777                case TPM_ALG_CTR:
1778                    _cpri__SM4DecryptCTR(decrypted, keySizeInBits, key, iv,
1779                                         dataSize, data);
1780                    break;
1781                case TPM_ALG_OFB:
1782                    _cpri__SM4DecryptOFB(decrypted, keySizeInBits, key, iv,
1783                                         dataSize, data);
1784                    break;
1785                case TPM_ALG_CBC:
1786                    _cpri__SM4DecryptCBC(decrypted, keySizeInBits, key, iv,
1787                                         dataSize, data);
1788                    break;
1789                case TPM_ALG_CFB:
1790                    _cpri__SM4DecryptCFB(decrypted, keySizeInBits, key, iv,
1791                                         dataSize, data);
1792                    break;
1793                case TPM_ALG_ECB:
1794                    _cpri__SM4DecryptECB(decrypted, keySizeInBits, key,
1795                                         dataSize, data);
1796                    break;
1797                default:
1798                    pAssert(0);
1799            }
1800            break;
1801    #endif //TPM_ALG_SM4
1802
1803        default:
1804            pAssert(FALSE);
1805            break;
1806        }
1807        return;
1808    }
```

DRAFT

9.16.10.6 CryptSecretEncrypt()

This function creates a secret value and its associated secret structure using an asymmetric algorithm.

This function is used by TPM2_MakeCredential().

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*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

```
1809  TPM_RC
1810  CryptSecretEncrypt(
1811      TPMI_DH_OBJECT           keyHandle,  // IN: encryption key handle
1812      const char              *label,      // IN: a null-terminated string as L
1813      TPM2B_DATA              *data,       // OUT: secret value
1814      TPM2B_ENCRYPTED_SECRET  *secret      // OUT: secret structure
1815  )
1816  {
1817      TPM_RC       result = TPM_RC_SUCCESS;
1818      OBJECT      *encryptKey = ObjectGet(keyHandle);  // TPM key used for encrypt
1819
1820      pAssert(data != NULL && secret != NULL);
1821
1822      // The output secret value has the size of the digest produced by the nameAlg.
1823      data->t.size = CryptGetHashDigestSize(encryptKey->publicArea.nameAlg);
1824
1825      pAssert(encryptKey->publicArea.objectAttributes.decrypt == SET);
1826
1827      switch(encryptKey->publicArea.type)
1828      {
1829  #ifdef TPM_ALG_RSA
1830          case TPM_ALG_RSA:
1831          {
1832              TPMT_RSA_DECRYPT            scheme;
1833
1834              // Use OAEP scheme
1835              scheme.scheme = TPM_ALG_OAEP;
1836              scheme.details.oaep.hashAlg = encryptKey->publicArea.nameAlg;
1837
1838              // Create secret data from RNG
1839              CryptGenerateRandom(data->t.size, data->t.buffer);
1840
1841              // Encrypt the data by RSA OAEP into encrypted secret
1842              result = CryptEncryptRSA(&secret->t.size, secret->t.secret,
1843                                       encryptKey, &scheme,
1844                                       data->t.size, data->t.buffer, label);
1845          }
1846          break;
1847  #endif //TPM_ALG_RSA
1848
1849  #ifdef TPM_ALG_ECC
1850          case TPM_ALG_ECC:
1851          {
1852              TPMS_ECC_POINT      eccPublic;
1853              TPM2B_ECC_PARAMETER eccPrivate;
1854              TPMS_ECC_POINT      eccSecret;
1855              BYTE               *buffer = secret->t.secret;
1856
1857              // Need to make sure that the public point of the key is on the
```

```
1858                // curve defined by the key.
1859                if(!_cpri__EccIsPointOnCurve(
1860                            encryptKey->publicArea.parameters.eccDetail.curveID,
1861                            &encryptKey->publicArea.unique.ecc))
1862                    result = TPM_RC_KEY;
1863                else
1864                {
1865
1866                    // Call crypto engine to create an auxiliary ECC key
1867                    // We assume crypt engine initialization should always success.
1868                    // Otherwise, TPM should go to failure mode.
1869                    CryptNewEccKey(encryptKey->publicArea.parameters.eccDetail.curveID,
1870                                   &eccPublic, &eccPrivate);
1871
1872                    // Marshal ECC public to secret structure. This will be used by the
1873                    // recipient to decrypt the secret with their private key.
1874                    secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
1875
1876                    // Compute ECDH shared secret which is R = [d]Q where d is the
1877                    // private part of the ephemeral key and Q is the public part of a
1878                    // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
1879                    // because the auxiliary ECC key is just created according to the
1880                    // parameters of input ECC encrypt key.
1881                    if(    CryptEccPointMultiply(&eccSecret,
1882                                   encryptKey->publicArea.parameters.eccDetail.curveID,
1883                                   &eccPrivate,
1884                                   &encryptKey->publicArea.unique.ecc)
1885                       != CRYPT_SUCCESS)
1886                        result = TPM_RC_KEY;
1887                    else
1888
1889                        // The secret value is computed from Z using KDFe as:
1890                        // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
1891                        // Where:
1892                        //  HashID  the nameAlg of the decrypt key
1893                        //  Z   the x coordinate (Px) of the product (P) of the point
1894                        //      (Q) of the secret and the private x coordinate (de,V)
1895                        //      of the decryption key
1896                        //  Use a null-terminated string containing "SECRET"
1897                        //  PartyUInfo  the x coordinate of the point in the secret
1898                        //              (Qe,U )
1899                        //  PartyVInfo  the x coordinate of the public key (Qs,V )
1900                        //  bits    the number of bits in the digest of HashID
1901                        // Retrieve seed from KDFe
1902
1903                        CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
1904                                  label, &eccPublic.x.b,
1905                                  &encryptKey->publicArea.unique.ecc.x.b,
1906                                  data->t.size * 8, data->t.buffer);
1907                }
1908            }
1909            break;
1910    #endif //TPM_ALG_ECC
1911
1912        default:
1913            FAIL(FATAL_ERROR_INTERNAL);
1914            break;
1915        }
1916
1917        return result;
1918    }
```

DRAFT

9.16.10.7 CryptSecretDecrypt()

Decrypt a secret value by asymmetric (or symmetric) algorithm This function is used for ActivateCredential() and Import for asymmetric decryption, and StartAuthSession() for both asymmetric and symmetric decryption process

Error Returns	Meaning
TPM_RC_ATTRIBUTES	RSA key is not a decryption key
TPM_RC_BINDING	Invalid RSA key (public and private parts are not cryptographically bound.
TPM_RC_ECC_POINT	ECC point in the secret is not on the curve
TPM_RC_INSUFFICIENT	failed to retrieve ECC point from the secret
TPM_RC_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

```
1919  TPM_RC
1920  CryptSecretDecrypt(
1921      TPM_HANDLE               tpmKey,          // IN: decrypt key
1922      TPM2B_NONCE             *nonceCaller,     // IN: nonceCaller.  It is needed for
1923                                                //     symmetric decryption.  For
1924                                                //     asymmetric decryption, this
1925                                                //     parameter is NULL
1926      const char               *label,          // IN: a null-terminated string as L
1927      TPM2B_ENCRYPTED_SECRET   *secret,         // IN: input secret
1928      TPM2B_DATA               *data            // OUT: decrypted secret value
1929  )
1930  {
1931      TPM_RC      result = TPM_RC_SUCCESS;
1932      OBJECT      *decryptKey = ObjectGet(tpmKey);   //TPM key used for decrypting
1933
1934      // Decryption for secret
1935      switch(decryptKey->publicArea.type)
1936      {
1937
1938  #ifdef TPM_ALG_RSA
1939          case TPM_ALG_RSA:
1940          {
1941              TPMT_RSA_DECRYPT        scheme;
1942
1943              // Use OAEP scheme
1944              scheme.scheme = TPM_ALG_OAEP;
1945              scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
1946
1947              // Set the output buffer capacity
1948              data->t.size = sizeof(data->t.buffer);
1949
1950              // Decrypt seed by RSA OAEP
1951              result = CryptDecryptRSA(&data->t.size, data->t.buffer, decryptKey,
1952                                       &scheme,
1953                                       secret->t.size, secret->t.secret,label);
1954              if(   (result == TPM_RC_SUCCESS)
1955                 && (data->t.size
1956                      > CryptGetHashDigestSize(decryptKey->publicArea.nameAlg)))
1957                  result = TPM_RC_VALUE;
```

```
1958            }
1959            break;
1960    #endif //TPM_ALG_RSA
1961
1962    #ifdef TPM_ALG_ECC
1963            case TPM_ALG_ECC:
1964            {
1965                TPMS_ECC_POINT       eccPublic;
1966                TPMS_ECC_POINT       eccSecret;
1967                BYTE                *buffer = secret->t.secret;
1968                INT32                size = secret->t.size;
1969
1970                // Retrieve ECC point from secret buffer
1971                result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
1972                if(result == TPM_RC_SUCCESS)
1973                {
1974                    result = CryptEccPointMultiply(&eccSecret,
1975                                   decryptKey->publicArea.parameters.eccDetail.curveID,
1976                                   &decryptKey->sensitive.sensitive.ecc,
1977                                   &eccPublic);
1978
1979                    if(result == TPM_RC_SUCCESS)
1980                    {
1981
1982                        // Set the size of the "recovered" secret value to be the size
1983                        // of the digest produced by the nameAlg.
1984                        data->t.size =
1985                                CryptGetHashDigestSize(decryptKey->publicArea.nameAlg);
1986
1987                        // The secret value is computed from Z using KDFe as:
1988                        // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
1989                        // Where:
1990                        //  HashID -- the nameAlg of the decrypt key
1991                        //  Z --  the x coordinate (Px) of the product (P) of the point
1992                        //        (Q) of the secret and the private x coordinate (de,V)
1993                        //        of the decryption key
1994                        //  Use -- a null-terminated string containing "SECRET"
1995                        //  PartyUInfo -- the x coordinate of the point in the secret
1996                        //                (Qe,U )
1997                        //  PartyVInfo -- the x coordinate of the public key (Qs,V )
1998                        //  bits -- the number of bits in the digest of HashID
1999                        // Retrieve seed from KDFe
2000                        CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
2001                                  &eccPublic.x.b,
2002                                  &decryptKey->publicArea.unique.ecc.x.b,
2003                                  data->t.size * 8, data->t.buffer);
2004                    }
2005                }
2006            }
2007            break;
2008    #endif //TPM_ALG_ECC
2009
2010            case TPM_ALG_KEYEDHASH:
2011                // The seed size can not be bigger than the digest size of nameAlg
2012                if(secret->t.size >
2013                        CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
2014                    result = TPM_RC_VALUE;
2015                else
2016                {
2017                    // Retrieve seed by XOR Obfuscation:
2018                    //    seed = XOR(secret, hash, key, nonceCaller, nullNonce)
2019                    //    where:
2020                    //    secret  the secret parameter from the TPM2_StartAuthHMAC
2021                    //            command
2022                    //            which contains the seed value
2023                    //    hash    nameAlg  of tpmKey
```

```
2024                //    key     the key or data value in the object referenced by
2025                //            entityHandle in the TPM2_StartAuthHMAC command
2026                //    nonceCaller the parameter from the TPM2_StartAuthHMAC command
2027                //    nullNonce   a zero-length nonce
2028                // XOR Obfuscation in place
2029                CryptXORObfuscation(decryptKey->publicArea.nameAlg,
2030                                    &decryptKey->sensitive.sensitive.bits.b,
2031                                    &nonceCaller->b, NULL,
2032                                    secret->t.size, secret->t.secret);
2033                // Copy decrypted seed
2034                MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
2035            }
2036            break;
2037        case TPM_ALG_SYMCIPHER:
2038            {
2039                TPM2B_IV                iv = {0};
2040                TPMT_SYM_DEF_OBJECT     *symDef;
2041                // The seed size can not be bigger than the digest size of nameAlg
2042                if(secret->t.size >
2043                        CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
2044                    result = TPM_RC_VALUE;
2045                else
2046                {
2047                    symDef = &decryptKey->publicArea.parameters.symDetail.sym;
2048                    iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
2049                                                           symDef->keyBits.sym);
2050                    pAssert(iv.t.size != 0);
2051                    if(nonceCaller->t.size >= iv.t.size)
2052                        MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size,
2053                                   sizeof(iv.t.buffer));
2054                    else
2055                        MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
2056                                   nonceCaller->t.size, sizeof(iv.t.buffer));
2057                    // CFB decrypt in place, using nonceCaller as iv
2058                    CryptSymmetricDecrypt(secret->t.secret, symDef->algorithm,
2059                                          symDef->keyBits.sym, TPM_ALG_CFB,
2060                                          decryptKey->sensitive.sensitive.sym.t.buffer,
2061                                          &iv, secret->t.size, secret->t.secret);
2062
2063                    // Copy decrypted seed
2064                    MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
2065                }
2066            }
2067            break;
2068        default:
2069            pAssert(0);
2070            break;
2071    }
2072    return result;
2073 }
```

9.16.10.8 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```
2074 void
2075 CryptParameterEncryption(
2076     TPM_HANDLE           handle,            // IN: encrypt session handle
2077     TPM2B               *nonceCaller,       // IN: nonce caller
2078     UINT16               leadingSizeInByte, // IN: the size of the leading size
2079                                             //     field in bytes
2080     TPM2B_AUTH          *extraKey,          // IN: additional key material other
2081                                             //     than session auth
2082     BYTE                *buffer             // IN/OUT: parameter buffer to be
```

```
2083                                                //         encrypted
2084    )
2085    {
2086        SESSION     *session = SessionGet(handle);  // encrypt session
2087        TPM2B_TYPE(SYM_KEY, (  sizeof(extraKey->t.buffer)
2088                             + sizeof(session->sessionKey.t.buffer)));
2089        TPM2B_SYM_KEY        key;                   // encryption key
2090        UINT32               cipherSize = 0;        // size of cipher text
2091
2092        pAssert(session->sessionKey.t.size + extraKey->t.size <= <K>sizeof(key.t.buffer));
2093
2094        // Retrieve encrypted data size.
2095        if(leadingSizeInByte == 2)
2096        {
2097            // Extract the first two bytes as the size field as the data size
2098            // encrypt
2099            cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
2100            // advance the buffer
2101            buffer = &buffer[2];
2102        }
2103    #ifdef      TPM4B
2104        else if(leadingSizeInByte == 4)
2105        {
2106            // use the first four bytes to indicate the number of bytes to encrypt
2107            cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
2108            //advance pointer
2109            buffer = &buffer[4];
2110        }
2111    #endif
2112        else
2113        {
2114            pAssert(FALSE);
2115        }
2116
2117        // Compute encryption key by concatenating sessionAuth with extra key
2118        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
2119        MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
2120
2121        if (session->symmetric.algorithm == TPM_ALG_XOR)
2122
2123            // XOR parameter encryption formulation:
2124            //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
2125            CryptXORObfuscation(session->authHashAlg, &(key.b),
2126                                &(session->nonceTPM.b),
2127                                nonceCaller, cipherSize, buffer);
2128        else
2129            ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
2130                           session->symmetric.keyBits.aes, &(key.b),
2131                           nonceCaller, &(session->nonceTPM.b),
2132                           cipherSize, buffer);
2133        return;
2134    }
```

9.16.10.9 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.

```
2135    TPM_RC
2136    CryptParameterDecryption(
```

```
2137        TPM_HANDLE           handle,             // IN: encrypted session handle
2138        TPM2B               *nonceCaller,        // IN: nonce caller
2139        UINT32               bufferSize,         // IN: size of parameter buffer
2140        UINT16               leadingSizeInByte,  // IN: the size of the leading size
2141                                                 //     field in byte
2142        TPM2B_AUTH          *extraKey,           // IN: the authValue
2143        BYTE                *buffer              // IN/OUT: parameter buffer to be
2144                                                 //         decrypted
2145    )
2146    {
2147        SESSION         *session = SessionGet(handle);  // encrypt session
2148        // The hmac key is going to be the concatenation of the session key and any
2149        // additional key material (like the authValue). The size of both of these
2150        // is the size of the buffer which can contain a TPMT_HA.
2151        TPM2B_TYPE(HMAC_KEY, (  sizeof(extraKey->t.buffer)
2152                              + sizeof(session->sessionKey.t.buffer)));
2153        TPM2B_HMAC_KEY          key;             // decryption key
2154        UINT32                  cipherSize = 0; // size of cipher text
2155
2156        pAssert(session->sessionKey.t.size + extraKey->t.size <= <K>sizeof(key.t.buffer));
2157
2158        // Retrieve encrypted data size.
2159        if(leadingSizeInByte == 2)
2160        {
2161            // The first two bytes of the buffer are the size of the
2162            // data to be decrypted
2163            cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
2164            buffer = &buffer[2];   // advance the buffer
2165        }
2166    #ifdef  TPM4B
2167        else if(leadingSizeInByte == 4)
2168        {
2169            // the leading size is four bytes so get the four byte size field
2170            cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
2171            buffer = &buffer[4];   //advance pointer
2172        }
2173    #endif
2174        else
2175        {
2176            pAssert(FALSE);
2177        }
2178        if(cipherSize > bufferSize)
2179            return TPM_RC_SIZE;
2180
2181        // Compute decryption key by concatenating sessionAuth with extra input key
2182        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
2183        MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
2184
2185        if(session->symmetric.algorithm == TPM_ALG_XOR)
2186            // XOR parameter decryption fo%rmulation:
2187            //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
2188            // Call XOR obfuscation function
2189            CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
2190                                &(session->nonceTPM.b), cipherSize, buffer);
2191        else
2192            // Assume that it is one of the symmetric block ciphers.
2193            ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
2194                           session->symmetric.keyBits.sym,
2195                           &key.b, nonceCaller, &session->nonceTPM.b,
2196                           cipherSize, buffer);
2197
2198        return TPM_RC_SUCCESS;
2199
2200    }
```

9.16.10.10 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```
2201  void
2202  CryptComputeSymmetricUnique(
2203      TPMI_ALG_HASH        nameAlg,           // IN: object name algorithm
2204      TPMT_SENSITIVE      *sensitive,         // IN: sensitive area
2205      TPM2B_DIGEST        *unique             // OUT: unique buffer
2206  )
2207  {
2208      HASH_STATE  hashState;
2209
2210      pAssert(sensitive != NULL || unique != NULL);
2211
2212      // Compute the public value as the hash of sensitive.symkey || unique.buffer
2213      unique->t.size = CryptGetHashDigestSize(nameAlg);
2214      CryptStartHash(nameAlg, &hashState);
2215
2216      // Add obfuscation value
2217      CryptUpdateDigest2B(&hashState, &sensitive->seedValue.b);
2218
2219      // Add sensitive value
2220      CryptUpdateDigest2B(&hashState, &sensitive->sensitive.any.b);
2221
2222      CryptCompleteHash2B(&hashState, &unique->b);
2223
2224      return;
2225  }
```

9.16.10.11 CryptComputeSymValue()

This function computes the *seedValue* field in sensitive. It contains the obfuscation value for symmetric object and a seed value for storage key.

```
2226  void
2227  CryptComputeSymValue(
2228      TPM_HANDLE           parentHandle,      // IN: parent handle of the
2229                                              // object to be created
2230      TPMT_PUBLIC         *publicArea,        // IN/OUT: the public area template
2231      TPMT_SENSITIVE      *sensitive,         // IN: sensitive area
2232      TPM2B_SEED          *seed,              // IN: the seed
2233      TPMI_ALG_HASH        hashAlg,           // IN: hash algorithm for KDFa
2234      TPM2B_NAME          *name               // IN: object name
2235  )
2236  {
2237      TPM2B_AUTH   *proof = NULL;
2238
2239      if(CryptIsAsymAlgorithm(publicArea->type))
2240      {
2241          // Generate seedValue only when an asymmetric key is a storage key
2242          if(publicArea->objectAttributes.decrypt == SET
2243                  && publicArea->objectAttributes.restricted == SET)
2244          {
2245              // If this is a primary object in the endorsement hierarchy, use
2246              // ehProof in the creation of the symmetric seed so that child
2247              // objects in the endorsement hierarchy are voided on TPM2_Clear()
2248              // or TPM2_ChangeEPS()
2249              if(   parentHandle == TPM_RH_ENDORSEMENT
2250                 && publicArea->objectAttributes.fixedTPM == SET)
2251                  proof = &gp.ehProof;
2252          }
2253          else
```

```
2254            {
2255                sensitive->seedValue.t.size = 0;
2256                return;
2257            }
2258        }
2259
2260        // For all the object type, the size of seedValue is the digest size of nameAlg
2261        sensitive->seedValue.t.size = CryptGetHashDigestSize(publicArea->nameAlg);
2262
2263        // Compute seedValue using KDFa
2264        CryptKDFa(hashAlg,
2265                  &seed->b,
2266                  "seedValue",                  // This string is a vendor-
2267                                                // specific information
2268                  &name->b,                     // computed from the public template
2269                  proof,
2270                  sensitive->seedValue.t.size * 8,
2271                  sensitive->seedValue.t.buffer, NULL);
2272
2273        return;
2274
2275    }
```

9.16.10.12 CryptCreateObject()

This function creates an object. It:

a) fills in the created key in public and sensitive area;

b) creates a random number in sensitive area for symmetric keys; and

c) compute the unique id in public area for symmetric keys.

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

```
2276    TPM_RC
2277    CryptCreateObject(
2278        TPM_HANDLE                parentHandle,      // IN/OUT: indication of the
2279                                                     //         seed source
2280        TPMT_PUBLIC              *publicArea,        // IN/OUT: public area
2281        TPMS_SENSITIVE_CREATE    *sensitiveCreate,   // IN: sensitive creation
2282        TPMT_SENSITIVE           *sensitive          // OUT: sensitive area
2283    )
2284    {
2285        // Next value is a placeholder for a random seed that is used in
2286        // key creation when the parent is not a primary seed. It has the same
2287        // size as the primary seed.
2288
2289        TPM2B_SEED       localSeed;     // data to seed key creation if this
2290                                        // is not a primary seed
2291
2292        TPM2B_SEED      *seed = NULL;
2293        TPM_RC           result = TPM_RC_SUCCESS;
2294
```

```
2295        TPM2B_NAME       name;
2296        TPM_ALG_ID       hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
2297        OBJECT          *parent;
2298        UINT32           counter;
2299
2300        // Set the sensitive type for the object
2301        sensitive->sensitiveType = publicArea->type;
2302        ObjectComputeName(publicArea, &name);
2303
2304        // For all objects, copy the initial auth data
2305        sensitive->authValue = sensitiveCreate->userAuth;
2306
2307        // If this is a permanent handle assume that it is a hierarchy
2308        if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
2309        {
2310            seed = HierarchyGetPrimarySeed(parentHandle);
2311        }
2312        else
2313        {
2314            // If not hierarchy handle, get parent
2315            parent = ObjectGet(parentHandle);
2316            hashAlg = parent->publicArea.nameAlg;
2317
2318            // Use random value as seed for non-primary objects
2319            localSeed.t.size = PRIMARY_SEED_SIZE;
2320            CryptGenerateRandom(PRIMARY_SEED_SIZE, localSeed.t.buffer);
2321            seed = &localSeed;
2322        }
2323
2324        switch(publicArea->type)
2325        {
2326    #ifdef TPM_ALG_RSA
2327            // Create RSA key
2328        case TPM_ALG_RSA:
2329            result = CryptGenerateKeyRSA(publicArea, sensitive,
2330                                         hashAlg, seed, &name, &counter);
2331            break;
2332    #endif // TPM_ALG_RSA
2333
2334    #ifdef TPM_ALG_ECC
2335            // Create ECC key
2336        case TPM_ALG_ECC:
2337            result = CryptGenerateKeyECC(publicArea, sensitive,
2338                                         hashAlg, seed, &name, &counter);
2339            break;
2340    #endif // TPM_ALG_ECC
2341
2342            // Collect symmetric key information
2343        case TPM_ALG_SYMCIPHER:
2344            return CryptGenerateKeySymmetric(publicArea, sensitiveCreate,
2345                                             sensitive, hashAlg, seed, &name);
2346            break;
2347        case TPM_ALG_KEYEDHASH:
2348            return CryptGenerateKeyedHash(publicArea, sensitiveCreate,
2349                                          sensitive, hashAlg, seed, &name);
2350            break;
2351        default:
2352            pAssert(0);
2353            break;
2354        }
2355        if(result == TPM_RC_SUCCESS)
2356            // Only asymmetric keys should reach here
2357            CryptComputeSymValue(parentHandle, publicArea, sensitive, seed,
2358                                 hashAlg, &name);
2359
2360        return result;
```

```
}
```

9.16.10.13 CryptObjectIsPublicConsistent()

This function checks that the key sizes in the public area are consistent. For an asymmetric key, the size of the public key must match the size indicated by the public->parameters.

Checks for the algorithm types matching the key type are handled by the unmarshaling operation.

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```
BOOL
CryptObjectIsPublicConsistent(
    TPMT_PUBLIC         *publicArea           // IN: public area
)
{
    BOOL                 OK = TRUE;
    switch (publicArea->type)
    {
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            OK = CryptAreKeySizesConsistent(publicArea);
            break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            {
                const ECC_CURVE                *curveValue;

                // Check that the public point is on the indicated curve.
                OK = CryptEccIsPointOnCurve(
                               publicArea->parameters.eccDetail.curveID,
                               &publicArea->unique.ecc);
                if(OK)
                {
                    curveValue = CryptEccGetCurveDataPointer(
                                        publicArea->parameters.eccDetail.curveID);
                    pAssert(curveValue != NULL);

                    // The input ECC curve must be a supported curve
                    // IF a scheme is defined for the curve, then that scheme must
                    // be used.
                    OK =    (curveValue->sign.scheme == TPM_ALG_NULL
                         || (   publicArea->parameters.eccDetail.scheme.scheme
                             == curveValue->sign.scheme));
                    OK = OK && CryptAreKeySizesConsistent(publicArea);
                }
            }
            break;
#endif //TPM_ALG_ECC

        default:
            // Symmetric object common checks
            // There is noting to check with a symmetric key that is public only.
            // Also not sure that there is anything useful to be done with it
            // either.
            break;
    }
```

```
2411        return OK;
2412    }
```

9.16.10.14 CryptObjectPublicPrivateMatch()

This function checks the cryptographic binding between the public and sensitive areas.

Error Returns	Meaning
TPM_RC_TYPE	the type of the public and private areas are not the same
TPM_RC_FAILURE	crypto error
TPM_RC_BINDING	the public and private areas are not cryptographically matched.

```
2413    TPM_RC
2414    CryptObjectPublicPrivateMatch(
2415        OBJECT              *object     // IN: the object to check
2416    )
2417    {
2418        TPMT_PUBLIC         *publicArea;
2419        TPMT_SENSITIVE      *sensitive;
2420        TPM_RC               result = TPM_RC_SUCCESS;
2421        BOOL                 isAsymmetric = FALSE;
2422
2423        pAssert(object != NULL);
2424        publicArea = &object->publicArea;
2425        sensitive = &object->sensitive;
2426        if(publicArea->type != sensitive->sensitiveType)
2427            return TPM_RC_TYPE;
2428
2429        switch(publicArea->type)
2430        {
2431    #ifdef TPM_ALG_RSA
2432        case TPM_ALG_RSA:
2433            isAsymmetric = TRUE;
2434            // The public and private key sizes need to be consistent
2435            if(sensitive->sensitive.rsa.t.size != publicArea->unique.rsa.t.size/2)
2436                result = TPM_RC_BINDING;
2437            else
2438            // Load key by computing the private exponent
2439                result = CryptLoadPrivateRSA(object);
2440            break;
2441    #endif
2442    #ifdef TPM_ALG_ECC
2443            // This function is called from ObjectLoad() which has already checked to
2444            // see that the public point is on the curve so no need to repeat that
2445            // check.
2446        case TPM_ALG_ECC:
2447            isAsymmetric = TRUE;
2448            if(   publicArea->unique.ecc.x.t.size
2449                    != sensitive->sensitive.ecc.t.size)
2450                result = TPM_RC_BINDING;
2451            else if(publicArea->nameAlg != TPM_ALG_NULL)
2452            {
2453                TPMS_ECC_POINT          publicToCompare;
2454                // Compute ECC public key
2455                CryptEccPointMultiply(&publicToCompare,
2456                                      publicArea->parameters.eccDetail.curveID,
2457                                      &sensitive->sensitive.ecc, NULL);
2458                // Compare ECC public key
2459                if(   (!Memory2BEqual(&publicArea->unique.ecc.x.b,
2460                                      &publicToCompare.x.b))
2461                   || (!Memory2BEqual(&publicArea->unique.ecc.y.b,
2462                                      &publicToCompare.y.b)))
```

```
2463                    result = TPM_RC_BINDING;
2464            }
2465            break;
2466    #endif
2467        case TPM_ALG_KEYEDHASH:
2468            break;
2469        case TPM_ALG_SYMCIPHER:
2470            if(   (publicArea->parameters.symDetail.sym.keyBits.sym + 7)/8
2471               != sensitive->sensitive.sym.t.size)
2472                result = TPM_RC_BINDING;
2473            break;
2474        default:
2475            // The choice here is an assert or a return of a bad type for the object
2476            pAssert(0);
2477            break;
2478        }
2479
2480        // For asymmetric keys, the algorithm for validating the linkage between
2481        // the public and private areas is algorithm dependent. For symmetric keys
2482        // the linkage is based on hashing the symKey and obfuscation values.
2483        if(   result == TPM_RC_SUCCESS && !isAsymmetric
2484           && publicArea->nameAlg != TPM_ALG_NULL)
2485        {
2486            TPM2B_DIGEST    uniqueToCompare;
2487
2488            // Compute unique for symmetric key
2489            CryptComputeSymmetricUnique(publicArea->nameAlg, sensitive,
2490                                        &uniqueToCompare);
2491            // Compare unique
2492            if(!Memory2BEqual(&publicArea->unique.sym.b,
2493                              &uniqueToCompare.b))
2494                result = TPM_RC_BINDING;
2495        }
2496        return result;
2497
2498    }
```

9.16.10.15 CryptGetSignHashAlg()

Get the hash algorithm of signature from a TPMT_SIGNATURE structure. It assumes the signature is not NULL This is a function for easy access

```
2499    TPMI_ALG_HASH
2500    CryptGetSignHashAlg(
2501        TPMT_SIGNATURE      *auth               // IN: signature
2502    )
2503    {
2504        pAssert(auth->sigAlg != TPM_ALG_NULL);
2505
2506        // Get authHash algorithm based on signing scheme
2507        switch(auth->sigAlg)
2508        {
2509
2510    #ifdef  TPM_ALG_RSA
2511            case TPM_ALG_RSASSA:
2512                return auth->signature.rsassa.hash;
2513
2514            case TPM_ALG_RSAPSS:
2515                return auth->signature.rsapss.hash;
2516
2517        #endif //TPM_ALG_RSA
2518
2519        #ifdef TPM_ALG_ECC
2520            case TPM_ALG_ECDSA:
```

```
2521                return auth->signature.ecdsa.hash;
2522
2523        #endif //TPM_ALG_ECC
2524
2525            case TPM_ALG_HMAC:
2526                return auth->signature.hmac.hashAlg;
2527
2528            default:
2529                return TPM_ALG_NULL;
2530        }
2531    }
```

9.16.10.16 CryptIsSplitSign()

This function us used to determine if the signing operation is a split signing operation that required a TPM2_Commit().

```
2532    BOOL
2533    CryptIsSplitSign(
2534        TPM_ALG_ID          scheme              // IN: the algorithm selector
2535    )
2536    {
2537        if(   scheme != scheme
2538    #   ifdef   TPM_ALG_ECDAA
2539           || scheme == TPM_ALG_ECDAA
2540    #   endif   // TPM_ALG_ECDAA
2541
2542
2543          )
2544            return TRUE;
2545        return FALSE;
2546    }
```

9.16.10.17 CryptIsSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```
2547    BOOL
2548    CryptIsSignScheme(
2549        TPMI_ALG_ASYM_SCHEME    scheme
2550    )
2551    {
2552        BOOL            isSignScheme = FALSE;
2553
2554        switch(scheme)
2555        {
2556    #ifdef TPM_ALG_RSA
2557            // If RSA is implemented, then both signing schemes are required
2558        case TPM_ALG_RSASSA:
2559        case TPM_ALG_RSAPSS:
2560            isSignScheme = TRUE;
2561            break;
2562    #endif //TPM_ALG_RSA
2563
2564    #ifdef TPM_ALG_ECC
2565            // If ECC is implemented ECDSA is required
2566        case TPM_ALG_ECDSA:
2567    #ifdef  TPM_ALG_ECDAA
2568            // ECDAA is optional
2569        case TPM_ALG_ECDAA:
2570    #endif
2571    #ifdef   TPM_ALG_ECSCHNORR
2572            // Schnorr is also optional
```

```
2573        case TPM_ALG_ECSCHNORR:
2574    #endif
2575    #ifdef  TPM_ALG_SM2
2576        case TPM_ALG_SM2:
2577    #endif
2578            isSignScheme = TRUE;
2579            break;
2580    #endif //TPM_ALG_ECC
2581        default:
2582            break;
2583        }
2584        return isSignScheme;
2585    }
```

9.16.10.18 CryptIsDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
2586    BOOL
2587    CryptIsDecryptScheme(
2588        TPMI_ALG_ASYM_SCHEME    scheme
2589    )
2590    {
2591        BOOL        isDecryptScheme = FALSE;
2592
2593        switch(scheme)
2594        {
2595    #ifdef TPM_ALG_RSA
2596            // If RSA is implemented, then both decrypt schemes are required
2597        case TPM_ALG_RSAES:
2598        case TPM_ALG_OAEP:
2599             isDecryptScheme = TRUE;
2600            break;
2601    #endif //TPM_ALG_RSA
2602
2603    #ifdef TPM_ALG_ECC
2604            // If ECC is implemented ECDH is required
2605        case TPM_ALG_ECDH:
2606    #ifdef TPM_ALG_SM2
2607        case TPM_ALG_SM2:
2608    #endif
2609    #ifdef  TPM_ALG_ECMQV
2610        case TPM_ALG_ECMQV:
2611    #endif
2612            isDecryptScheme = TRUE;
2613            break;
2614    #endif //TPM_ALG_ECC
2615        default:
2616            break;
2617        }
2618        return isDecryptScheme;
2619    }
```

9.16.10.19 CryptSelectSignScheme()

This function is used by the attestation and signing commands. It implements the rules for selecting the signature scheme to use in signing. This function requires that the signing key either be TPM_RH_NULL or be loaded.

If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both object and input scheme has a non-NULL scheme algorithm, if the schemes are compatible, the input scheme will be chosen.

Error Returns	Meaning
TPM_RC_KEY	key referenced by *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*

```
2620    TPM_RC
2621    CryptSelectSignScheme(
2622        TPMI_DH_OBJECT       signHandle,        // IN: handle of signing key
2623        TPMT_SIG_SCHEME     *scheme             // IN/OUT: signing scheme
2624    )
2625    {
2626        OBJECT              *signObject;
2627        TPMT_SIG_SCHEME     *objectScheme;
2628        TPMT_PUBLIC         *publicArea;
2629        TPM_RC               result = TPM_RC_SUCCESS;
2630
2631        // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
2632        // of the setting of scheme
2633        if(signHandle == TPM_RH_NULL)
2634        {
2635            scheme->scheme = TPM_ALG_NULL;
2636            scheme->details.any.hashAlg = TPM_ALG_NULL;
2637        }
2638        else
2639        {
2640            // sign handle is not NULL so...
2641            // Get sign object pointer
2642            signObject = ObjectGet(signHandle);
2643            publicArea = &signObject->publicArea;
2644
2645            // is this a signing key?
2646            if(!publicArea->objectAttributes.sign)
2647                result = TPM_RC_KEY;
2648            else
2649            {
2650                // "parms" defined to avoid long code lines.
2651                TPMU_PUBLIC_PARMS   *parms = &publicArea->parameters;
2652                if(CryptIsAsymAlgorithm(publicArea->type))
2653                    objectScheme = (TPMT_SIG_SCHEME *)&parms->asymDetail.scheme;
2654                else
2655                    objectScheme = (TPMT_SIG_SCHEME *)&parms->keyedHashDetail.scheme;
2656
2657                // If the object doesn't have a default scheme, then use the
2658                // input scheme.
2659                if(objectScheme->scheme == TPM_ALG_NULL)
2660                {
2661                    // Input and default can't both be NULL
2662                    if(scheme->scheme == TPM_ALG_NULL)
2663                        result = TPM_RC_SCHEME;
2664
2665                    // Assume that the scheme is compatible with the key. If not,
2666                    // we will generate an error in the signing operation.
2667
2668                }
2669                else if(scheme->scheme == TPM_ALG_NULL)
2670                {
2671                    // input scheme is NULL so use default
2672
2673                    // First, check to see if the default requires that the caller
2674                    // provided scheme data
2675                    if(CryptIsSplitSign(objectScheme->scheme))
2676                        result = TPM_RC_SCHEME;
```

```
2677                    else
2678                    {
2679                        scheme->scheme = objectScheme->scheme;
2680                        scheme->details.any.hashAlg = objectScheme->details.any.hashAlg;
2681                    }
2682                }
2683                else
2684                {
2685                    // Both input and object have scheme selectors
2686                    // If the scheme and the hash are not the same then...
2687                    if(   objectScheme->scheme != scheme->scheme
2688                       || (   objectScheme->details.any.hashAlg
2689                           != scheme->details.any.hashAlg))
2690                        result = TPM_RC_SCHEME;
2691                }
2692            }
2693
2694        }
2695        return result;
2696    }
```

9.16.10.20 CryptSign()

Sign a digest with asymmetric key or HMAC. This function is called by attestation commands and the generic TPM2_Sign() command. This function checks the key scheme and digest size. It does not check if the sign operation is allowed for restricted key. It should be checked before the function is called. The function will assert if the key is not a signing key.

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)

```
2697    TPM_RC
2698    CryptSign(
2699        TPMI_DH_OBJECT       signHandle,         // IN: The handle of sign key
2700        TPMT_SIG_SCHEME     *signScheme,         // IN: sign scheme.
2701        TPM2B_DIGEST        *digest,             // IN: The digest being signed
2702        TPMT_SIGNATURE      *signature           // OUT: signature
2703    )
2704    {
2705        OBJECT              *signKey = ObjectGet(signHandle);
2706        TPM_RC               result = TPM_RC_SCHEME;
2707
2708        // check if input handle is a sign key
2709        pAssert(signKey->publicArea.objectAttributes.sign == SET);
2710
2711        // Must have the private portion loaded.  This check is made during
2712        // authorization.
2713        pAssert(signKey->attributes.publicOnly == CLEAR);
2714
2715        // Initialize signature scheme
2716        signature->sigAlg = signScheme->scheme;
2717
2718        // Initialize signature hash
2719        signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
2720
2721        // perform sign operation based on different key type
2722        switch (signKey->publicArea.type)
2723        {
```

```
2724
2725    #ifdef TPM_ALG_RSA
2726            case TPM_ALG_RSA:
2727                result = CryptSignRSA(signKey, signScheme, digest, signature);
2728                break;
2729    #endif //TPM_ALG_RSA
2730
2731    #ifdef TPM_ALG_ECC
2732            case TPM_ALG_ECC:
2733                result = CryptSignECC(signKey, signScheme, digest, signature);
2734                break;
2735    #endif //TPM_ALG_ECC
2736            case TPM_ALG_KEYEDHASH:
2737                result = CryptSignHMAC(signKey, signScheme, digest, signature);
2738                break;
2739            default:
2740                break;
2741        }
2742
2743        return result;
2744    }
```

9.16.10.21 CryptVerifySignature()

This function is used to verify a signature. It is called by TPM2_VerifySignature() and TPM2_PolicySigned().

Since this operation only requires use of a public key, no consistency checks are necessary for the key to signature type because a caller can load any public key that they like with any scheme that they like. This routine simply makes sure that the signature is correct, whatever the type.

This function requires that *auth* is not a NULL pointer.

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SCHEME	the scheme is not supported

```
2745    TPM_RC
2746    CryptVerifySignature(
2747        TPMI_DH_OBJECT       keyHandle,          // IN: The handle of sign key
2748        TPM2B_DIGEST        *digest,             // IN: The digest being validated
2749        TPMT_SIGNATURE      *signature           // IN: signature
2750    )
2751    {
2752        OBJECT              *authObject = ObjectGet(keyHandle);
2753        TPMT_PUBLIC         *publicArea = &authObject->publicArea;
2754        TPM_RC               result = TPM_RC_SCHEME;
2755
2756
2757        switch (publicArea->type)
2758        {
2759
2760    #ifdef TPM_ALG_RSA
2761        case TPM_ALG_RSA:
2762            result = CryptRSAVerifySignature(authObject, digest, signature);
2763            break;
2764    #endif //TPM_ALG_RSA
2765
2766    #ifdef TPM_ALG_ECC
2767        case TPM_ALG_ECC:
2768            result = CryptECCVerifySignature(authObject, digest, signature);
2769            break;
```

```
2770
2771    #endif // TMP_ALG_ECC
2772
2773        case TPM_ALG_KEYEDHASH:
2774            result = CryptHMACVerifySignature(authObject, digest, signature);
2775            break;
2776
2777        default:
2778            break;
2779        }
2780        return result;
2781
2782    }
```

9.16.11 Math functions

9.16.11.1 CryptDivide()

This function interfaces to the math library for large number divide.

Error Returns	Meaning
TPM_RC_SIZE	*quotient* or *remainder* is too small to receive the result

```
2783    TPM_RC
2784    CryptDivide(
2785        TPM2B       *numerator,     // IN: numerator
2786        TPM2B       *denominator,   // IN: denominator
2787        TPM2B       *quotient,      // OUT: quotient = numerator / denominator.
2788        TPM2B       *remainder      // OUT: numerator mod denominator.
2789    )
2790    {
2791        pAssert(   numerator != NULL && denominator!= NULL
2792                && (quotient != NULL || remainder != NULL)
2793               );
2794        // assume denominator is not 0
2795        pAssert(denominator->size != 0);
2796
2797        return TranslateCryptErrors(_math__Div(numerator,
2798                                               denominator,
2799                                               quotient,
2800                                               remainder)
2801                                    );
2802    }
```

9.16.11.2 CryptCompare()

This function interfaces to the math library for large number, unsigned compare.

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
2803    int
2804    CryptCompare(
2805        const UINT32             aSize,          // IN: size of a
2806        const BYTE              *a,              // IN: a buffer
2807        const UINT32             bSize,          // IN: size of b
```

```
2808        const BYTE                *b                    // IN: b buffer
2809    )
2810    {
2811        return _math__uComp(aSize, a, bSize, b);
2812    }
```

9.16.11.3 CryptCompareSigned()

This function interfaces to the math library for large number, signed compare.

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
2813    int
2814    CryptCompareSigned(
2815        UINT32                aSize,              // IN: size of a
2816        BYTE                 *a,                  // IN: a buffer
2817        UINT32                bSize,              // IN: size of b
2818        BYTE                 *b                   // IN: b buffer
2819    )
2820    {
2821        return _math__Comp(aSize, a, bSize, b);
2822    }
```

9.16.12 Self Testing Functions

9.16.12.1 Introduction

Self testing mechanism is hardware dependent and is not available at a software simulator environment. So we do not really deploy a self testing mechanism here, but always gives a pseudo return for all the self-test functions. Vendors should replace these functions with implementations that perform proper self-test.

9.16.12.2 CryptSelfTest

This function is called to start a full self-test.

NOTE: the behavior in this function is NOT the correct behavior for a real TPM implementation. An artificial behavior is placed here due to the limitation of a software simulation environment. For the correct behavior, consult the part 3 specification for TPM2_SelfTest().

Error Returns	Meaning
TPM_RC_TESTING	if *fullTest* is YES

```
2823    TPM_RC
2824    CryptSelfTest(
2825        TPMI_YES_NO           fullTest            // IN: if full test is required
2826    )
2827    {
2828        if(fullTest == YES)
2829            return TPM_RC_TESTING;
2830        else
2831            return TPM_RC_SUCCESS;
2832    }
```

9.16.12.3 CryptIncrementalSelfTest

This function is used to start an incremental self-test.

Error Returns	Meaning
TPM_RC_TESTING	if *toTest* list is not empty

```
2833    TPM_RC
2834    CryptIncrementalSelfTest(
2835        TPML_ALG            *toTest,            // IN: list of algorithms to be tested
2836        TPML_ALG            *toDoList           // OUT: list of algorithms needing test
2837    )
2838    {
2839        CRYPT_RESULT        retVal;
2840        retVal = _cpri__IncrementalSelfTest(toTest, toDoList);
2841        if(TranslateCryptErrors(retVal) == TPM_RC_SUCCESS)
2842            return TPM_RC_SUCCESS;
2843        else
2844            return TPM_RC_TESTING;
2845    }
```

9.16.12.4 CryptGetTestResult

This function returns the results of a self-test function.

NOTE: the behavior in this function is NOT the correct behavior for a real TPM implementation. An artificial behavior is placed here due to the limitation of a software simulation environment. For the correct behavior, consult the part 3 specification for TPM2_GetTestResult().

```
2846    TPM_RC
2847    CryptGetTestResult(
2848        TPM2B_MAX_BUFFER    *outData            // OUT: test result data
2849    )
2850    {
2851        outData->t.size = 0;
2852        return TPM_RC_SUCCESS;
2853    }
```

9.16.13 Capability Support

9.16.13.1 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

Return Value	Meaning
YES	if no more ECC curve is available
NO	if there are more ECC curves not reported

```
2854    #ifdef TPM_ALG_ECC //% 5
2855    TPMI_YES_NO
2856    CryptCapGetECCCurve(
2857        TPM_ECC_CURVE        curveID,           // IN: the starting ECC curve
2858        UINT32               maxCount,          // IN: count of returned curves
2859        TPML_ECC_CURVE      *curveList          // OUT: ECC curve list
2860    )
2861    {
2862        TPMI_YES_NO       more = NO;
2863        UINT16            i;
```

```
2864        UINT32            count = _cpri__EccGetCurveCount();
2865        TPM_ECC_CURVE     curve;
2866
2867        // Initialize output property list
2868        curveList->count = 0;
2869
2870        // The maximum count of curves we may return is MAX_ECC_CURVES
2871        if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
2872
2873        // Scan the eccCurveValues array
2874        for(i = 0; i < count; i++)
2875        {
2876            curve = _cpri__GetCurveIdByIndex(i);
2877            // If curveID is less than the starting curveID, skip it
2878            if(curve < curveID)
2879                continue;
2880
2881            if(curveList->count < maxCount)
2882            {
2883                // If we have not filled up the return list, add more curves to
2884                // it
2885                curveList->eccCurves[curveList->count] = curve;
2886                curveList->count++;
2887            }
2888            else
2889            {
2890                // If the return list is full but we still have curves
2891                // available, report this and stop iterating
2892                more = YES;
2893                break;
2894            }
2895
2896        }
2897
2898        return more;
2899
2900    }
```

9.16.13.2 CryptCapGetEccCurveNumber()

This function returns the number of ECC curves supported by the TPM.

```
2901    UINT32
2902    CryptCapGetEccCurveNumber(void)
2903    {
2904        // There is an array that holds the curve data. Its size divided by the
2905        // size of an entry is the number of values in the table.
2906        return _cpri__EccGetCurveCount();
2907    }
2908    #endif //TPM_ALG_ECC //% 5
```

9.16.13.3 CryptAreKeySizesConsistent()

This function validates that the public key size values are consistent for an asymmetric key.

NOTE: This is not a comprehensive test of the public key.

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```
2909  BOOL
2910  CryptAreKeySizesConsistent(
2911      TPMT_PUBLIC         *publicArea         // IN: the public area to check
2912  )
2913  {
2914      BOOL            consistent = FALSE;
2915
2916      switch (publicArea->type)
2917      {
2918  #ifdef TPM_ALG_RSA
2919          case TPM_ALG_RSA:
2920              // The key size in bits is filtered by the unmarshaling
2921              consistent =  (    ((publicArea->parameters.rsaDetail.keyBits+7)/8)
2922                              == publicArea->unique.rsa.t.size);
2923              break;
2924  #endif //TPM_ALG_RSA
2925
2926  #ifdef TPM_ALG_ECC
2927          case TPM_ALG_ECC:
2928              {
2929                  UINT16           keySizeInBytes;
2930                  TPM_ECC_CURVE    curveId = publicArea->parameters.eccDetail.curveID;
2931
2932                  keySizeInBytes = CryptEccGetKeySizeInBytes(curveId);
2933
2934                  consistent =   keySizeInBytes > 0
2935                              && publicArea->unique.ecc.x.t.size <= keySizeInBytes
2936                              && publicArea->unique.ecc.y.t.size <= keySizeInBytes;
2937              }
2938              break;
2939  #endif //TPM_ALG_ECC
2940          default:
2941              break;
2942      }
2943
2944      return consistent;
2945  }
```

9.17 Ticket.c

9.17.1 Introduction

This clause contains the functions used for ticket computations.

9.17.2 Includes

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

9.17.3 Functions

9.17.3.1 TicketIsSafe()

This function indicates if producing a ticket is safe. It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE or its substring of canonical form. If so, it is not safe to produce ticket for an input buffer claiming to be TPM generated buffer

Return Value	Meaning
TRUE	It is safe to produce ticket
FALSE	It is not safe to produce ticket

```
BOOL
TicketIsSafe(
    TPM2B           *buffer
)
{
    TPM_GENERATED   valueToCompare = TPM_GENERATED_VALUE;
    BYTE            bufferToCompare[sizeof(valueToCompare)];
    BYTE            *marshalBuffer;

    // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
    // it is not safe to generate a ticket
    if(buffer->size < <K>sizeof(valueToCompare))
        return FALSE;

    marshalBuffer = bufferToCompare;
    TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);
    if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
        return FALSE;
    else
        return TRUE;
}
```

9.17.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```
void
TicketComputeVerified(
    TPMI_RH_HIERARCHY    hierarchy,          // IN: hierarchy constant for ticket
    TPM2B_DIGEST        *digest,            // IN: digest
    TPM2B_NAME          *keyName,           // IN: name of key that signed the
                                            //     values
    TPMT_TK_VERIFIED    *ticket             // OUT: verified ticket
)
{
```

```
32      TPM2B_AUTH          *proof;
33      HMAC_STATE           hmacState;
34
35      // Fill in ticket fields
36      ticket->tag = TPM_ST_VERIFIED;
37      ticket->hierarchy = hierarchy;
38
39      // Use the proof value of the hierarchy
40      proof = HierarchyGetProof(hierarchy);
41
42      // Start HMAC
43      ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
44                                               &proof->b, &hmacState);
45
46      // add TPM_ST_VERIFIED
47      CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
48
49      // add digest
50      CryptUpdateDigest2B(&hmacState, &digest->b);
51
52      // add key name
53      CryptUpdateDigest2B(&hmacState, &keyName->b);
54
55      // complete HMAC
56      CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
57
58      return;
59  }
```

9.17.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```
60  void
61  TicketComputeAuth(
62      TPM_ST               type,              // IN: the type of ticket.
63      TPMI_RH_HIERARCHY    hierarchy,         // IN: hierarchy constant for ticket
64      UINT64               timeout,           // IN: timeout
65      TPM2B_DIGEST        *cpHashA,           // IN: input cpHashA
66      TPM2B_NONCE         *policyRef,         // IN: input policyRef
67      TPM2B_NAME          *entityName,        // IN: name of entity
68      TPMT_TK_AUTH        *ticket             // OUT: Created ticket
69  )
70  {
71      TPM2B_AUTH          *proof;
72      HMAC_STATE           hmacState;
73
74      // Get proper proof
75      proof = HierarchyGetProof(hierarchy);
76
77      // Fill in ticket fields
78      ticket->tag = type;
79      ticket->hierarchy = hierarchy;
80
81      // Start HMAC
82      ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
83                                               &proof->b, &hmacState);
84
85      // Adding TPM_ST_AUTH
86      CryptUpdateDigestInt(&hmacState, sizeof(UINT16), &ticket->tag);
87
88      // Adding timeout
89      CryptUpdateDigestInt(&hmacState, sizeof(UINT64), &timeout);
90
```

```
91      // Adding cpHash
92      CryptUpdateDigest2B(&hmacState, &cpHashA->b);
93
94      // Adding policyRef
95      CryptUpdateDigest2B(&hmacState, &policyRef->b);
96
97      // Adding keyName
98      CryptUpdateDigest2B(&hmacState, &entityName->b);
99
100     // Compute HMAC
101     CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
102
103     return;
104 }
```

9.17.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```
105 void
106 TicketComputeHashCheck(
107     TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
108     TPM2B_DIGEST        *digest,         // IN: input digest
109     TPMT_TK_HASHCHECK   *ticket          // OUT: Created ticket
110 )
111 {
112     TPM2B_AUTH          *proof;
113     HMAC_STATE           hmacState;
114
115     // Get proper proof
116     proof = HierarchyGetProof(hierarchy);
117
118     // Fill in ticket fields
119     ticket->tag = TPM_ST_HASHCHECK;
120     ticket->hierarchy = hierarchy;
121
122     ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
123                                              &proof->b, &hmacState);
124
125     // Add TPM_ST_HASHCHECK
126     CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
127
128     // Add digest
129     CryptUpdateDigest2B(&hmacState, &digest->b);
130
131     // Compute HMAC
132     CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
133
134     return;
135 }
```

9.17.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```
136 void
137 TicketComputeCreation(
138     TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy for ticket
139     TPM2B_NAME          *name,           // IN: object name
140     TPM2B_DIGEST        *creation,       // IN: creation hash
141     TPMT_TK_CREATION    *ticket          // OUT: created ticket
142 )
143 {
```

```
144         TPM2B_AUTH          *proof;
145         HMAC_STATE           hmacState;
146 
147         // Get proper proof
148         proof = HierarchyGetProof(hierarchy);
149 
150         // Fill in ticket fields
151         ticket->tag = TPM_ST_CREATION;
152         ticket->hierarchy = hierarchy;
153 
154         ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
155                                                  &proof->b, &hmacState);
156 
157         // Add TPM_ST_CREATION
158         CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);
159 
160         // Add name
161         CryptUpdateDigest2B(&hmacState, &name->b);
162 
163         // Add creation hash
164         CryptUpdateDigest2B(&hmacState, &creation->b);
165 
166         // Compute HMAC
167         CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);
168 
169         return;
170     }
```

Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are derived from the values in the annexes of part 2. This file would change based on the implementation.

The values shown in this version of the file reflect the example settings in part 2.

A.2 Implementation.h

```
#ifndef _IMPLEMENTATION_H
#define _IMPLEMENTATION_H
#include    "BaseTypes.h"
#undef TRUE
#undef FALSE
```

Change these deinnitions to turn all algorithms ir commands on or off

```
#define      ALG_YES      YES
#define      ALG_NO       NO
#define      CC_YES       YES
#define      CC_NO        NO
```

Table 3 - Definition of Base Types Table 206 - Defines for SHA1 Hash Values DefinesTable()

```
#define  SHA1_DIGEST_SIZE    20
#define  SHA1_BLOCK_SIZE     64
#define  SHA1_DER_SIZE       15
#define  SHA1_DER            {\
    0x30,0x21,0x30,0x09,0x06,0x05,0x2B,0x0E,0x03,0x02,0x1A,0x05,0x00,0x04,0x14}
```

Table 207 - Defines for SHA256 Hash Values DefinesTable()

```
#define  SHA256_DIGEST_SIZE    32
#define  SHA256_BLOCK_SIZE     64
#define  SHA256_DER_SIZE       19
#define  SHA256_DER            {\
    0x30,0x31,0x30,0x0D,0X06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X01,    \
    0X05,0X00,0X04,0X20}
```

Table 208 - Defines for SHA384 Hash Values DefinesTable()

```
#define  SHA384_DIGEST_SIZE    48
#define  SHA384_BLOCK_SIZE     128
#define  SHA384_DER_SIZE       19
#define  SHA384_DER            {\
    0x30,0x41,0x30,0x0D,0X06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X02,    \
    0X05,0X00,0X04,0X30}
```

Table 209 - Defines for SHA512 Hash Values DefinesTable()

```
#define  SHA512_DIGEST_SIZE    64
#define  SHA512_BLOCK_SIZE     128
#define  SHA512_DER_SIZE       19
#define  SHA512_DER            {\
    0x30,0x51,0x30,0x0D,0X06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X03,    \
    0X05,0X00,0X04,0X40}
```

 Copyright © TCG 2006-2013

DRAFT

Table 210 - Defines for SM3_256 Hash Values DefinesTable()

```
33  #define  SM3_256_DIGEST_SIZE    32
34  #define  SM3_256_BLOCK_SIZE     64
35  #define  SM3_256_DER_SIZE       18
36  #define  SM3_256_DER            {\
37      0x30,0x30,0x30,0x0C,0X06,0X08,0X2A,0X81,0X1C,0X81,0X45,0X01,0X83,0X11,0X05,    \
38      0X00,0X04,0X20}
```

Table 211 - Defines for Architectural Limits Values DefinesTable()

```
39  #define  MAX_SESSION_NUMBER    3
```

Table 212 - Defines for Logic Values DefinesTable()

```
40  #define  YES      1
41  #define  NO       0
42  #define  TRUE     1
43  #define  FALSE    0
44  #define  SET      1
45  #define  CLEAR    0
```

Table 213 - Defines for Processor Values DefinesTable()

```
46  #define  BIG_ENDIAN_TPM       NO
47  #define  LITTLE_ENDIAN_TPM    YES
48  #define  NO_AUTO_ALIGN        NO
```

Table 214 - Defines for Implemented Algorithms ImplementedDefines()

```
49  #define  ALG_RSA               ALG_YES
50  #define  ALG_SHA1              ALG_YES
51  #define  ALG_HMAC              ALG_YES
52  #define  ALG_AES               ALG_YES
53  #define  ALG_MGF1              ALG_YES
54  #define  ALG_XOR               ALG_YES
55  #define  ALG_KEYEDHASH         ALG_YES
56  #define  ALG_SHA256            ALG_YES
57  #define  ALG_SHA384            ALG_NO
58  #define  ALG_SHA512            ALG_NO
59  #define  ALG_SM3_256           ALG_YES
60  #define  ALG_SM4               ALG_YES
61  #define  ALG_RSASSA            (ALG_YES*ALG_RSA)
62  #define  ALG_RSAES             (ALG_YES*ALG_RSA)
63  #define  ALG_RSAPSS            (ALG_YES*ALG_RSA)
64  #define  ALG_OAEP              (ALG_YES*ALG_RSA)
65  #define  ALG_ECC               ALG_YES
66  #define  ALG_ECDH              (ALG_YES*ALG_ECC)
67  #define  ALG_ECDSA             (ALG_YES*ALG_ECC)
68  #define  ALG_ECDAA             (ALG_YES*ALG_ECC)
69  #define  ALG_SM2               (ALG_YES*ALG_ECC)
70  #define  ALG_ECSCHNORR         (ALG_YES*ALG_ECC)
71  #define  ALG_ECMQV             (ALG_NO*ALG_ECC)
72  #define  ALG_SYMCIPHER         ALG_YES
73  #define  ALG_KDF1_SP800_56a    (ALG_YES*ALG_ECC)
74  #define  ALG_KDF2              ALG_NO
75  #define  ALG_KDF1_SP800_108    ALG_YES
76  #define  ALG_CTR               ALG_YES
77  #define  ALG_OFB               ALG_YES
78  #define  ALG_CBC               ALG_YES
79  #define  ALG_CFB               ALG_YES
80  #define  ALG_ECB               ALG_YES
```

Table 215 - Defines for Implemented Commands ImplementedDefines()

```
#define  CC_ActivateCredential            CC_YES
#define  CC_Certify                       CC_YES
#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                        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                ALG_ECC
#define  CC_ECDH_KeyGen                   ALG_ECC
#define  CC_ECDH_ZGen                     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
```

```
145   #define  CC_PCR_SetAuthPolicy             CC_YES
146   #define  CC_PCR_SetAuthValue              CC_YES
147   #define  CC_PolicyAuthorize               CC_YES
148   #define  CC_PolicyAuthValue               CC_YES
149   #define  CC_PolicyCommandCode             CC_YES
150   #define  CC_PolicyCounterTimer            CC_YES
151   #define  CC_PolicyCpHash                  CC_YES
152   #define  CC_PolicyDuplicationSelect       CC_YES
153   #define  CC_PolicyGetDigest               CC_YES
154   #define  CC_PolicyLocality                CC_YES
155   #define  CC_PolicyNameHash                CC_YES
156   #define  CC_PolicyNV                      CC_YES
157   #define  CC_PolicyOR                      CC_YES
158   #define  CC_PolicyPassword                CC_YES
159   #define  CC_PolicyPCR                     CC_YES
160   #define  CC_PolicyPhysicalPresence        CC_YES
161   #define  CC_PolicyRestart                 CC_YES
162   #define  CC_PolicySecret                  CC_YES
163   #define  CC_PolicySigned                  CC_YES
164   #define  CC_PolicyTicket                  CC_YES
165   #define  CC_PP_Commands                   CC_YES
166   #define  CC_Quote                         CC_YES
167   #define  CC_ReadClock                     CC_YES
168   #define  CC_ReadPublic                    CC_YES
169   #define  CC_Rewrap                        CC_YES
170   #define  CC_RSA_Decrypt                   ALG_RSA
171   #define  CC_RSA_Encrypt                   ALG_RSA
172   #define  CC_SelfTest                      CC_YES
173   #define  CC_SequenceComplete              CC_YES
174   #define  CC_SequenceUpdate                CC_YES
175   #define  CC_SetAlgorithmSet               CC_YES
176   #define  CC_SetCommandCodeAuditStatus     CC_YES
177   #define  CC_SetPrimaryPolicy              CC_YES
178   #define  CC_Shutdown                      CC_YES
179   #define  CC_Sign                          CC_YES
180   #define  CC_StartAuthSession              CC_YES
181   #define  CC_Startup                       CC_YES
182   #define  CC_StirRandom                    CC_YES
183   #define  CC_TestParms                     CC_YES
184   #define  CC_Unseal                        CC_YES
185   #define  CC_VerifySignature               CC_YES
186   #define  CC_ZGen_2Phase                   CC_YES
187   #define  CC_EC_Ephemeral                  CC_YES
188   #define  CC_PolicyNvWritten               CC_YES
```

Table 216 - Defines for RSA Algorithm Constants DefinesTable()

```
189   #define  RSA_KEY_SIZES_BITS    {1024,2048}
190   #define  MAX_RSA_KEY_BITS      2048
191   #define  MAX_RSA_KEY_BYTES     ((MAX_RSA_KEY_BITS+7)/8)
```

Table 217 - Defines for ECC Algorithm Constants DefinesTable()

```
192   #define  ECC_CURVES            {TPM_ECC_NIST_P256,TPM_ECC_BN_P256,TPM_ECC_SM2_P256}
193   #define  ECC_KEY_SIZES_BITS    {256}
194   #define  MAX_ECC_KEY_BITS      256
195   #define  MAX_ECC_KEY_BYTES     ((MAX_ECC_KEY_BITS+7)/8)
```

Table 218 - Defines for AES Algorithm Constants DefinesTable()

```
196   #define  AES_KEY_SIZES_BITS          {128}
197   #define  MAX_AES_KEY_BITS            128
198   #define  MAX_AES_BLOCK_SIZE_BYTES    16
199   #define  MAX_AES_KEY_BYTES           ((MAX_AES_KEY_BITS+7)/8)
```

Table 219 - Defines for SM4 Algorithm Constants DefinesTable()

```
200  #define  SM4_KEY_SIZES_BITS          {128}
201  #define  MAX_SM4_KEY_BITS            128
202  #define  MAX_SM4_BLOCK_SIZE_BYTES    16
203  #define  MAX_SM4_KEY_BYTES           ((MAX_SM4_KEY_BITS+7)/8)
```

Table 220 - Defines for Symmetric Algorithm Constants DefinesTable()

```
204  #define  MAX_SYM_KEY_BITS       MAX_AES_KEY_BITS
205  #define  MAX_SYM_KEY_BYTES      MAX_AES_KEY_BYTES
206  #define  MAX_SYM_BLOCK_SIZE     MAX_AES_BLOCK_SIZE_BYTES
```

Table 221 - Defines for Implementation Values DefinesTable()

```
207  #define  FIELD_UPGRADE_IMPLEMENTED        NO
208  #define  BSIZE                            UINT16
209  #define  BUFFER_ALIGNMENT                 4
210  #define  IMPLEMENTATION_PCR               24
211  #define  PLATFORM_PCR                     24
212  #define  DRTM_PCR                         17
213  #define  HCRTM_PCR                        0
214  #define  NUM_LOCALITIES                   5
215  #define  MAX_HANDLE_NUM                   3
216  #define  MAX_ACTIVE_SESSIONS              64
217  #define  CONTEXT_SLOT                     UINT16
218  #define  CONTEXT_COUNTER                  UINT64
219  #define  MAX_LOADED_SESSIONS              3
220  #define  MAX_SESSION_NUM                  3
221  #define  MAX_LOADED_OBJECTS               3
222  #define  MIN_EVICT_OBJECTS                2
223  #define  PCR_SELECT_MIN                   ((PLATFORM_PCR+7)/8)
224  #define  PCR_SELECT_MAX                   ((IMPLEMENTATION_PCR+7)/8)
225  #define  NUM_POLICY_PCR_GROUP             1
226  #define  NUM_AUTHVALUE_PCR_GROUP          1
227  #define  MAX_CONTEXT_SIZE                 4000
228  #define  MAX_DIGEST_BUFFER                1024
229  #define  MAX_NV_INDEX_SIZE                2048
230  #define  MAX_NV_BUFFER_SIZE               1024
231  #define  MAX_CAP_BUFFER                   1024
232  #define  NV_MEMORY_SIZE                   16384
233  #define  NUM_STATIC_PCR                   16
234  #define  MAX_ALG_LIST_SIZE                64
235  #define  TIMER_PRESCALE                   100000
236  #define  PRIMARY_SEED_SIZE                32
237  #define  CONTEXT_ENCRYPT_ALG              TPM_ALG_AES
238  #define  CONTEXT_ENCRYPT_KEY_BITS         MAX_SYM_KEY_BITS
239  #define  CONTEXT_ENCRYPT_KEY_BYTES        ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
240  #define  CONTEXT_INTEGRITY_HASH_ALG       TPM_ALG_SHA256
241  #define  CONTEXT_INTEGRITY_HASH_SIZE      SHA256_DIGEST_SIZE
242  #define  PROOF_SIZE                       CONTEXT_INTEGRITY_HASH_SIZE
243  #define  NV_CLOCK_UPDATE_INTERVAL         12
244  #define  NUM_POLICY_PCR                   1
245  #define  MAX_COMMAND_SIZE                 4096
246  #define  MAX_RESPONSE_SIZE                4096
247  #define  ORDERLY_BITS                     8
248  #define  MAX_ORDERLY_COUNT                ((1<<ORDERLY_BITS)-1)
249  #define  ALG_ID_FIRST                     TPM_ALG_FIRST
250  #define  ALG_ID_LAST                      TPM_ALG_LAST
251  #define  MAX_SYM_DATA                     128
252  #define  MAX_RNG_ENTROPY_SIZE             64
253  #define  RAM_INDEX_SPACE                  512
254  #define  RSA_DEFAULT_PUBLIC_EXPONENT      0x00010001
255  #define  ENABLE_PCR_NO_INCREMENT          YES
```

 Copyright © TCG 2006-2013

```
256   #define  CRT_FORMAT_RSA                    YES
257   #define  PRIVATE_VENDOR_SPECIFIC_BYTES     \
258       ((MAX_RSA_KEY_BYTES/2)*(3+CRT_FORMAT_RSA*2))
```

Table 7 - Definition of (UINT16) TPM_ALG_ID Constants <IN/OUT, S> TPM_ALG_ID_Processing()

```
259   typedef  UINT16             TPM_ALG_ID;
260   #define  TPM_ALG_ERROR             (TPM_ALG_ID)(0x0000)
261   #define  TPM_ALG_FIRST             (TPM_ALG_ID)(0x0001)
262   #if defined ALG_RSA && ALG_RSA == YES
263   #define  TPM_ALG_RSA               (TPM_ALG_ID)(0x0001)
264   #endif
265   #define  TPM_ALG_SHA               (TPM_ALG_ID)(0x0004)
266   #if defined ALG_SHA1 && ALG_SHA1 == YES
267   #define  TPM_ALG_SHA1              (TPM_ALG_ID)(0x0004)
268   #endif
269   #if defined ALG_HMAC && ALG_HMAC == YES
270   #define  TPM_ALG_HMAC              (TPM_ALG_ID)(0x0005)
271   #endif
272   #if defined ALG_AES && ALG_AES == YES
273   #define  TPM_ALG_AES               (TPM_ALG_ID)(0x0006)
274   #endif
275   #if defined ALG_MGF1 && ALG_MGF1 == YES
276   #define  TPM_ALG_MGF1              (TPM_ALG_ID)(0x0007)
277   #endif
278   #if defined ALG_KEYEDHASH && ALG_KEYEDHASH == YES
279   #define  TPM_ALG_KEYEDHASH         (TPM_ALG_ID)(0x0008)
280   #endif
281   #if defined ALG_XOR && ALG_XOR == YES
282   #define  TPM_ALG_XOR               (TPM_ALG_ID)(0x000A)
283   #endif
284   #if defined ALG_SHA256 && ALG_SHA256 == YES
285   #define  TPM_ALG_SHA256            (TPM_ALG_ID)(0x000B)
286   #endif
287   #if defined ALG_SHA384 && ALG_SHA384 == YES
288   #define  TPM_ALG_SHA384            (TPM_ALG_ID)(0x000C)
289   #endif
290   #if defined ALG_SHA512 && ALG_SHA512 == YES
291   #define  TPM_ALG_SHA512            (TPM_ALG_ID)(0x000D)
292   #endif
293   #define  TPM_ALG_NULL              (TPM_ALG_ID)(0x0010)
294   #if defined ALG_SM3_256 && ALG_SM3_256 == YES
295   #define  TPM_ALG_SM3_256           (TPM_ALG_ID)(0x0012)
296   #endif
297   #if defined ALG_SM4 && ALG_SM4 == YES
298   #define  TPM_ALG_SM4               (TPM_ALG_ID)(0x0013)
299   #endif
300   #if defined ALG_RSASSA && ALG_RSASSA == YES
301   #define  TPM_ALG_RSASSA            (TPM_ALG_ID)(0x0014)
302   #endif
303   #if defined ALG_RSAES && ALG_RSAES == YES
304   #define  TPM_ALG_RSAES             (TPM_ALG_ID)(0x0015)
305   #endif
306   #if defined ALG_RSAPSS && ALG_RSAPSS == YES
307   #define  TPM_ALG_RSAPSS            (TPM_ALG_ID)(0x0016)
308   #endif
309   #if defined ALG_OAEP && ALG_OAEP == YES
310   #define  TPM_ALG_OAEP              (TPM_ALG_ID)(0x0017)
311   #endif
312   #if defined ALG_ECDSA && ALG_ECDSA == YES
313   #define  TPM_ALG_ECDSA             (TPM_ALG_ID)(0x0018)
314   #endif
315   #if defined ALG_ECDH && ALG_ECDH == YES
316   #define  TPM_ALG_ECDH              (TPM_ALG_ID)(0x0019)
317   #endif
```

```
#if defined ALG_ECDAA && ALG_ECDAA == YES
#define  TPM_ALG_ECDAA              (TPM_ALG_ID)(0x001A)
#endif
#if defined ALG_SM2 && ALG_SM2 == YES
#define  TPM_ALG_SM2                (TPM_ALG_ID)(0x001B)
#endif
#if defined ALG_ECSCHNORR && ALG_ECSCHNORR == YES
#define  TPM_ALG_ECSCHNORR          (TPM_ALG_ID)(0x001C)
#endif
#if defined ALG_ECMQV && ALG_ECMQV == YES
#define  TPM_ALG_ECMQV              (TPM_ALG_ID)(0x001D)
#endif
#if defined ALG_KDF1_SP800_56a && ALG_KDF1_SP800_56a == YES
#define  TPM_ALG_KDF1_SP800_56a     (TPM_ALG_ID)(0x0020)
#endif
#if defined ALG_KDF2 && ALG_KDF2 == YES
#define  TPM_ALG_KDF2               (TPM_ALG_ID)(0x0021)
#endif
#if defined ALG_KDF1_SP800_108 && ALG_KDF1_SP800_108 == YES
#define  TPM_ALG_KDF1_SP800_108     (TPM_ALG_ID)(0x0022)
#endif
#if defined ALG_ECC && ALG_ECC == YES
#define  TPM_ALG_ECC                (TPM_ALG_ID)(0x0023)
#endif
#if defined ALG_SYMCIPHER && ALG_SYMCIPHER == YES
#define  TPM_ALG_SYMCIPHER          (TPM_ALG_ID)(0x0025)
#endif
#if defined ALG_CTR && ALG_CTR == YES
#define  TPM_ALG_CTR                (TPM_ALG_ID)(0x0040)
#endif
#if defined ALG_OFB && ALG_OFB == YES
#define  TPM_ALG_OFB                (TPM_ALG_ID)(0x0041)
#endif
#if defined ALG_CBC && ALG_CBC == YES
#define  TPM_ALG_CBC                (TPM_ALG_ID)(0x0042)
#endif
#if defined ALG_CFB && ALG_CFB == YES
#define  TPM_ALG_CFB                (TPM_ALG_ID)(0x0043)
#endif
#if defined ALG_ECB && ALG_ECB == YES
#define  TPM_ALG_ECB                (TPM_ALG_ID)(0x0044)
#endif
#define  TPM_ALG_LAST               (TPM_ALG_ID)(0x0044)
```

Table 8 - Definition of (UINT16) {ECC} TPM_ECC_CURVE Constants <IN/OUT, S> EnumTable()

```
typedef  UINT16             TPM_ECC_CURVE;
#define  TPM_ECC_NONE         (TPM_ECC_CURVE)(0x0000)
#define  TPM_ECC_NIST_P192    (TPM_ECC_CURVE)(0x0001)
#define  TPM_ECC_NIST_P224    (TPM_ECC_CURVE)(0x0002)
#define  TPM_ECC_NIST_P256    (TPM_ECC_CURVE)(0x0003)
#define  TPM_ECC_NIST_P384    (TPM_ECC_CURVE)(0x0004)
#define  TPM_ECC_NIST_P521    (TPM_ECC_CURVE)(0x0005)
#define  TPM_ECC_BN_P256      (TPM_ECC_CURVE)(0x0010)
#define  TPM_ECC_BN_P638      (TPM_ECC_CURVE)(0x0011)
#define  TPM_ECC_SM2_P256     (TPM_ECC_CURVE)(0x0020)
```

Table 11 - Definition of (UINT32) TPM_CC Constants (Numeric Order) <IN/OUT, S> TPM_CC_Processing()

```
typedef  UINT32             TPM_CC;
#define  TPM_CC_FIRST                          (TPM_CC)(0x0000011F)
#define  TPM_CC_PP_FIRST                       (TPM_CC)(0x0000011F)
#if defined CC_NV_UndefineSpaceSpecial && CC_NV_UndefineSpaceSpecial == YES
#define  TPM_CC_NV_UndefineSpaceSpecial        (TPM_CC)(0x0000011F)
```

```
376   #endif
377   #if defined CC_EvictControl && CC_EvictControl == YES
378   #define  TPM_CC_EvictControl                  (TPM_CC)(0x00000120)
379   #endif
380   #if defined CC_HierarchyControl && CC_HierarchyControl == YES
381   #define  TPM_CC_HierarchyControl              (TPM_CC)(0x00000121)
382   #endif
383   #if defined CC_NV_UndefineSpace && CC_NV_UndefineSpace == YES
384   #define  TPM_CC_NV_UndefineSpace              (TPM_CC)(0x00000122)
385   #endif
386   #if defined CC_ChangeEPS && CC_ChangeEPS == YES
387   #define  TPM_CC_ChangeEPS                     (TPM_CC)(0x00000124)
388   #endif
389   #if defined CC_ChangePPS && CC_ChangePPS == YES
390   #define  TPM_CC_ChangePPS                     (TPM_CC)(0x00000125)
391   #endif
392   #if defined CC_Clear && CC_Clear == YES
393   #define  TPM_CC_Clear                         (TPM_CC)(0x00000126)
394   #endif
395   #if defined CC_ClearControl && CC_ClearControl == YES
396   #define  TPM_CC_ClearControl                  (TPM_CC)(0x00000127)
397   #endif
398   #if defined CC_ClockSet && CC_ClockSet == YES
399   #define  TPM_CC_ClockSet                      (TPM_CC)(0x00000128)
400   #endif
401   #if defined CC_HierarchyChangeAuth && CC_HierarchyChangeAuth == YES
402   #define  TPM_CC_HierarchyChangeAuth           (TPM_CC)(0x00000129)
403   #endif
404   #if defined CC_NV_DefineSpace && CC_NV_DefineSpace == YES
405   #define  TPM_CC_NV_DefineSpace                (TPM_CC)(0x0000012A)
406   #endif
407   #if defined CC_PCR_Allocate && CC_PCR_Allocate == YES
408   #define  TPM_CC_PCR_Allocate                  (TPM_CC)(0x0000012B)
409   #endif
410   #if defined CC_PCR_SetAuthPolicy && CC_PCR_SetAuthPolicy == YES
411   #define  TPM_CC_PCR_SetAuthPolicy             (TPM_CC)(0x0000012C)
412   #endif
413   #if defined CC_PP_Commands && CC_PP_Commands == YES
414   #define  TPM_CC_PP_Commands                   (TPM_CC)(0x0000012D)
415   #endif
416   #if defined CC_SetPrimaryPolicy && CC_SetPrimaryPolicy == YES
417   #define  TPM_CC_SetPrimaryPolicy              (TPM_CC)(0x0000012E)
418   #endif
419   #if defined CC_FieldUpgradeStart && CC_FieldUpgradeStart == YES
420   #define  TPM_CC_FieldUpgradeStart             (TPM_CC)(0x0000012F)
421   #endif
422   #if defined CC_ClockRateAdjust && CC_ClockRateAdjust == YES
423   #define  TPM_CC_ClockRateAdjust               (TPM_CC)(0x00000130)
424   #endif
425   #if defined CC_CreatePrimary && CC_CreatePrimary == YES
426   #define  TPM_CC_CreatePrimary                 (TPM_CC)(0x00000131)
427   #endif
428   #if defined CC_NV_GlobalWriteLock && CC_NV_GlobalWriteLock == YES
429   #define  TPM_CC_NV_GlobalWriteLock            (TPM_CC)(0x00000132)
430   #endif
431   #define  TPM_CC_PP_LAST                       (TPM_CC)(0x00000132)
432   #if defined CC_GetCommandAuditDigest && CC_GetCommandAuditDigest == YES
433   #define  TPM_CC_GetCommandAuditDigest         (TPM_CC)(0x00000133)
434   #endif
435   #if defined CC_NV_Increment && CC_NV_Increment == YES
436   #define  TPM_CC_NV_Increment                  (TPM_CC)(0x00000134)
437   #endif
438   #if defined CC_NV_SetBits && CC_NV_SetBits == YES
439   #define  TPM_CC_NV_SetBits                    (TPM_CC)(0x00000135)
440   #endif
441   #if defined CC_NV_Extend && CC_NV_Extend == YES
```

```
442   #define  TPM_CC_NV_Extend                    (TPM_CC)(0x00000136)
443   #endif
444   #if defined CC_NV_Write && CC_NV_Write == YES
445   #define  TPM_CC_NV_Write                     (TPM_CC)(0x00000137)
446   #endif
447   #if defined CC_NV_WriteLock && CC_NV_WriteLock == YES
448   #define  TPM_CC_NV_WriteLock                 (TPM_CC)(0x00000138)
449   #endif
450   #if defined CC_DictionaryAttackLockReset && CC_DictionaryAttackLockReset == YES
451   #define  TPM_CC_DictionaryAttackLockReset    (TPM_CC)(0x00000139)
452   #endif
453   #if defined CC_DictionaryAttackParameters && CC_DictionaryAttackParameters == YES
454   #define  TPM_CC_DictionaryAttackParameters   (TPM_CC)(0x0000013A)
455   #endif
456   #if defined CC_NV_ChangeAuth && CC_NV_ChangeAuth == YES
457   #define  TPM_CC_NV_ChangeAuth                (TPM_CC)(0x0000013B)
458   #endif
459   #if defined CC_PCR_Event && CC_PCR_Event == YES
460   #define  TPM_CC_PCR_Event                    (TPM_CC)(0x0000013C)
461   #endif
462   #if defined CC_PCR_Reset && CC_PCR_Reset == YES
463   #define  TPM_CC_PCR_Reset                    (TPM_CC)(0x0000013D)
464   #endif
465   #if defined CC_SequenceComplete && CC_SequenceComplete == YES
466   #define  TPM_CC_SequenceComplete             (TPM_CC)(0x0000013E)
467   #endif
468   #if defined CC_SetAlgorithmSet && CC_SetAlgorithmSet == YES
469   #define  TPM_CC_SetAlgorithmSet              (TPM_CC)(0x0000013F)
470   #endif
471   #if defined CC_SetCommandCodeAuditStatus && CC_SetCommandCodeAuditStatus == YES
472   #define  TPM_CC_SetCommandCodeAuditStatus    (TPM_CC)(0x00000140)
473   #endif
474   #if defined CC_FieldUpgradeData && CC_FieldUpgradeData == YES
475   #define  TPM_CC_FieldUpgradeData             (TPM_CC)(0x00000141)
476   #endif
477   #if defined CC_IncrementalSelfTest && CC_IncrementalSelfTest == YES
478   #define  TPM_CC_IncrementalSelfTest          (TPM_CC)(0x00000142)
479   #endif
480   #if defined CC_SelfTest && CC_SelfTest == YES
481   #define  TPM_CC_SelfTest                     (TPM_CC)(0x00000143)
482   #endif
483   #if defined CC_Startup && CC_Startup == YES
484   #define  TPM_CC_Startup                      (TPM_CC)(0x00000144)
485   #endif
486   #if defined CC_Shutdown && CC_Shutdown == YES
487   #define  TPM_CC_Shutdown                     (TPM_CC)(0x00000145)
488   #endif
489   #if defined CC_StirRandom && CC_StirRandom == YES
490   #define  TPM_CC_StirRandom                   (TPM_CC)(0x00000146)
491   #endif
492   #if defined CC_ActivateCredential && CC_ActivateCredential == YES
493   #define  TPM_CC_ActivateCredential           (TPM_CC)(0x00000147)
494   #endif
495   #if defined CC_Certify && CC_Certify == YES
496   #define  TPM_CC_Certify                      (TPM_CC)(0x00000148)
497   #endif
498   #if defined CC_PolicyNV && CC_PolicyNV == YES
499   #define  TPM_CC_PolicyNV                     (TPM_CC)(0x00000149)
500   #endif
501   #if defined CC_CertifyCreation && CC_CertifyCreation == YES
502   #define  TPM_CC_CertifyCreation              (TPM_CC)(0x0000014A)
503   #endif
504   #if defined CC_Duplicate && CC_Duplicate == YES
505   #define  TPM_CC_Duplicate                    (TPM_CC)(0x0000014B)
506   #endif
507   #if defined CC_GetTime && CC_GetTime == YES
```

 Copyright © TCG 2006-2013

```
508   #define  TPM_CC_GetTime                          (TPM_CC)(0x0000014C)
509   #endif
510   #if defined CC_GetSessionAuditDigest && CC_GetSessionAuditDigest == YES
511   #define  TPM_CC_GetSessionAuditDigest            (TPM_CC)(0x0000014D)
512   #endif
513   #if defined CC_NV_Read && CC_NV_Read == YES
514   #define  TPM_CC_NV_Read                          (TPM_CC)(0x0000014E)
515   #endif
516   #if defined CC_NV_ReadLock && CC_NV_ReadLock == YES
517   #define  TPM_CC_NV_ReadLock                      (TPM_CC)(0x0000014F)
518   #endif
519   #if defined CC_ObjectChangeAuth && CC_ObjectChangeAuth == YES
520   #define  TPM_CC_ObjectChangeAuth                 (TPM_CC)(0x00000150)
521   #endif
522   #if defined CC_PolicySecret && CC_PolicySecret == YES
523   #define  TPM_CC_PolicySecret                     (TPM_CC)(0x00000151)
524   #endif
525   #if defined CC_Rewrap && CC_Rewrap == YES
526   #define  TPM_CC_Rewrap                           (TPM_CC)(0x00000152)
527   #endif
528   #if defined CC_Create && CC_Create == YES
529   #define  TPM_CC_Create                           (TPM_CC)(0x00000153)
530   #endif
531   #if defined CC_ECDH_ZGen && CC_ECDH_ZGen == YES
532   #define  TPM_CC_ECDH_ZGen                        (TPM_CC)(0x00000154)
533   #endif
534   #if defined CC_HMAC && CC_HMAC == YES
535   #define  TPM_CC_HMAC                             (TPM_CC)(0x00000155)
536   #endif
537   #if defined CC_Import && CC_Import == YES
538   #define  TPM_CC_Import                           (TPM_CC)(0x00000156)
539   #endif
540   #if defined CC_Load && CC_Load == YES
541   #define  TPM_CC_Load                             (TPM_CC)(0x00000157)
542   #endif
543   #if defined CC_Quote && CC_Quote == YES
544   #define  TPM_CC_Quote                            (TPM_CC)(0x00000158)
545   #endif
546   #if defined CC_RSA_Decrypt && CC_RSA_Decrypt == YES
547   #define  TPM_CC_RSA_Decrypt                      (TPM_CC)(0x00000159)
548   #endif
549   #if defined CC_HMAC_Start && CC_HMAC_Start == YES
550   #define  TPM_CC_HMAC_Start                       (TPM_CC)(0x0000015B)
551   #endif
552   #if defined CC_SequenceUpdate && CC_SequenceUpdate == YES
553   #define  TPM_CC_SequenceUpdate                   (TPM_CC)(0x0000015C)
554   #endif
555   #if defined CC_Sign && CC_Sign == YES
556   #define  TPM_CC_Sign                             (TPM_CC)(0x0000015D)
557   #endif
558   #if defined CC_Unseal && CC_Unseal == YES
559   #define  TPM_CC_Unseal                           (TPM_CC)(0x0000015E)
560   #endif
561   #if defined CC_PolicySigned && CC_PolicySigned == YES
562   #define  TPM_CC_PolicySigned                     (TPM_CC)(0x00000160)
563   #endif
564   #if defined CC_ContextLoad && CC_ContextLoad == YES
565   #define  TPM_CC_ContextLoad                      (TPM_CC)(0x00000161)
566   #endif
567   #if defined CC_ContextSave && CC_ContextSave == YES
568   #define  TPM_CC_ContextSave                      (TPM_CC)(0x00000162)
569   #endif
570   #if defined CC_ECDH_KeyGen && CC_ECDH_KeyGen == YES
571   #define  TPM_CC_ECDH_KeyGen                      (TPM_CC)(0x00000163)
572   #endif
573   #if defined CC_EncryptDecrypt && CC_EncryptDecrypt == YES
```

```
574   #define  TPM_CC_EncryptDecrypt                (TPM_CC)(0x00000164)
575   #endif
576   #if defined CC_FlushContext && CC_FlushContext == YES
577   #define  TPM_CC_FlushContext                  (TPM_CC)(0x00000165)
578   #endif
579   #if defined CC_LoadExternal && CC_LoadExternal == YES
580   #define  TPM_CC_LoadExternal                  (TPM_CC)(0x00000167)
581   #endif
582   #if defined CC_MakeCredential && CC_MakeCredential == YES
583   #define  TPM_CC_MakeCredential                (TPM_CC)(0x00000168)
584   #endif
585   #if defined CC_NV_ReadPublic && CC_NV_ReadPublic == YES
586   #define  TPM_CC_NV_ReadPublic                 (TPM_CC)(0x00000169)
587   #endif
588   #if defined CC_PolicyAuthorize && CC_PolicyAuthorize == YES
589   #define  TPM_CC_PolicyAuthorize               (TPM_CC)(0x0000016A)
590   #endif
591   #if defined CC_PolicyAuthValue && CC_PolicyAuthValue == YES
592   #define  TPM_CC_PolicyAuthValue               (TPM_CC)(0x0000016B)
593   #endif
594   #if defined CC_PolicyCommandCode && CC_PolicyCommandCode == YES
595   #define  TPM_CC_PolicyCommandCode             (TPM_CC)(0x0000016C)
596   #endif
597   #if defined CC_PolicyCounterTimer && CC_PolicyCounterTimer == YES
598   #define  TPM_CC_PolicyCounterTimer            (TPM_CC)(0x0000016D)
599   #endif
600   #if defined CC_PolicyCpHash && CC_PolicyCpHash == YES
601   #define  TPM_CC_PolicyCpHash                  (TPM_CC)(0x0000016E)
602   #endif
603   #if defined CC_PolicyLocality && CC_PolicyLocality == YES
604   #define  TPM_CC_PolicyLocality                (TPM_CC)(0x0000016F)
605   #endif
606   #if defined CC_PolicyNameHash && CC_PolicyNameHash == YES
607   #define  TPM_CC_PolicyNameHash                (TPM_CC)(0x00000170)
608   #endif
609   #if defined CC_PolicyOR && CC_PolicyOR == YES
610   #define  TPM_CC_PolicyOR                      (TPM_CC)(0x00000171)
611   #endif
612   #if defined CC_PolicyTicket && CC_PolicyTicket == YES
613   #define  TPM_CC_PolicyTicket                  (TPM_CC)(0x00000172)
614   #endif
615   #if defined CC_ReadPublic && CC_ReadPublic == YES
616   #define  TPM_CC_ReadPublic                    (TPM_CC)(0x00000173)
617   #endif
618   #if defined CC_RSA_Encrypt && CC_RSA_Encrypt == YES
619   #define  TPM_CC_RSA_Encrypt                   (TPM_CC)(0x00000174)
620   #endif
621   #if defined CC_StartAuthSession && CC_StartAuthSession == YES
622   #define  TPM_CC_StartAuthSession              (TPM_CC)(0x00000176)
623   #endif
624   #if defined CC_VerifySignature && CC_VerifySignature == YES
625   #define  TPM_CC_VerifySignature               (TPM_CC)(0x00000177)
626   #endif
627   #if defined CC_ECC_Parameters && CC_ECC_Parameters == YES
628   #define  TPM_CC_ECC_Parameters                (TPM_CC)(0x00000178)
629   #endif
630   #if defined CC_FirmwareRead && CC_FirmwareRead == YES
631   #define  TPM_CC_FirmwareRead                  (TPM_CC)(0x00000179)
632   #endif
633   #if defined CC_GetCapability && CC_GetCapability == YES
634   #define  TPM_CC_GetCapability                 (TPM_CC)(0x0000017A)
635   #endif
636   #if defined CC_GetRandom && CC_GetRandom == YES
637   #define  TPM_CC_GetRandom                     (TPM_CC)(0x0000017B)
638   #endif
639   #if defined CC_GetTestResult && CC_GetTestResult == YES
```

```
640   #define  TPM_CC_GetTestResult                 (TPM_CC)(0x0000017C)
641   #endif
642   #if defined CC_Hash && CC_Hash == YES
643   #define  TPM_CC_Hash                          (TPM_CC)(0x0000017D)
644   #endif
645   #if defined CC_PCR_Read && CC_PCR_Read == YES
646   #define  TPM_CC_PCR_Read                      (TPM_CC)(0x0000017E)
647   #endif
648   #if defined CC_PolicyPCR && CC_PolicyPCR == YES
649   #define  TPM_CC_PolicyPCR                     (TPM_CC)(0x0000017F)
650   #endif
651   #if defined CC_PolicyRestart && CC_PolicyRestart == YES
652   #define  TPM_CC_PolicyRestart                 (TPM_CC)(0x00000180)
653   #endif
654   #if defined CC_ReadClock && CC_ReadClock == YES
655   #define  TPM_CC_ReadClock                     (TPM_CC)(0x00000181)
656   #endif
657   #if defined CC_PCR_Extend && CC_PCR_Extend == YES
658   #define  TPM_CC_PCR_Extend                    (TPM_CC)(0x00000182)
659   #endif
660   #if defined CC_PCR_SetAuthValue && CC_PCR_SetAuthValue == YES
661   #define  TPM_CC_PCR_SetAuthValue              (TPM_CC)(0x00000183)
662   #endif
663   #if defined CC_NV_Certify && CC_NV_Certify == YES
664   #define  TPM_CC_NV_Certify                    (TPM_CC)(0x00000184)
665   #endif
666   #if defined CC_EventSequenceComplete && CC_EventSequenceComplete == YES
667   #define  TPM_CC_EventSequenceComplete         (TPM_CC)(0x00000185)
668   #endif
669   #if defined CC_HashSequenceStart && CC_HashSequenceStart == YES
670   #define  TPM_CC_HashSequenceStart             (TPM_CC)(0x00000186)
671   #endif
672   #if defined CC_PolicyPhysicalPresence && CC_PolicyPhysicalPresence == YES
673   #define  TPM_CC_PolicyPhysicalPresence        (TPM_CC)(0x00000187)
674   #endif
675   #if defined CC_PolicyDuplicationSelect && CC_PolicyDuplicationSelect == YES
676   #define  TPM_CC_PolicyDuplicationSelect       (TPM_CC)(0x00000188)
677   #endif
678   #if defined CC_PolicyGetDigest && CC_PolicyGetDigest == YES
679   #define  TPM_CC_PolicyGetDigest               (TPM_CC)(0x00000189)
680   #endif
681   #if defined CC_TestParms && CC_TestParms == YES
682   #define  TPM_CC_TestParms                     (TPM_CC)(0x0000018A)
683   #endif
684   #if defined CC_Commit && CC_Commit == YES
685   #define  TPM_CC_Commit                        (TPM_CC)(0x0000018B)
686   #endif
687   #if defined CC_PolicyPassword && CC_PolicyPassword == YES
688   #define  TPM_CC_PolicyPassword                (TPM_CC)(0x0000018C)
689   #endif
690   #if defined CC_ZGen_2Phase && CC_ZGen_2Phase == YES
691   #define  TPM_CC_ZGen_2Phase                   (TPM_CC)(0x0000018D)
692   #endif
693   #if defined CC_EC_Ephemeral && CC_EC_Ephemeral == YES
694   #define  TPM_CC_EC_Ephemeral                  (TPM_CC)(0x0000018E)
695   #endif
696   #if defined CC_PolicyNvWritten && CC_PolicyNvWritten == YES
697   #define  TPM_CC_PolicyNvWritten               (TPM_CC)(0x0000018F)
698   #endif
699   #define  TPM_CC_LAST                          (TPM_CC)(0x0000018F)
700   #ifndef MAX
701   #define MAX(a, b) ((a) > (b) ? (a) : (b))
702   #endif
703   #define MAX_HASH_BLOCK_SIZE  (                    \
704       MAX(ALG_SHA1 * SHA1_BLOCK_SIZE,               \
705       MAX(ALG_SHA256 * SHA256_BLOCK_SIZE,           \
```

```
706         MAX(ALG_SHA384 * SHA384_BLOCK_SIZE,          \
707         MAX(ALG_SM3_256 * SM3_256_BLOCK_SIZE,        \
708         MAX(ALG_SHA512 * SHA512_BLOCK_SIZE,          \
709         0 ))))))
710     #define MAX_DIGEST_SIZE      (                   \
711         MAX(ALG_SHA1 * SHA1_DIGEST_SIZE,             \
712         MAX(ALG_SHA256 * SHA256_DIGEST_SIZE,         \
713         MAX(ALG_SHA384 * SHA384_DIGEST_SIZE,         \
714         MAX(ALG_SM3_256 * SM3_256_DIGEST_SIZE,       \
715         MAX(ALG_SHA512 * SHA512_DIGEST_SIZE,         \
716         0 ))))))
717     #if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
718     #error "Hash data not valid"
719     #endif
720     #define HASH_COUNT (ALG_SHA1+ALG_SHA256+ALG_SHA384+ALG_SM3_256+ALG_SHA512)
721     #endif  // _IMPLEMENTATION_H
```

DRAFT

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    "TPMB.h"
10  #include    "bool.h"
11  #include    "Platform.h"
12  #ifndef NULL
13  #define NULL    0
14  #endif
15  typedef UINT16  NUMBYTES;       // When a size is a number of bytes
16  typedef UINT32  NUMDIGITS;      // When a size is a number of "digits"
```

 Copyright © TCG 2006-2013

B.3.2. General Purpose Macros

```
#ifndef MAX
#   define MAX(a, b) ((a) > (b) ? (a) : b)
#endif
```

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 structue needs to start on an alignemnt 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 alighment

```
#define IsAddressAligned(address, align)                                     \
                    (((intptr_t)(address) & (align - 1)) == 0)
```

This is the structure that is used for passing a context into the hashing funcitons. 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.

```
47  typedef struct _HASH_STATE
48  {
49      ALIGNED_HASH_STATE       state;
50      TPM_ALG_ID               hashAlg;
51  } CPRI_HASH_STATE, *PCPRI_HASH_STATE;
52  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.

```
53  typedef struct {
54      ALIGNED_HASH_STATE      buffer;
55      TPM_ALG_ID              hashAlg;
56  } EXPORT_HASH_STATE;
57  typedef enum {
58      IMPORT_STATE,       // Converts externally formatted state to internal
59      EXPORT_STATE        // Converts internal formatted state to external
60  } 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.

```
61  typedef enum {
62      GET_STATE,      // Get the state to save to NV
63      PUT_STATE       // Restore the state from NV
64  } 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

```
65  #define DRBG_KEY_SIZE_BITS      128
66  #define DRBG_IV_SIZE_BITS       128
67  #define DRBG_ALGORITHM      TPM_ALG_AES
68  #if ((DRBG_KEY_SIZE_BITS % 8) != 0) || ((DRBG_IV_SIZE_BITS % 8) != 0)
69  #error "Key size and IV for DRBG must be even multiples of 8"
70  #endif
71  #if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
72  #error "Key size for DRBG must be even multiple of the cypher block size"
73  #endif
74  typedef BYTE    DRBG_SEED[(DRBG_KEY_SIZE_BITS + DRBG_IV_SIZE_BITS) / 8];
75  typedef struct {
76      UINT64      reseedCounter;
77      UINT32      magic;
78      DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
79  } DRBG_STATE, *pDRBG_STATE;
```

B.3.4.1. Asymmetric Structures and Values

```
80  #ifdef TPM_ALG_ECC
```

B.3.5. 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.

```
81   typedef struct {
82       UINT32                  curveID;       // The curve identifier
83       TPMS_ECC_POINT         *publicPoint;   // Pointer to the public point
84       TPM2B                  *privateKey;    // Pointer to the private key
85   } ECC_KEY;
86   #endif // TPM_ALG_ECC
87   #ifdef TPM_ALG_RSA
```

B.3.6. RSA-related Structures

This structure is a succinct representation of the cryptographic components of an RSA key.

```
88   typedef struct {
89       UINT32         exponent;      // The public exponent pointer
90       TPM2B         *publicKey;     // Pointer to the public modulus
91       TPM2B         *privateKey;    // The private exponent (not a prime)
92   } RSA_KEY;
93   #endif // TPM_ALG_RSA
94   #ifdef TPM_ALG_RSA
95   #   ifdef TPM_ALG_ECC
96   #       if   MAX_RSA_KEY_BYTES > MAX_ECC_KEY_BYTES
97   #           define  MAX_NUMBER_SIZE         MAX_RSA_KEY_BYTES
98   #       else
99   #           define  MAX_NUMBER_SIZE         MAX_ECC_KEY_BYTES
100  #       endif
101  #   else // RSA but no ECC
102  #       define MAX_NUMBER_SIZE             MAX_RSA_KEY_BYTES
103  #   endif
104  #elif defined TPM_ALG_ECC
105  #   define MAX_NUMBER_SIZE                MAX_ECC_KEY_BYTES
106  #else
107  #   error No assymmetric algorithm implemented.
108  #endif
109  typedef INT16     CRYPT_RESULT;
110  #define CRYPT_RESULT_MIN    INT16_MIN
111  #define CRYPT_RESULT_MAX    INT16_MAX
```

< 0	recoverable error
0	success
> 0	command specific return value (generally a digest size)

```
112  #define CRYPT_FAIL          ((CRYPT_RESULT)  1)
113  #define CRYPT_SUCCESS       ((CRYPT_RESULT)  0)
114  #define CRYPT_NO_RESULT     ((CRYPT_RESULT) -1)
115  #define CRYPT_SCHEME        ((CRYPT_RESULT) -2)
116  #define CRYPT_PARAMETER     ((CRYPT_RESULT) -3)
117  #define CRYPT_UNDERFLOW     ((CRYPT_RESULT) -4)
118  #define CRYPT_POINT         ((CRYPT_RESULT) -5)
119  #define CRYPT_CANCEL        ((CRYPT_RESULT) -6)
120  typedef UINT64              HASH_CONTEXT[MAX_HASH_STATE_SIZE/sizeof(UINT64)];
121  #include    "CpriCryptPri_fp.h"
122  #ifdef  TPM_ALG_ECC
123  #   include "CpriDataEcc.h"
124  #   include "CpriECC_fp.h"
```

```
125   #endif
126   #include    "MathFunctions_fp.h"
127   #include    "CpriRNG_fp.h"
128   #include    "CpriHash_fp.h"
129   #include    "CpriSym_fp.h"
130   #ifdef  TPM_ALG_RSA
131   #   include    "CpriRSA_fp.h"
132   #endif
133   #endif // !_CRYPT_PRI_H
```

9.18 CryptoBaseTypes.h

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

NULL definition

```
#ifndef         NULL
#define         NULL        (0)
#endif
```

Avoid include of BaseTypes.h if this file is included

```
#define _BASETYPES_H
typedef  uint8_t            UINT8;
typedef  uint8_t            BYTE;
typedef  int8_t             INT8;
typedef  int                BOOL;
typedef  uint16_t           UINT16;
typedef  int16_t            INT16;
typedef  uint64_t           UINT64;
typedef  int64_t            INT64;
typedef struct {
    UINT16        size;
    BYTE          buffer[1];
} TPM2B;
#endif
```

DRAFT

B.4 CpriData.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.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   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 acomplished by adding bytes of zero at the start of the number.

Return Value	Meaning
TRUE	number denormalized
FALSE	number already larger than the desired size

```
17  BOOL
18  _math__Denormalize2B(
19      TPM2B       *in,    // IN:OUT TPM2B number to denormalize
20      UINT32       size   // IN: the desired size
21      )
22  {
23      UINT32       to;
24      UINT32       from;
25      // If the current size is greater than the requested size, see if this can be
26      // normalized to a value smaller than the requested size and then de-normalize
27      if(in->size > size)
28      {
29          _math__Normalize2B(in);
30          if(in->size > size)
31              return FALSE;
32      }
33      // If the size is already what is requested, leave
34      if(in->size == size)
35          return TRUE;
36
37      // move the bytes to the 'right'
38      for(from = in->size, to = size; from > 0;)
39          in->buffer[--to] = in->buffer[--from];
40
41      // 'to' will always be greater than 0 because we checked for equal above.
42      for(; to > 0;)
43          in->buffer[--to] = 0;
44
45      in->size = (UINT16)size;
46      return TRUE;
47  }
```

B.5.2.3. _math__sub()

This function to subtract one unsiged value from another *c* = *a* - *b*. *c* may be the same as *a* or *b*.

Return Value	Meaning
1	if (a > b) so no borrow
0	if (a = b) so no borrow and b == a
-1	if (a < b) so there was a borrow

```
48  int
49  _math__sub(
50      const UINT32         aSize,     // IN: size of a
51      const BYTE          *a,         // IN: a
52      const UINT32         bSize,     // IN: size of b
53      const BYTE          *b,         // IN: b
54      UINT16              *cSize,     // OUT: set to MAX(aSize, bSize)
55      BYTE                *c          // OUT: the difference
56      )
```

```
{
    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

```
int
_math__Inc(
    UINT32      aSize,      // IN: size of a
    BYTE       *a           // IN: a
    )
{
```


DRAFT

```
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     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 denormalized so that the resulting size is exactly *aSize* + *bSize*. If *pSize* is provided, then the actual size of the result is returned. The initial value for *pSize* must be at least *aSize* + *pSize*.

Return Value	Meaning
< 0	indicates an error
>= 0	the size of the product

```
131     int
132     _math__Mul(
133         const UINT32     aSize,     // IN: size of a
134         const BYTE      *a,         // IN: a
135         const UINT32     bSize,     // IN: size of b
136         const BYTE      *b,         // IN: b
137         UINT32          *pSize,     // IN/OUT: size of the product
138         BYTE            *p          // OUT: product. length of product = aSize + bSize
139         )
140     {
141         BIGNUM          *bnA;
142         BIGNUM          *bnB;
143         BIGNUM          *bnP;
144         BN_CTX          *context;
145         int             retVal = 0;
146
147
148         // First check that pSize is large enough if present
149         if((pSize != NULL) && (*pSize < (aSize + bSize)))
150             return CRYPT_PARAMETER;
151         pAssert(pSize == 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
175         // If the size of the results is allowed to float, then set the return
176         // size. Otherwise, it might be necessary to denormalize the results
177         retVal = BN_num_bytes(bnP);
178         if(pSize == NULL)
179         {
180             BN_bn2bin(bnP, &p[aSize + bSize - retVal]);
181             memset(p, 0, aSize + bSize - retVal);
182             retVal = aSize + bSize;
183         }
184         else
185         {
186             BN_bn2bin(bnP, p);
187             *pSize = retVal;
188         }
189
190         BN_CTX_end(context);
191         BN_CTX_free(context);
192         return retVal;
193     }
```

B.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

```
194     CRYPT_RESULT
195     _math__Div(
196         const TPM2B         *n,         // IN: numerator
197         const TPM2B         *d,         // IN: denominator
198         TPM2B               *q,         // OUT: quotient
199         TPM2B               *r          // OUT: remainder
200         )
201     {
202         BIGNUM          *bnN;
203         BIGNUM          *bnD;
204         BIGNUM          *bnQ;
```

```
205         BIGNUM          *bnR;
206         BN_CTX          *context;
207         CRYPT_RESULT     retVal = CRYPT_SUCCESS;
208
209         // Get structures for the big number representations
210         context = BN_CTX_new();
211         if(context == NULL)
212             FAIL(FATAL_ERROR_ALLOCATION);
213         BN_CTX_start(context);
214         bnN = BN_CTX_get(context);
215         bnD = BN_CTX_get(context);
216         bnQ = BN_CTX_get(context);
217         bnR = BN_CTX_get(context);
218
219         // Errors in BN_CTX_get() are sticky so only need to check the last allocation
220         if (   bnR == NULL
221             || BN_bin2bn(n->buffer, n->size, bnN) == NULL
222             || BN_bin2bn(d->buffer, d->size, bnD) == NULL)
223                 FAIL(FATAL_ERROR_INTERNAL);
224
225         // Check for divide by zero.
226         if(BN_num_bits(bnD) == 0)
227             FAIL(FATAL_ERROR_DIVIDE_ZERO);
228
229         // Perform the division
230         if (BN_div(bnQ, bnR, bnN, bnD, context) != 1)
231             FAIL(FATAL_ERROR_INTERNAL);
232
233
234         // Convert the BIGNUM result back to our format
235         if(q != NULL)   // If the quotient is being returned
236         {
237             if(!BnTo2B(q, bnQ, q->size))
238             {
239                 retVal = CRYPT_UNDERFLOW;
240                 goto Done;
241             }
242          }
243         if(r != NULL)   // If the remainder is being returned
244         {
245             if(!BnTo2B(r, bnR, r->size))
246                 retVal = CRYPT_UNDERFLOW;
247         }
248
249     Done:
250         BN_CTX_end(context);
251         BN_CTX_free(context);
252
253         return retVal;
254     }
```

B.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)

```
255     int
256     _math__uComp(
```

```
257         const UINT32        aSize,      // IN: size of a
258         const BYTE          *a,         // IN: a
259         const UINT32        bSize,      // IN: size of b
260         const BYTE          *b          // IN: b
261         )
262     {
263         int             borrow = 0;
264         int             notZero = 0;
265         int             i;
266         // If a has more digits than b, then a is greater than b if
267         // any of the more significant bytes is non zero
268         if((i = (int)aSize - (int)bSize) > 0)
269             for(; i > 0; i--)
270                 if(*a++) // means a > b
271                     return 1;
272         // If b has more digits than a, then b is greater if any of the
273         // more significant bytes is non zero
274         if(i < 0)  <Q>// Means that b is longer than a
275             for(; i < 0; i++)
276                 if(*b++) // means that b > a
277                     return -1;
278         // Either the vales are the same size or the upper bytes of a or b are
279         // all zero, so compare the rest
280         i = (aSize > bSize) ? bSize : aSize;
281         a = &a[i-1];
282         b = &b[i-1];
283         for(; i > 0; i--)
284         {
285             borrow = *a-- - *b-- + borrow;
286             notZero = notZero || borrow;
287             borrow >>= 8;
288         }
289         // if there is a borrow, then b > a
290         if(borrow)
291             return -1;
292         // either a > b or they are the same
293         return notZero;
294     }
```

B.5.2.9. _math__Comp()

Compare two signed integers:

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
295     int
296     _math__Comp(
297         const UINT32        aSize,      // IN: size of a
298         const BYTE          *a,         // IN: a buffer
299         const UINT32        bSize,      // IN: size of b
300         const BYTE          *b          // IN: b buffer
301         )
302     {
303         int      signA, signB;       // sign of a and b
304
305         // For positive or 0, sign_a is 1
306         // for negative, sign_a is 0
307         signA = ((a[0] & 0x80) == 0) ? 1 : 0;
308
```

```
309         // For positive or 0, sign_b is 1
310         // for negative, sign_b is 0
311         signB = ((b[0] & 0x80) == 0) ? 1 : 0;
312
313         if(signA != signB)
314         {
315             return signA - signB;
316         }
317
318         if(signA == 1)
319             // do unsigned compare function
320             return _math__uComp(aSize, a, bSize, b);
321         else
322             // do unsigned compare the other way
323             return 0 - _math__uComp(aSize, a, bSize, b);
324     }
```

B.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

```
325     CRYPT_RESULT
326     _math__ModExp(
327         UINT32           cSize,      // IN: size of the results
328         BYTE            *c,          // OUT: results buffer
329         const UINT32     mSize,      // IN: size of number to be exponentiated
330         const BYTE      *m,          // IN: number to be exponentiated
331         const UINT32     eSize,      // IN: size of power
332         const BYTE      *e,          // IN: power
333         const UINT32     nSize,      // IN: modulus size
334         const BYTE      *n           // IN: modulus
335         )
336     {
337         CRYPT_RESULT     retVal = CRYPT_SUCCESS;
338         BN_CTX          *context;
339         BIGNUM          *bnC;
340         BIGNUM          *bnM;
341         BIGNUM          *bnE;
342         BIGNUM          *bnN;
343         INT32            i;
344
345         context = BN_CTX_new();
346         if(context == NULL)
347             FAIL(FATAL_ERROR_ALLOCATION);
348         BN_CTX_start(context);
349         bnC = BN_CTX_get(context);
350         bnM = BN_CTX_get(context);
351         bnE = BN_CTX_get(context);
352         bnN = BN_CTX_get(context);
353
```

```
354         // Errors for BN_CTX_get are sticky so only need to check last allocation
355         if(bnN == NULL)
356             FAIL(FATAL_ERROR_ALLOCATION);
357
358         //convert arguments
359         if (   BN_bin2bn(m, mSize, bnM) == NULL
360             || BN_bin2bn(e, eSize, bnE) == NULL
361             || BN_bin2bn(n, nSize, bnN) == NULL)
362                 FAIL(FATAL_ERROR_INTERNAL);
363
364         // Don't do exponentiation if the number being exponentiated is
365         // larger than the modulus.
366         if(BN_ucmp(bnM, bnN) >= 0)
367         {
368             retVal = CRYPT_PARAMETER;
369             goto Cleanup;
370         }
371         // Perform the exponentiation
372         if(!(BN_mod_exp(bnC, bnM, bnE, bnN, context)))
373             FAIL(FATAL_ERROR_INTERNAL);
374
375         // Convert the results
376         // Make sure that the results will fit in the provided buffer.
377         if((unsigned)BN_num_bytes(bnC) > cSize)
378         {
379             retVal = CRYPT_UNDERFLOW;
380             goto Cleanup;
381         }
382         i = cSize - BN_num_bytes(bnC);
383         BN_bn2bin(bnC, &c[i]);
384         memset(c, 0, i);
385
386     Cleanup:
387         // Free up allocated BN values
388         BN_CTX_end(context);
389         BN_CTX_free(context);
390         return retVal;
391     }
```

B.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

```
392     BOOL
393     _math__IsPrime(
394         const UINT32     prime
395         )
396     {
397         int     isPrime;
398         BIGNUM  *p;
399
400         // Assume the size variables are not overflow, which should not happen in
401         // the contexts that this function will be called.
402         if((p = BN_new()) == NULL)
403             FAIL(FATAL_ERROR_ALLOCATION);
404         if(!BN_set_word(p, prime))
405             FAIL(FATAL_ERROR_INTERNAL);
406
```

```
407         //
408         // BN_is_prime returning -1 means that it ran into an error.
409         // It should only return 0 or 1
410         //
411         if((isPrime = BN_is_prime_ex(p, BN_prime_checks, NULL, NULL)) < 0)
412             FAIL(FATAL_ERROR_INTERNAL);
413
414         if(p != NULL)
415             BN_clear_free(p);
416         return (isPrime == 1);
417     }
```

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

```
#include "OsslCryptoEngine.h"
```

B.6.3. Functions

B.6.3.1. _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 shoudl be called as a result of _TPM_Init().

```
CRYPT_RESULT
_cpri__InitCryptoUnits(void)
{
    _cpri__RngStartup();
    _cpri__HashStartup();
    _cpri__SymStartup();

#ifdef TPM_ALG_RSA
    _cpri__RsaStartup();
#endif

#ifdef TPM_ALG_ECC
    _cpri__EccStartup();
#endif

    return 0;
}
```

B.6.3.2. _cpri__StopCryptoUnits()

This function calls the shutdown functions of the other crypto modules that are part of the crypto engine for this implementation.

```
void
_cpri__StopCryptoUnits(void)
{
    return;
}
```

B.6.3.3. _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().

```
BOOL
_cpri__Startup(
    void
    )
{

```

DRAFT

```
30      return(   _cpri__HashStartup()
31             && _cpri__RngStartup()
32  #ifdef TPM_ALG_RSA
33             && _cpri__RsaStartup()
34  #endif // TPM_ALG_RSA
35  #ifdef TPM_ALG_ECC
36             && _cpri__EccStartup()
37  #endif // TPM_ALG_ECC
38             && _cpri__SymStartup());
39  }
```

B.6.3.4. _cpri__IncrementalSelfTest()

This function is used to start an incremental self-test. It always returns success.

NOTE: the behavior in this function is NOT the correct behavior for a real TPM implementation. An artificial behavior is placed here due to the limitation of a software simulation environment. For the correct behavior, consult the part 3 specification for TPM2_IncrementalSelfTest().

```
40  CRYPT_RESULT
41  _cpri__IncrementalSelfTest(
42      TPML_ALG            *toTest,            // IN: list of algorithms to be tested
43      TPML_ALG            *toDoList           // OUT: list of algorithms need test
44  )
45  {
46      // Always copy toTest list to toDoList
47      *toDoList = *toTest;
48
49      return CRYPT_SUCCESS;
50  }
```

B.7 CpriRNG.c

```
1   //#define __TPM_RNG_FOR_DEBUG__
```

B.7.1. Inroduction

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 genrator.

```
4   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.

 Copyright © TCG 2006-2013

```
35  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  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  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  }
77  #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    "CpriData.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;
```

```
#endif
    case TPM_ALG_NULL:
        return NULL;
    default:
        FAIL(FATAL_ERROR_INTERNAL);
        return NULL;
    }
}
```

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.

```
static UINT16
MarshalHashState(
    EVP_MD_CTX      *ctxt,          // IN: Context to marshal
    BYTE            *buf            // OUT: The buffer that will receive the
                                    //      context. This buffer is at least
                                    //      MAX_HASH_STATE_SIZE bytes
)
{
    // make sure everything will fit
    pAssert(ctxt->digest->ctx_size <= OSSL_HASH_STATE_DATA_SIZE);

    // Copy the context data
    memcpy(buf, (void*) ctxt->md_data, ctxt->digest->ctx_size);

    return (UINT16)ctxt->digest->ctx_size;
}
```

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).

```
static UINT16
GetHashState(
    EVP_MD_CTX      *ctxt,          // OUT: The context structure to receive
                                    //      the result of unmarshaling.
    TPM_ALG_ID       algType,       // IN: The hash algorithm selector
    BYTE            *buf            // IN: Buffer containing marshaled hash data
)
{
    EVP_MD          *evpmdAlgorithm = NULL;

    pAssert(ctxt != NULL);

    EVP_MD_CTX_init(ctxt);

    evpmdAlgorithm = GetHashServer(algType);
    if(evpmdAlgorithm == NULL)
        return 0;

    // This also allocates the ctxt->md_data
    if((EVP_DigestInit_ex(ctxt, evpmdAlgorithm, NULL)) != 1)
        FAIL(FATAL_ERROR_INTERNAL);

    pAssert(ctxt->digest->ctx_size < <K>sizeof(ALIGNED_HASH_STATE));
```

```
90      memcpy(ctxt->md_data, buf, ctxt->digest->ctx_size);
91      return (UINT16)ctxt->digest->ctx_size;
92  }
```

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.

```
93  static const HASH_INFO *
94  GetHashInfoPointer(
95      TPM_ALG_ID   hashAlg
96  )
97  {
98      UINT32 i, tableSize;
99
100     // Get the table size of g_hashData
101     tableSize = sizeof(g_hashData) / sizeof(g_hashData[0]);
102
103     for(i = 0; i < tableSize - 1; i++)
104     {
105         if(g_hashData[i].alg == hashAlg)
106             return &g_hashData[i];
107     }
108     return &g_hashData[tableSize-1];
109 }
```

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.

```
110 BOOL
111 _cpri__HashStartup(
112     void
113 )
114 {
115     // On startup, make sure that the structure sizes are compatible. It would
116     // be nice if this could be done at compile time but I couldn't figure it out.
117     CPRI_HASH_STATE *cpriState = NULL;
118 //    NUMBYTES         evpCtxSize = sizeof(EVP_MD_CTX);
119     NUMBYTES        cpriStateSize = sizeof(cpriState->state);
120 //    OSSL_HASH_STATE *osslState;
121     NUMBYTES        osslStateSize = sizeof(OSSL_HASH_STATE);
122 //    int              dataSize = sizeof(osslState->u.data);
123     pAssert(cpriStateSize >= osslStateSize);
124
125     return TRUE;
126 }
```

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

```
127  TPM_ALG_ID
128  _cpri__GetHashAlgByIndex(
129      UINT32      index       // IN: the index
130  )
131  {
132      if(index >= HASH_COUNT)
133          return TPM_ALG_NULL;
134      return g_hashData[index].alg;
135  }
```

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)

```
136  UINT16
137  _cpri__GetHashBlockSize(
138      TPM_ALG_ID  hashAlg     // IN: hash algorithm to look up
139  )
140  {
141      return GetHashInfoPointer(hashAlg)->blockSize;
142  }
```

B.8.4.4. _cpri__GetHashDER

This function returns a pointer to the DER string for the algorithm and indicates its size.

```
143  UINT16
144  _cpri__GetHashDER(
145      TPM_ALG_ID              hashAlg,    // IN: the algorithm to look up
146      const BYTE            **p
147  )
148  {
149      const HASH_INFO       *q;
150      q = GetHashInfoPointer(hashAlg);
151      *p = &q->der[0];
152      return q->derSize;
153  }
```

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

```
154  UINT16
```

```
155 _cpri__GetDigestSize(
156     TPM_ALG_ID  hashAlg     // IN: hash algorithm to look up
157 )
158 {
159     return GetHashInfoPointer(hashAlg)->digestSize;
160 }
```

B.8.4.6. _cpri__GetContextAlg()

This function returns the algorithm associated with a hash context

```
161 TPM_ALG_ID
162 _cpri__GetContextAlg(
163     CPRI_HASH_STATE         *hashState  // IN: the hash context
164 )
165 {
166     return hashState->hashAlg;
167 }
```

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.

```
168 UINT16
169 _cpri__CopyHashState (
170     CPRI_HASH_STATE    *out,        // OUT: destination of the state
171     CPRI_HASH_STATE    *in          // IN: source of the state
172 )
173 {
174     OSSL_HASH_STATE    *i = (OSSL_HASH_STATE *)&in->state;
175     OSSL_HASH_STATE    *o = (OSSL_HASH_STATE *)&out->state;
176     pAssert(sizeof(i) <= <K>sizeof(in->state));
177
178     EVP_MD_CTX_init(&o->u.context);
179     EVP_MD_CTX_copy_ex(&o->u.context, &i->u.context);
180     o->copySize = i->copySize;
181     out->hashAlg = in->hashAlg;
182     return sizeof(CPRI_HASH_STATE);
183 }
```

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

```
184 UINT16
185 _cpri__StartHash(
186     TPM_ALG_ID       hashAlg,       // IN: hash algorithm
187     BOOL             sequence,      // IN: TRUE if the state should be saved
188     CPRI_HASH_STATE *hashState      // OUT: the state of hash stack.
189 )
190 {
191     EVP_MD_CTX       localState;
```

```
192         OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
193         BYTE            *stateData = state->u.data;
194         EVP_MD_CTX      *context;
195         EVP_MD          *evpmdAlgorithm = NULL;
196         UINT16           retVal = 0;
197
198         if(sequence)
199             context = &localState;
200         else
201             context = &state->u.context;
202
203         hashState->hashAlg = hashAlg;
204
205         EVP_MD_CTX_init(context);
206         evpmdAlgorithm = GetHashServer(hashAlg);
207         if(evpmdAlgorithm == NULL)
208             goto Cleanup;
209
210         if(EVP_DigestInit_ex(context, evpmdAlgorithm, NULL) != 1)
211             FAIL(FATAL_ERROR_INTERNAL);
212         retVal = (CRYPT_RESULT)EVP_MD_CTX_size(context);
213
214     Cleanup:
215         if(retVal > 0)
216         {
217             if (sequence)
218             {
219                 if((state->copySize = MarshalHashState(context, stateData)) == 0)
220                 {
221                     // If MarshalHashState returns a negative number, it is an error
222                     // code and not a hash size so copy the error code to be the return
223                     // from this function and set the actual stateSize to zero.
224                     retVal = state->copySize;
225                     state->copySize = 0;
226                 }
227                 // Do the cleanup
228                 EVP_MD_CTX_cleanup(context);
229             }
230             else
231                 state->copySize = -1;
232         }
233         else
234             state->copySize = 0;
235         return retVal;
236     }
```

B.8.4.9. _cpri__UpdateHash()

Add data to a hash or HMAC stack.

```
237     void
238     _cpri__UpdateHash(
239         CPRI_HASH_STATE     *hashState,     // IN: the hash context information
240         UINT32               dataSize,      // IN: the size of data to be added to
241                                             //     the digest
242         BYTE                *data           // IN: data to be hashed
243     )
244     {
245         EVP_MD_CTX       localContext;
246         OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
247         BYTE            *stateData = state->u.data;
248         EVP_MD_CTX      *context;
249         CRYPT_RESULT     retVal = CRYPT_SUCCESS;
250
```

```
251         // If there is no context, return
252         if(state->copySize == 0)
253             return;
254         if(state->copySize > 0)
255         {
256             context = &localContext;
257             if((retVal = GetHashState(context, hashState->hashAlg, stateData)) <= 0)
258                 return;
259         }
260         else
261             context = &state->u.context;
262 
263         if(EVP_DigestUpdate(context, data, dataSize) != 1)
264             FAIL(FATAL_ERROR_INTERNAL);
265         else if(   state->copySize > 0
266                    && (retVal= MarshalHashState(context, stateData)) >= 0)
267         {
268             // retVal is the size of the marshaled data. Make sure that it is consistent
269             // by ensuring that we didn't get more than allowed
270             if(retVal < state->copySize)
271                 FAIL(FATAL_ERROR_INTERNAL);
272             else
273                 EVP_MD_CTX_cleanup(context);
274         }
275         return;
276     }
```

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

```
277     UINT16
278     _cpri__CompleteHash(
279         CPRI_HASH_STATE     *hashState,     // IN: the state of hash stack
280         UINT32               dOutSize,      // IN: size of digest buffer
281         BYTE                *dOut           // OUT: hash digest
282     )
283     {
284         EVP_MD_CTX       localState;
285         OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
286         BYTE            *stateData = state->u.data;
287         EVP_MD_CTX      *context;
288         UINT16           retVal;
289         int              hLen;
290         BYTE             temp[MAX_DIGEST_SIZE];
291         BYTE            *rBuffer = dOut;
292 
293         if(state->copySize == 0)
294             return 0;
295         if(state->copySize > 0)
296         {
297             context = &localState;
298             if((retVal = GetHashState(context, hashState->hashAlg, stateData)) <= 0)
299                 goto Cleanup;
300         }
301         else
```

```
302             context = &state->u.context;
303
304         hLen = EVP_MD_CTX_size(context);
305         if((unsigned)hLen > dOutSize)
306             rBuffer = temp;
307         if(EVP_DigestFinal_ex(context, rBuffer, NULL) == 1)
308         {
309             if(rBuffer != dOut)
310             {
311                 if(dOut != NULL)
312                 {
313                     memcpy(dOut, temp, dOutSize);
314                 }
315                 retVal = (UINT16)dOutSize;
316             }
317             else
318             {
319                 retVal = (UINT16)hLen;
320             }
321             state->copySize = 0;
322         }
323         else
324         {
325             retVal = 0; // Indicate that no data is returned
326         }
327     Cleanup:
328         EVP_MD_CTX_cleanup(context);
329         return retVal;
330     }
```

B.8.4.11. _cpri__ImportExportHashState()

This funciton is used to import or export the hash state. This funciton would be called to export state when a sequence object was being prepared for export

```
331     void
332     _cpri__ImportExportHashState(
333         CPRI_HASH_STATE         *osslFmt,           // IN/OUT: the hash state formated
334                                                     //          for use by openSSL
335         EXPORT_HASH_STATE       *externalFmt,       // IN/OUT: the exported hash state
336         IMPORT_EXPORT            direction          //
337         )
338     {
339         UNREFERENCED_PARAMETER(direction);
340         UNREFERENCED_PARAMETER(externalFmt);
341         UNREFERENCED_PARAMETER(osslFmt);
342         return;
343
344     #if 0
345         if(direction == IMPORT_STATE)
346         {
347             // don't have the import export functions yet so just copy
348             _cpri__CopyHashState(osslFmt, (CPRI_HASH_STATE *)externalFmt);
349         }
350         else
351         {
352             _cpri__CopyHashState((CPRI_HASH_STATE *)externalFmt, osslFmt);
353         }
354     #endif
355     }
```

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)

```
356  UINT16
357  _cpri__HashBlock(
358      TPM_ALG_ID   hashAlg,        // IN: The hash algorithm
359      UINT32       dataSize,       // IN: size of buffer to hash
360      BYTE        *data,           // IN: the buffer to hash
361      UINT32       digestSize,     // IN: size of the digest buffer
362      BYTE        *digest          // OUT: hash digest
363  )
364  {
365      EVP_MD_CTX       hashContext;
366      EVP_MD          *hashServer = NULL;
367      UINT16           retVal = 0;
368      BYTE             b[MAX_DIGEST_SIZE]; // temp buffer in case digestSize not
369      // a full digest
370      unsigned int     dSize = _cpri__GetDigestSize(hashAlg);
371
372
373      // If there is no digest to compute return
374      if(dSize == 0)
375          return 0;
376
377      // After the call to EVP_MD_CTX_init(), will need to call EVP_MD_CTX_cleanup()
378      EVP_MD_CTX_init(&hashContext);     // Initialize the local hash context
379      hashServer = GetHashServer(hashAlg); // Find the hash server
380
381      // It is an error if the digest size is non-zero but there is no server
382      if(   (hashServer == NULL)
383         || (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
384         || (EVP_DigestUpdate(&hashContext, data, dataSize) != 1))
385          FAIL(FATAL_ERROR_INTERNAL);
386      else
387      {
388          // If the size of the digest produced (dSize) is larger than the available
389          // buffer (digestSize), then put the digest in a temp buffer and only copy
390          // the most significant part into the available buffer.
391          if(dSize > digestSize)
392          {
393              if(EVP_DigestFinal_ex(&hashContext, b, &dSize) != 1)
394                  FAIL(FATAL_ERROR_INTERNAL);
395              memcpy(digest, b, digestSize);
396              retVal = (UINT16)digestSize;
397          }
398          else
399          {
400              if((EVP_DigestFinal_ex(&hashContext, digest, &dSize)) != 1)
401                  FAIL(FATAL_ERROR_INTERNAL);
402              retVal = (UINT16) dSize;
403          }
404      }
405      EVP_MD_CTX_cleanup(&hashContext);
406      return retVal;
407  }
```

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)

```
408  UINT16
409  _cpri__StartHMAC(
410      TPM_ALG_ID       hashAlg,    // IN: the algorithm to use
411      BOOL             sequence,   // IN: indicates if the state should be saved
412      CPRI_HASH_STATE *state,      // IN/OUT: the state buffer
413      UINT16           keySize,    // IN: the size of the HMAC key
414      BYTE            *key,        // IN: the HMAC key
415      TPM2B           *oPadKey     // OUT: the key prepared for the oPad round
416  )
417  {
418      CPRI_HASH_STATE  localState;
419      UINT16           blockSize = _cpri__GetHashBlockSize(hashAlg);
420      UINT16           digestSize;
421      BYTE            *pb;         // temp pointer
422      UINT32           i;
423
424      // If the key size is larger than the block size, then the hash of the key
425      // is used as the key
426      if(keySize > blockSize)
427      {
428          // large key so digest
429          if((digestSize = _cpri__StartHash(hashAlg, FALSE, &localState)) == 0)
430              return 0;
431          _cpri__UpdateHash(&localState, keySize, key);
432          _cpri__CompleteHash(&localState, digestSize, oPadKey->buffer);
433          oPadKey->size = digestSize;
434      }
435      else
436      {
437          // key size is ok
438          memcpy(oPadKey->buffer, key, keySize);
439          oPadKey->size = keySize;
440      }
441      // XOR the key with iPad (0x36)
442      pb = oPadKey->buffer;
443      for(i = oPadKey->size; i > 0; i--)
444          *pb++ ^= 0x36;
445
446      // if the keySize is smaller than a block, fill the rest with 0x36
447      for(i = blockSize - oPadKey->size; i >  0; i--)
448          *pb++ = 0x36;
449
450      // Increase the oPadSize to a full block
451      oPadKey->size = blockSize;
452
453      // Start a new hash with the HMAC key
454      // This will go in the caller's state structure and may be a sequence or not
455
456      if((digestSize = _cpri__StartHash(hashAlg, sequence, state)) > 0)
457      {
458
```

```
459             _cpri__UpdateHash(state, oPadKey->size, oPadKey->buffer);
460
461             // XOR the key block with 0x5c ^ 0x36
462             for(pb = oPadKey->buffer, i = blockSize; i > 0; i--)
463                 *pb++ ^= (0x5c ^ 0x36);
464         }
465
466         return digestSize;
467     }
```

B.8.5.2. _cpri_CompleteHMAC()

This function is called to complete an HMAC. It will finish the current digest, and start a new digest. It will then add the *oPadKey* and the completed digest and return the results in *dOut*. It will not return more than *dOutSize* bytes.

Return Value	Meaning
>= 0	number of bytes in *dOut* (may be zero)

```
468     UINT16
469     _cpri__CompleteHMAC(
470         CPRI_HASH_STATE     *hashState,     // IN: the state of hash stack
471         TPM2B               *oPadKey,       // IN: the HMAC key in oPad format
472         UINT32               dOutSize,      // IN: size of digest buffer
473         BYTE                *dOut           // OUT: hash digest
474     )
475     {
476         BYTE             digest[MAX_DIGEST_SIZE];
477         CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
478         CPRI_HASH_STATE  localState;
479         UINT16           digestSize = _cpri__GetDigestSize(state->hashAlg);
480
481
482         _cpri__CompleteHash(hashState, digestSize, digest);
483
484         // Using the local hash state, do a hash with the oPad
485         if(_cpri__StartHash(state->hashAlg, FALSE, &localState) != digestSize)
486             return 0;
487
488         _cpri__UpdateHash(&localState, oPadKey->size, oPadKey->buffer);
489         _cpri__UpdateHash(&localState, digestSize, digest);
490         return _cpri__CompleteHash(&localState, dOutSize, dOut);
491     }
```

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*

```
492     CRYPT_RESULT
493     _cpri__MGF1(
```

```
494         UINT32      mSize,      // IN: length of the mask to be produced
495         BYTE       *mask,       // OUT: buffer to receive the mask
496         TPM_ALG_ID  hashAlg,    // IN: hash to use
497         UINT32      sSize,      // IN: size of the seed
498         BYTE       *seed        // IN: seed size
499     )
500     {
501         EVP_MD_CTX           hashContext;
502         EVP_MD              *hashServer = NULL;
503         CRYPT_RESULT         retVal = 0;
504         BYTE                 b[MAX_DIGEST_SIZE]; // temp buffer in case mask is not an
505         // even multiple of a full digest
506         CRYPT_RESULT         dSize = _cpri__GetDigestSize(hashAlg);
507         unsigned int         digestSize = (UINT32)dSize;
508         UINT32               remaining;
509         UINT32               counter;
510         BYTE                 swappedCounter[4];
511
512         // Parameter check
513         if(mSize > (1024*16)) // Semi-arbitrary maximum
514             FAIL(FATAL_ERROR_INTERNAL);
515
516         // If there is no digest to compute return
517         if(dSize <= 0)
518             return 0;
519
520         EVP_MD_CTX_init(&hashContext);     // Initialize the local hash context
521         hashServer = GetHashServer(hashAlg); // Find the hash server
522         if(hashServer == NULL)
523             // If there is no server, then there is no digest
524             return 0;
525
526         for(counter = 0, remaining = mSize; remaining > 0; counter++)
527         {
528             // Because the system may be either Endian...
529             UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
530
531             // Start the hash and include the seed and counter
532             if(   (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
533                || (EVP_DigestUpdate(&hashContext, seed, sSize) != 1)
534                || (EVP_DigestUpdate(&hashContext, swappedCounter, 4) != 1)
535               )
536                 FAIL(FATAL_ERROR_INTERNAL);
537
538             // Handling the completion depends on how much space remains in the mask
539             // buffer. If it can hold the entire digest, put it there. If not
540             // put the digest in a temp buffer and only copy the amount that
541             // will fit into the mask buffer.
542             if(remaining < (<K>unsigned)dSize)
543             {
544                 if(EVP_DigestFinal_ex(&hashContext, b, &digestSize) != 1)
545                     FAIL(FATAL_ERROR_INTERNAL);
546                 memcpy(mask, b, remaining);
547                 break;
548             }
549             else
550             {
551                 if(EVP_DigestFinal_ex(&hashContext, mask, &digestSize) != 1)
552                     FAIL(FATAL_ERROR_INTERNAL);
553                 remaining -= dSize;
554                 mask = &mask[dSize];
555             }
556             retVal = (CRYPT_RESULT)mSize;
557         }
558
559         EVP_MD_CTX_cleanup(&hashContext);
```

```
560         return retVal;
561     }
```

B.8.6.2. _cpri_KDFa()

This function performs the key generation according to 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

```
562     UINT16
563     _cpri__KDFa(
564         TPM_ALG_ID   hashAlg,       // IN: hash algorithm used in HMAC
565         TPM2B       *key,           // IN: HMAC key
566         const char  *label,         // IN: a 0-byte terminated label used in KDF
567         TPM2B       *contextU,      // IN: context U
568         TPM2B       *contextV,      // IN: context V
569         UINT32       sizeInBits,    // IN: size of generated key in bits
570         BYTE        *keyStream,     // OUT: key buffer
571         UINT32      *counterInOut,  // IN/OUT: caller may provide the iteration counter
572                                     //         for incremental operations to avoid
573                                     //         large intermediate buffers.
574         BOOL         once           // IN: TRUE if only one iteration is performed
575                                     //     FALSE if iteration count determined by
576                                     //     "sizeInBits"
577     )
578     {
579         UINT32                  counter = 0;    // counter value
580         INT32                   lLen = 0;       // length of the label
581         INT16                   hLen;           // length of the hash
582         INT16                   bytes;          // number of bytes to produce
583         BYTE                   *stream = keyStream;
584         BYTE                    marshaledUint32[4];
585         CPRI_HASH_STATE         hashState;
586         TPM2B_MAX_HASH_BLOCK    hmacKey;
587
588         pAssert(key != NULL && keyStream != NULL);
589         pAssert(once == FALSE || (sizeInBits & 7) == 0);
590
591         if(counterInOut != NULL)
592             counter = *counterInOut;
593
594         // Prepare label buffer.  Calculate its size and keep the last 0 byte
595         if(label != NULL)
596             for(lLen = 0; label[lLen++] != 0; );
597
598         // Get the hash size.  If it is less than or 0, either the
599         // algorithm is not supported or the hash is TPM_ALG_NULL
```

```
600         // In either case the digest size is zero.  This is the only return
601         // other than the one at the end. All other exits from this function
602         // are fatal errors. After we check that the algorithm is supported
603         // anything else that goes wrong is an implementation flaw.
604         if((hLen = (INT16) _cpri__GetDigestSize(hashAlg)) == 0)
605             return 0;
606 
607         // If the size of the request is larger than the numbers will handle,
608         // it is a fatal error.
609         pAssert(((sizeInBits + 7)/ 8) <= INT16_MAX);
610 
611         bytes = once ? hLen : (INT16)((sizeInBits + 7) / 8);
612 
613         // Generate required bytes
614         for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
615         {
616             if(bytes < hLen)
617                 hLen = bytes;
618 
619             counter++;
620             // Start HMAC
621             if(_cpri__StartHMAC(hashAlg,
622                                 FALSE,
623                                 &hashState,
624                                 key->size,
625                                 &key->buffer[0],
626                                 &hmacKey.b)         <= 0)
627                 FAIL(FATAL_ERROR_INTERNAL);
628 
629             // Adding counter
630             UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
631             _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
632 
633             // Adding label
634             if(label != NULL)
635                 _cpri__UpdateHash(&hashState,  lLen, (BYTE *)label);
636 
637             // Adding contextU
638             if(contextU != NULL)
639                 _cpri__UpdateHash(&hashState, contextU->size, contextU->buffer);
640 
641             // Adding contextV
642             if(contextV != NULL)
643                 _cpri__UpdateHash(&hashState, contextV->size, contextV->buffer);
644 
645             // Adding size in bits
646             UINT32_TO_BYTE_ARRAY(sizeInBits, marshaledUint32);
647             _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
648 
649             // Compute HMAC. At the start of each iteration, hLen is set
650             // to the smaller of hLen and bytes. This causes bytes to decrement
651             // exactly to zero to complete the loop
652             _cpri__CompleteHMAC(&hashState, &hmacKey.b, hLen, stream);
653         }
654 
655         // Mask off bits if the required bits is not a multiple of byte size
656         if((sizeInBits % 8) != 0)
657             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
658         if(counterInOut != NULL)
659             *counterInOut = counter;
660         return (CRYPT_RESULT)((sizeInBits + 7)/8);
661     }
```

B.8.6.3. _cpri__KDFe()

KDFe() as defined in 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

```
662 UINT16
663 _cpri__KDFe(
664     TPM_ALG_ID       hashAlg,           // IN: hash algorithm used in HMAC
665     TPM2B           *Z,                 // IN: Z
666     const char      *label,             // IN: a 0 terminated label using in KDF
667     TPM2B           *partyUInfo,        // IN: PartyUInfo
668     TPM2B           *partyVInfo,        // IN: PartyVInfo
669     UINT32           sizeInBits,        // IN: size of generated key in bits
670     BYTE            *keyStream          // OUT: key buffer
671 )
672 {
673     UINT32       counter = 0;       // counter value
674     UINT32       lSize = 0;
675     BYTE        *stream = keyStream;
676     CPRI_HASH_STATE         hashState;
677     INT16        hLen = (INT16) _cpri__GetDigestSize(hashAlg);
678     INT16        bytes;             // number of bytes to generate
679     BYTE         marshaledUint32[4];
680
681     pAssert(   keyStream != NULL
682                && Z != NULL
683                && ((sizeInBits + 7) / 8) < INT16_MAX);
684
685     if(hLen == 0)
686         return 0;
687
688     bytes = (INT16)((sizeInBits + 7) / 8);
689
690     // Prepare label buffer.  Calculate its size and keep the last 0 byte
691     if(label != NULL)
692         for(lSize = 0; label[lSize++] != 0;);
693
694     // Generate required bytes
695     //The inner loop of that KDF uses:
696     //  Hashi := H(counter | Z | OtherInfo) (5)
697     // Where:
698     //  Hashi   the hash generated on the i-th iteration of the loop.
699     //  H()     an approved hash function
700     //  counter a 32-bit counter that is initialized to 1 and incremented
701     //          on each iteration
702     //  Z       the X coordinate of the product of a public ECC key and a
703     //          different private ECC key.
704     //  OtherInfo   a collection of qualifying data for the KDF defined below.
705     //  In this specification, OtherInfo will be constructed by:
706     //      OtherInfo := Use | PartyUInfo  | PartyVInfo
707     for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
708     {
709         if(bytes < hLen)
710             hLen = bytes;
```

```
711
712             counter++;
713             // Start hash
714             if(_cpri__StartHash(hashAlg, FALSE,  &hashState) == 0)
715                 return 0;
716
717             // Add counter
718             UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
719             _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);
720
721             // Add Z
722             if(Z != NULL)
723                 _cpri__UpdateHash(&hashState, Z->size, Z->buffer);
724
725             // Add label
726             if(label != NULL)
727                 _cpri__UpdateHash(&hashState, lSize, (BYTE *)label);
728             else
729
730                 // The SP800-108 specification requires a zero between the label
731                 // and the context.
732                 _cpri__UpdateHash(&hashState, 1, (BYTE *)"");
733
734             // Add PartyUInfo
735             if(partyUInfo != NULL)
736                 _cpri__UpdateHash(&hashState, partyUInfo->size, partyUInfo->buffer);
737
738             // Add PartyVInfo
739             if(partyVInfo != NULL)
740                 _cpri__UpdateHash(&hashState, partyVInfo->size, partyVInfo->buffer);
741
742             // Compute Hash. hLen was changed to be the smaller of bytes or hLen
743             // at the start of each iteration.
744             _cpri__CompleteHash(&hashState, hLen, stream);
745         }
746
747         // Mask off bits if the required bits is not a multiple of byte size
748         if((sizeInBits % 8) != 0)
749             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
750
751         return (CRYPT_RESULT)((sizeInBits + 7) / 8);
752
753     }
```

B.9 CpriSym.c

B.9.1. 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.9.2. Includes, Defines, and Typedefs

```
#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.

```
typedef AES_KEY SM4_KEY;
#define SM4_set_encrypt_key     AES_set_encrypt_key
#define SM4_set_decrypt_key     AES_set_decrypt_key
#define SM4_decrypt             AES_decrypt
#define SM4_encrypt             AES_encrypt
```

B.9.3. Utility Functions

B.9.3.1. _cpri_SymStartup()

```
BOOL
_cpri__SymStartup(
    void
)
{
    return TRUE;
}
```

B.9.3.2. _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

```
INT16
_cpri__GetSymmetricBlockSize(
    TPM_ALG_ID      symmetricAlg,   // IN: the symmetric algorithm
    UINT16          keySizeInBits   // IN: the key size
)
{
    switch (symmetricAlg)
    {
#ifdef TPM_ALG_AES
    case TPM_ALG_AES:
#endif
#ifdef TPM_ALG_SM4 // Both AES and SM4 use the same block size
    case TPM_ALG_SM4:
#endif
        if(keySizeInBits != 0)  // This is mostly to have a reference to
```

```
29              // keySizeInBits for the compiler
30              return  16;
31          else
32              return 0;
33          break;
34
35      default:
36          return 0;
37      }
38  }
```

B.9.4. AES Encryption

B.9.4.1. _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

```
39  CRYPT_RESULT
40  _cpri__AESEncryptCBC(
41      BYTE        *dOut,          // OUT:
42      UINT32       keySizeInBits, // IN: key size in bits
43      BYTE        *key,           // IN: key buffer. The size of this buffer
44                                  //     in bytes is (keySizeInBits + 7) / 8
45      BYTE        *iv,            // IN/OUT: IV for decryption.
46      UINT32       dInSize,       // IN: data size (is required to be a multiple
47                                  //     of 16 bytes
48      BYTE        *dIn            // IN/OUT: data buffer
49  )
50  {
51      AES_KEY      AesKey;
52      BYTE        *pIv;
53      INT32        dSize;         // Need a signed version
54      int          i;
55
56      pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
57
58      if(dInSize == 0)
59          return CRYPT_SUCCESS;
60
61      pAssert(dInSize <= INT32_MAX);
62      dSize = (INT32)dInSize;
63
64      // For CBC, the data size must be an even multiple of the
65      // cipher block size
66      if((dSize % 16) != 0)
67          return CRYPT_PARAMETER;
68
69      // Create AES encrypt key schedule
70      if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
71          FAIL(FATAL_ERROR_INTERNAL);
72
73      // XOR the data block into the IV, encrypt the IV into the IV
74      // and then copy the IV to the output
75      for(; dSize > 0; dSize -= 16)
76      {
```

```
77          pIv = iv;
78          for(i = 16; i > 0; i--)
79              *pIv++ ^= *dIn++;
80          AES_encrypt(iv, iv, &AesKey);
81          pIv = iv;
82          for(i = 16; i > 0; i--)
83              *dOut++ = *pIv++;
84      }
85      return CRYPT_SUCCESS;
86  }
```

B.9.4.2. _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  CRYPT_RESULT
88  _cpri__AESDecryptCBC(
89      BYTE        *dOut,              // OUT: the decrypted data
90      UINT32       keySizeInBits,     // IN: key size in bits
91      BYTE        *key,               // IN: key buffer. The size of this buffer
92                                      //     in bytes is (keySizeInBits + 7) / 8
93      BYTE        *iv,                // IN/OUT: IV for decryption. The size of
94                                      // this buffer if 16 byte.
95      UINT32       dInSize,           // IN: data size
96      BYTE        *dIn                // IN: data buffer
97  )
98  {
99      AES_KEY      AesKey;
100     BYTE        *pIv;
101     int          i;
102     BYTE         tmp[16];
103     BYTE        *pT = NULL;
104     INT32        dSize;
105
106     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
107
108     if(dInSize == 0)
109         return CRYPT_SUCCESS;
110
111     pAssert(dInSize <= INT32_MAX);
112     dSize = (INT32)dInSize;
113
114     // For CBC, the data size must be an even multiple of the
115     // cipher block size
116     if((dSize % 16) != 0)
117         return CRYPT_PARAMETER;
118
119     // Create AES key schedule
120     if (AES_set_decrypt_key(key, keySizeInBits, &AesKey) != 0)
121         FAIL(FATAL_ERROR_INTERNAL);
122
123     // Copy the input data to a temp buffer, decrypt the buffer into the output;
124     // XOR in the IV, and copy the temp buffer to the IV and repeat.
125     for(; dSize > 0; dSize -= 16)
126     {
```

 Copyright © TCG 2006-2013

```
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.9.4.3. _cpri__AESEncryptCFB()

This function performs AES encryption in CFB chain mode. The *dOut* buffer receives the values encrypted *dIn*. The input *iv* is assumed to be the size of an encryption block (16 bytes). The *iv* buffer will be modified to contain the last encrypted block.

Return Value	Meaning
CRYPT_SUCCESS	no non-fatal errors

```
141     CRYPT_RESULT
142     _cpri__AESEncryptCFB(
143         BYTE        *dOut,          // OUT: the encrypted
144         UINT32       keySizeInBits, // IN: key size in bit
145         BYTE        *key,           // IN: key buffer. The size of this buffer
146                                     //     in bytes is (keySizeInBits + 7) / 8
147         BYTE        *iv,            // IN/OUT: IV for decryption.
148         UINT32       dInSize,       // IN: data size
149         BYTE        *dIn            // IN/OUT: data buffer
150     )
151     {
152         BYTE        *pIv = 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.9.4.4. _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     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
192                                     //     in bytes is (keySizeInBits + 7) / 8
193         BYTE        *iv,            // IN/OUT: IV for decryption.
194         UINT32       dInSize,       // IN: data size
195         BYTE        *dIn            // IN/OUT: 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.9.4.5. _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     CRYPT_RESULT
238     _cpri__AESEncryptCTR(
239         BYTE        *dOut,          // OUT: the encrypted data
240         UINT32       keySizeInBits, // IN: key size in bits
241         BYTE        *key,           // IN: key buffer. The size of this buffer
242                                     //     in 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.9.4.6. _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 //%
```

The //% is used by the prototype extraction program to cause it to include the line in the prototype file after removing the //%. Need an extra line with nothing on it so that a blank line will separate this macro from the next definition.

B.9.4.7. _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

```
293 CRYPT_RESULT
294 _cpri__AESEncryptECB(
295     BYTE        *dOut,          // OUT: encrypted data
296     UINT32       keySizeInBits, // IN: key size in bit
297     BYTE        *key,           // IN: key buffer. The size of this buffer
298                                 //     in bytes is (keySizeInBits + 7) / 8
299     UINT32       dInSize,       // IN: data size
300     BYTE        *dIn            // IN: clear text buffer
301 )
302 {
303     AES_KEY      AesKey;
304     INT32        dSize;
305
306     pAssert(dOut != NULL && key != NULL && dIn != NULL);
307
308     if(dInSize == 0)
309         return CRYPT_SUCCESS;
310
311     pAssert(dInSize <= INT32_MAX);
312     dSize = (INT32)dInSize;
313
314     // For ECB, the data size must be an even multiple of the
315     // cipher block size
316     if((dSize % 16) != 0)
317         return CRYPT_PARAMETER;
318     // Create AES encrypting key schedule
319     if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
320         FAIL(FATAL_ERROR_INTERNAL);
321
322     for(; dSize > 0; dSize -= 16)
323     {
324         AES_encrypt(dIn, dOut, &AesKey);
325         dIn = &dIn[16];
326         dOut = &dOut[16];
327     }
```

```
328         return CRYPT_SUCCESS;
329     }
```

B.9.4.8. _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

```
330     CRYPT_RESULT
331     _cpri__AESDecryptECB(
332         BYTE        *dOut,              // OUT: the clear text data
333         UINT32       keySizeInBits,     // IN: key size in bit
334         BYTE        *key,               // IN: key buffer. The size of this buffer
335                                         //     in bytes is (keySizeInBits + 7) / 8
336         UINT32       dInSize,           // IN: data size
337         BYTE        *dIn                // IN: cipher text buffer
338     )
339     {
340         AES_KEY      AesKey;
341         INT32        dSize;
342
343         pAssert(dOut != NULL && key != NULL && dIn != NULL);
344
345         if(dInSize == 0)
346             return CRYPT_SUCCESS;
347
348         pAssert(dInSize <= INT32_MAX);
349         dSize = (INT32)dInSize;
350
351         // For ECB, the data size must be an even multiple of the
352         // cipher block size
353         if((dSize % 16) != 0)
354             return CRYPT_PARAMETER;
355
356         // Create AES decryption key schedule
357         if (AES_set_decrypt_key(key, keySizeInBits, &AesKey) != 0)
358             FAIL(FATAL_ERROR_INTERNAL);
359
360         for(; dSize > 0; dSize -= 16)
361         {
362             AES_decrypt(dIn, dOut, &AesKey);
363             dIn = &dIn[16];
364             dOut = &dOut[16];
365         }
366         return CRYPT_SUCCESS;
367     }
```

B.9.4.9. _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

```
368 CRYPT_RESULT
369 _cpri__AESEncryptOFB(
370     BYTE        *dOut,          // OUT: the encrypted/decrypted data
371     UINT32       keySizeInBits, // IN: key size in bit
372     BYTE        *key,           // IN: key buffer. The size of this buffer
373                                 //     in bytes is (keySizeInBits + 7) / 8
374     BYTE        *iv,            // IN/OUT: IV for decryption. The size of
375                                 //     this buffer if 16 byte.
376     UINT32       dInSize,       // IN: data size
377     BYTE        *dIn            // IN: data buffer
378 )
379 {
380     BYTE        *pIv;
381     AES_KEY      AesKey;
382     INT32        dSize;
383     int          i;
384
385     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
386
387     if(dInSize == 0)
388         return CRYPT_SUCCESS;
389
390     pAssert(dInSize <= INT32_MAX);
391     dSize = (INT32)dInSize;
392
393     // Create AES key schedule
394     if (AES_set_encrypt_key(key, keySizeInBits, &AesKey) != 0)
395         FAIL(FATAL_ERROR_INTERNAL);
396
397     // This is written so that dIn and dOut may be the same
398
399     for(; dSize > 0; dSize -= 16)
400     {
401         // Encrypt the current value of the "IV"
402         AES_encrypt(iv, iv, &AesKey);
403
404         // XOR the encrypted IV into dIn to create the cipher text (dOut)
405         pIv = iv;
406         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
407             *dOut++ = (*pIv++ ^ *dIn++);
408     }
409     return CRYPT_SUCCESS;
410 }
```

B.9.4.10. _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)

```
411 //%#define _cpri__AESDecryptOFB(dOut,keySizeInBits, key, iv, dInSize, dIn) \
412 //%        _cpri__AESEncryptOFB (                                  \
413 //%                              ((BYTE *)dOut),                   \
414 //%                              ((UINT32)keySizeInBits),          \
415 //%                              ((BYTE *)key),                    \
416 //%                              ((BYTE *)iv),                     \
417 //%                              ((UINT32)dInSize),                \
418 //%                              ((BYTE *)dIn)                     \
419 //%                             )
420 //%
```

```
421 #ifdef  TPM_ALG_SM4     //%
```

B.9.5. SM4 Encryption

B.9.5.1. _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

```
422 CRYPT_RESULT
423 _cpri__SM4EncryptCBC(
424     BYTE        *dOut,          // OUT:
425     UINT32       keySizeInBits, // IN: key size in bits
426     BYTE        *key,           // IN: key buffer. The size of this buffer
427                                 //     in bytes is (keySizeInBits + 7) / 8
428     BYTE        *iv,            // IN/OUT: IV for decryption.
429     UINT32       dInSize,       // IN: data size (is required to be a multiple
430                                 //     of 16 bytes
431     BYTE        *dIn            // IN/OUT: data buffer
432 )
433 {
434     SM4_KEY      Sm4Key;
435     BYTE        *pIv;
436     INT32        dSize;         // Need a signed version
437     int          i;
438
439     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
440
441     if(dInSize == 0)
442         return CRYPT_SUCCESS;
443
444     pAssert(dInSize <= INT32_MAX);
445     dSize = (INT32)dInSize;
446
447     // For CBC, the data size must be an even multiple of the
448     // cipher block size
449     if((dSize % 16) != 0)
450         return CRYPT_PARAMETER;
451
452     // Create SM4 encrypt key schedule
453     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
454         FAIL(FATAL_ERROR_INTERNAL);
455
456     // XOR the data block into the IV, encrypt the IV into the IV
457     // and then copy the IV to the output
458     for(; dSize > 0; dSize -= 16)
459     {
460         pIv = iv;
461         for(i = 16; i > 0; i--)
462             *pIv++ ^= *dIn++;
463         SM4_encrypt(iv, iv, &Sm4Key);
464         pIv = iv;
465         for(i = 16; i > 0; i--)
466             *dOut++ = *pIv++;
467     }
468     return CRYPT_SUCCESS;
```

```
469 }
```

B.9.5.2. _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

```
470 CRYPT_RESULT
471 _cpri__SM4DecryptCBC(
472     BYTE        *dOut,              // OUT: the decrypted data
473     UINT32       keySizeInBits,     // IN: key size in bits
474     BYTE        *key,               // IN: key buffer. The size of this buffer
475                                     //     in bytes is (keySizeInBits + 7) / 8
476     BYTE        *iv,                // IN/OUT: IV for decryption. The size of
477                                     // this buffer if 16 byte.
478     UINT32       dInSize,           // IN: data size
479     BYTE        *dIn                // IN: data buffer
480 )
481 {
482     SM4_KEY      Sm4Key;
483     BYTE        *pIv;
484     int          i;
485     BYTE         tmp[16];
486     BYTE        *pT = NULL;
487     INT32        dSize;
488
489     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
490
491     if(dInSize == 0)
492         return CRYPT_SUCCESS;
493
494     pAssert(dInSize <= INT32_MAX);
495     dSize = (INT32)dInSize;
496
497     // For CBC, the data size must be an even multiple of the
498     // cipher block size
499     if((dSize % 16) != 0)
500         return CRYPT_PARAMETER;
501
502     // Create SM4 key schedule
503     if (SM4_set_decrypt_key(key, keySizeInBits, &Sm4Key) != 0)
504         FAIL(FATAL_ERROR_INTERNAL);
505
506     // Copy the input data to a temp buffer, decrypt the buffer into the output;
507     // XOR in the IV, and copy the temp buffer to the IV and repeat.
508     for(; dSize > 0; dSize -= 16)
509     {
510         pT = tmp;
511         for(i = 16; i> 0; i--)
512             *pT++ = *dIn++;
513         SM4_decrypt(tmp, dOut, &Sm4Key);
514         pIv = iv;
515         pT = tmp;
516         for(i = 16; i> 0; i--)
517         {
518             *dOut++ ^= *pIv;
```

```
519                 *pIv++ = *pT++;
520             }
521         }
522         return CRYPT_SUCCESS;
523     }
```

B.9.5.3. _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

```
524     CRYPT_RESULT
525     _cpri__SM4EncryptCFB(
526         BYTE        *dOut,          // OUT: the encrypted
527         UINT32       keySizeInBits, // IN: key size in bit
528         BYTE        *key,           // IN: key buffer. The size of this buffer
529                                     //     in bytes is (keySizeInBits + 7) / 8
530         BYTE        *iv,            // IN/OUT: IV for decryption.
531         UINT32       dInSize,       // IN: data size
532         BYTE        *dIn            // IN/OUT: data buffer
533     )
534     {
535         BYTE        *pIv;
536         SM4_KEY      Sm4Key;
537         INT32        dSize;         // Need a signed version of dInSize
538         int          i;
539
540         pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
541
542         if(dInSize == 0)
543             return CRYPT_SUCCESS;
544
545         pAssert(dInSize <= INT32_MAX);
546         dSize = (INT32)dInSize;
547
548         // Create SM4 encryption key schedule
549         if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
550             FAIL(FATAL_ERROR_INTERNAL);
551
552         // Encrypt the IV into the IV, XOR in the data, and copy to output
553         for(; dSize > 0; dSize -= 16)
554         {
555             // Encrypt the current value of the IV
556             SM4_encrypt(iv, iv, &Sm4Key);
557             pIv = iv;
558             for(i = (int)(dSize < 16) ? dSize : 16; i > 0; i--)
559                 // XOR the data into the IV to create the cipher text
560                 // and put into the output
561                 *dOut++ = *pIv++ ^= *dIn++;
562         }
563         return CRYPT_SUCCESS;
564     }
```

B.9.5.4. _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

```
565 CRYPT_RESULT
566 _cpri__SM4DecryptCFB(
567     BYTE        *dOut,          // OUT: the decrypted data
568     UINT32       keySizeInBits, // IN: key size in bit
569     BYTE        *key,           // IN: key buffer. The size of this buffer
570                                 //     in bytes is (keySizeInBits + 7) / 8
571     BYTE        *iv,            // IN/OUT: IV for decryption.
572     UINT32       dInSize,       // IN: data size
573     BYTE        *dIn            // IN/OUT: data buffer
574 )
575 {
576     BYTE        *pIv;
577     BYTE         tmp[16];
578     int          i;
579     BYTE        *pT;
580     SM4_KEY      Sm4Key;
581     INT32        dSize;
582
583     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
584
585     if(dInSize == 0)
586         return CRYPT_SUCCESS;
587
588     pAssert(dInSize <= INT32_MAX);
589     dSize = (INT32)dInSize;
590
591     // Create SM4 encryption key schedule
592     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
593         FAIL(FATAL_ERROR_INTERNAL);
594
595     for(; dSize > 0; dSize -= 16)
596     {
597         // Encrypt the IV into the temp buffer
598         SM4_encrypt(iv, tmp, &Sm4Key);
599         pT = tmp;
600         pIv = iv;
601         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
602             // Copy the current cipher text to IV, XOR
603             // with the temp buffer and put into the output
604             *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
605     }
606     // If the inner loop (i loop) was smaller than 16, then dSize
607     // would have been smaller than 16 and it is now negative
608     // If it is negative, then it indicates how may fill bytes
609     // are needed to pad out the IV for the next round.
610     for(; dSize < 0; dSize++)
611         *iv++ = 0;
612
613     return CRYPT_SUCCESS;
614 }
```

B.9.5.5. _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

```
615 CRYPT_RESULT
616 _cpri__SM4EncryptCTR(
617     BYTE        *dOut,          // OUT: the encrypted data
618     UINT32       keySizeInBits, // IN: key size in bits
619     BYTE        *key,           // IN: key buffer. The size of this buffer
620                                 //     in bytes is (keySizeInBits + 7) / 8
621     BYTE        *iv,            // IN/OUT: IV for decryption.
622     UINT32       dInSize,       // IN: data size
623     BYTE        *dIn            // IN: data buffer
624 )
625 {
626     BYTE         tmp[16];
627     BYTE        *pT;
628     SM4_KEY      Sm4Key;
629     int          i;
630     INT32        dSize;
631
632     pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
633
634     if(dInSize == 0)
635         return CRYPT_SUCCESS;
636
637     pAssert(dInSize <= INT32_MAX);
638     dSize = (INT32)dInSize;
639
640     // Create SM4 encryption schedule
641     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
642         FAIL(FATAL_ERROR_INTERNAL);
643
644     for(; dSize > 0; dSize--)
645     {
646         // Encrypt the current value of the IV(counter)
647         SM4_encrypt(iv, (BYTE *)tmp, &Sm4Key);
648
649         //increment the counter
650         for(i = 0; i < 16; i++)
651             if((iv[i] += 1) != 0)
652                 break;
653
654         // XOR the encrypted counter value with input and put into output
655         pT = tmp;
656         for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
657             *dOut++ = *dIn++ ^ *pT++;
658     }
659     return CRYPT_SUCCESS;
660 }
```

B.9.5.6. _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)

```
661 //% #define _cpri__SM4DecryptCTR(dOut, keySize, key, iv, dInSize, dIn) \
662 //%         _cpri__SM4EncryptCTR(                                     \
663 //%                              ((BYTE *)dOut),                      \
664 //%                              ((UINT32)keySize),                   \
665 //%                              ((BYTE *)key),                       \
666 //%                              ((BYTE *)iv),                        \
667 //%                              ((UINT32)dInSize),                   \
```

```
668 //%                              ((BYTE *)dIn)              \
669 //%                            )
670 //%
```

The //% is used by the prototype extraction program to cause it to include the line in the prototype file after removing the //%. Need an extra line with nothing on it so that a blank line will separate this macro from the next definition.

B.9.5.7. _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

```
671 CRYPT_RESULT
672 _cpri__SM4EncryptECB(
673     BYTE        *dOut,           // OUT: encrypted data
674     UINT32       keySizeInBits, // IN: key size in bit
675     BYTE        *key,            // IN: key buffer. The size of this buffer
676                                  //     in bytes is (keySizeInBits + 7) / 8
677     UINT32       dInSize,        // IN: data size
678     BYTE        *dIn             // IN: clear text buffer
679 )
680 {
681     SM4_KEY      Sm4Key;
682     INT32        dSize;
683
684     pAssert(dOut != NULL && key != NULL && dIn != NULL);
685
686     if(dInSize == 0)
687         return CRYPT_SUCCESS;
688
689     pAssert(dInSize <= INT32_MAX);
690     dSize = (INT32)dInSize;
691
692     // For ECB, the data size must be an even multiple of the
693     // cipher block size
694     if((dSize % 16) != 0)
695         return CRYPT_PARAMETER;
696     // Create SM4 encrypting key schedule
697     if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
698         FAIL(FATAL_ERROR_INTERNAL);
699
700     for(; dSize > 0; dSize -= 16)
701     {
702         SM4_encrypt(dIn, dOut, &Sm4Key);
703         dIn = &dIn[16];
704         dOut = &dOut[16];
705     }
706     return CRYPT_SUCCESS;
707 }
```

B.9.5.8. _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

```
708 CRYPT_RESULT
709 _cpri__SM4DecryptECB(
710     BYTE        *dOut,            // OUT: the clear text data
711     UINT32       keySizeInBits,   // IN: key size in bit
712     BYTE        *key,             // IN: key buffer. The size of this buffer
713                                   //     in bytes is (keySizeInBits + 7) / 8
714     UINT32       dInSize,         // IN: data size
715     BYTE        *dIn              // IN: cipher text buffer
716 )
717 {
718     SM4_KEY      Sm4Key;
719     INT32        dSize;
720
721     pAssert(dOut != NULL && key != NULL && dIn != NULL);
722
723     if(dInSize == 0)
724         return CRYPT_SUCCESS;
725
726     pAssert(dInSize <= INT32_MAX);
727     dSize = (INT32)dInSize;
728
729     // For ECB, the data size must be an even multiple of the
730     // cipher block size
731     if((dSize % 16) != 0)
732         return CRYPT_PARAMETER;
733
734     // Create SM4 decryption key schedule
735     if (SM4_set_decrypt_key(key, keySizeInBits, &Sm4Key) != 0)
736         FAIL(FATAL_ERROR_INTERNAL);
737
738     for(; dSize > 0; dSize -= 16)
739     {
740         SM4_decrypt(dIn, dOut, &Sm4Key);
741         dIn = &dIn[16];
742         dOut = &dOut[16];
743     }
744     return CRYPT_SUCCESS;
745 }
```

B.9.5.9. _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

```
746 CRYPT_RESULT
747 _cpri__SM4EncryptOFB(
748     BYTE        *dOut,            // OUT: the encrypted/decrypted data
749     UINT32       keySizeInBits,   // IN: key size in bit
750     BYTE        *key,             // IN: key buffer. The size of this buffer
751                                   //     in bytes is (keySizeInBits + 7) / 8
752     BYTE        *iv,              // IN/OUT: IV for decryption. The size of
753                                   //     this buffer if 16 byte.
```

```
754         UINT32       dInSize,       // IN: data size
755         BYTE        *dIn            // IN: data buffer
756     )
757     {
758         BYTE        *pIv;
759         SM4_KEY      Sm4Key;
760         INT32        dSize;
761         int          i;
762
763         pAssert(dOut != NULL && key != NULL && iv != NULL && dIn != NULL);
764
765         if(dInSize == 0)
766             return CRYPT_SUCCESS;
767
768         pAssert(dInSize <= INT32_MAX);
769         dSize = (INT32)dInSize;
770
771         // Create SM4 key schedule
772         if (SM4_set_encrypt_key(key, keySizeInBits, &Sm4Key) != 0)
773             FAIL(FATAL_ERROR_INTERNAL);
774
775         // This is written so that dIn and dOut may be the same
776
777         for(; dSize > 0; dSize -= 16)
778         {
779             // Encrypt the current value of the "IV"
780             SM4_encrypt(iv, iv, &Sm4Key);
781
782             // XOR the encrypted IV into dIn to create the cipher text (dOut)
783             pIv = iv;
784             for(i = (dSize < 16) ? dSize : 16; i > 0; i--)
785                 *dOut++ = (*pIv++ ^ *dIn++);
786         }
787         return CRYPT_SUCCESS;
788     }
```

B.9.5.10. _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)

```
789     //%#define _cpri__SM4DecryptOFB(dOut,keySizeInBits, key, iv, dInSize, dIn) \
790     //%        _cpri__SM4EncryptOFB (                                          \
791     //%                              ((BYTE *)dOut),                           \
792     //%                              ((UINT32)keySizeInBits),                  \
793     //%                              ((BYTE *)key),                            \
794     //%                              ((BYTE *)iv),                             \
795     //%                              ((UINT32)dInSize),                        \
796     //%                              ((BYTE *)dIn)                             \
797     //%                             )
798     //%
799     #endif      //% TPM_ALG_SM4
```

B.10 RSA Files

B.10.1. CpriRSA.c

B.10.1.1. 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.10.1.2. Includes

```
#include "OsslCryptoEngine.h"
```

B.10.1.3. Local Functions

B.10.1.3.1. 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 denormalized.

Return Value	Meaning
CRYPT_SUCCESS	private exponent computed
CRYPT_PARAMETER	prime is not half the size of the modulus, or the modulus is not evenly divisible by the prime, or no private exponent could be computed from the input parameters

```
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();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
```

```
24          BN_CTX_start(context);
25          bnE = BN_CTX_get(context);
26          bnD = BN_CTX_get(context);
27          bnN = BN_CTX_get(context);
28          bnP = BN_CTX_get(context);
29          bnPhi = BN_CTX_get(context);
30          bnQ = BN_CTX_get(context);
31          bnQr = BN_CTX_get(context);
32
33          if(bnQr == NULL)
34              FAIL(FATAL_ERROR_ALLOCATION);
35
36          // Assume the size of the public key value is within range
37          pAssert(key->publicKey->size <= MAX_RSA_KEY_BYTES);
38
39          if(   BN_bin2bn(key->publicKey->buffer, key->publicKey->size, bnN) == NULL
40             || BN_bin2bn(key->privateKey->buffer, key->privateKey->size, bnP) == NULL)
41
42              FAIL(FATAL_ERROR_INTERNAL);
43
44          // If P size is not 1/2 of n size, then this is not a valid value for this
45          // implementation. This will also catch the case were P is input as zero.
46          // This generates a return rather than an assert because the key being loaded
47          // might be SW generated and wrong.
48          if(BN_num_bits(bnP) < BN_num_bits(bnN)/2)
49          {
50              retVal = CRYPT_PARAMETER;
51              goto Cleanup;
52          }
53          // Get q = n/p;
54          if (BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
55              FAIL(FATAL_ERROR_INTERNAL);
56
57          // If there is a remainder, then this is not a valid n
58          if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
59          {
60              retVal = CRYPT_PARAMETER;       // problem may be recoverable
61              goto Cleanup;
62          }
63          // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
64          if(   BN_copy(bnPhi, bnN) == NULL
65             || !BN_sub(bnPhi, bnPhi, bnP)
66             || !BN_sub(bnPhi, bnPhi, bnQ)
67             || !BN_add_word(bnPhi, 1))
68              FAIL(FATAL_ERROR_INTERNAL);
69
70          // Compute the multiplicative inverse
71          BN_set_word(bnE, key->exponent);
72          if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
73          {
74              // Going to assume that the error is caused by a bad
75              // set of parameters. Specifically, an exponent that is
76              // not compatible with the primes. In an implementation that
77              // has better visibility to the error codes, this might be
78              // refined so that failures in the library would return
79              // a more informative value.  Should not assume here that
80              // the error codes will remain unchanged.
81
82              retVal = CRYPT_PARAMETER;
83              goto Cleanup;
84          }
85
86          fill = key->publicKey->size - BN_num_bytes(bnD);
87          BN_bn2bin(bnD, &key->privateKey->buffer[fill]);
88          memset(key->privateKey->buffer, 0, fill);
89
```

DRAFT

```
90      // Change the size of the private key so that it is known to contain
91      // a private exponent rather than a prime.
92      key->privateKey->size = key->publicKey->size;
93
94  Cleanup:
95      BN_CTX_end(context);
96      BN_CTX_free(context);
97      return retVal;
98  }
```

B.10.1.3.2. _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

```
99  CRYPT_RESULT
100 _cpri__TestKeyRSA(
101     TPM2B           *d,         // OUT: the address to receive the private exponent
102     UINT32           exponent,  // IN: the public modulus
103     TPM2B           *publicKey, // IN/OUT: an input if only one prime is provided.
104                                 //         an output if both primes are provided
105     TPM2B           *prime1,    // IN: a first prime
106     TPM2B           *prime2     // IN: an optional second prime
107 )
108 {
109     BN_CTX          *context;
110     BIGNUM          *bnD;
111     BIGNUM          *bnN;
112     BIGNUM          *bnP;
113     BIGNUM          *bnE;
114     BIGNUM          *bnPhi;
115     BIGNUM          *bnQ;
116     BIGNUM          *bnQr;
117     UINT32          fill;
118
119     CRYPT_RESULT    retVal = CRYPT_SUCCESS;     // Assume success
120
121     pAssert(publicKey != NULL && prime1 != NULL);
122     // Make sure that the sizes are within range
123     pAssert(   prime1->size <= MAX_RSA_KEY_BYTES/2
124             && publicKey->size <= MAX_RSA_KEY_BYTES);
125     pAssert( prime2 == NULL || prime2->size < MAX_RSA_KEY_BYTES/2);
126
127     if(publicKey->size/2 != prime1->size)
128         return CRYPT_PARAMETER;
129
130     context = BN_CTX_new();
131     if(context == NULL)
132         FAIL(FATAL_ERROR_ALLOCATION);
133     BN_CTX_start(context);
134     bnE = BN_CTX_get(context);      // public exponent (e)
135     bnD = BN_CTX_get(context);      // private exponent (d)
136     bnN = BN_CTX_get(context);      // public modulus (n)
137     bnP = BN_CTX_get(context);      // prime1 (p)
```

```
138         bnPhi = BN_CTX_get(context);     // (p-1)(q-1)
139         bnQ = BN_CTX_get(context);       // prime2 (q)
140         bnQr = BN_CTX_get(context);      // n mod p
141
142         if(bnQr == NULL)
143             FAIL(FATAL_ERROR_ALLOCATION);
144
145         if(BN_bin2bn(prime1->buffer, prime1->size, bnP) == NULL)
146             FAIL(FATAL_ERROR_INTERNAL);
147
148         // If prime2 is provided, then compute n
149         if(prime2 != NULL)
150         {
151             // Two primes provided so use them to compute n
152             if(BN_bin2bn(prime2->buffer, prime2->size, bnQ) == NULL)
153                 FAIL(FATAL_ERROR_INTERNAL);
154
155             // Make sure that the sizes of the primes are compatible
156             if(BN_num_bits(bnQ) != BN_num_bits(bnP))
157             {
158                 retVal = CRYPT_PARAMETER;
159                 goto Cleanup;
160             }
161             // Multiply the primes to get the public modulus
162
163             if(BN_mul(bnN, bnP, bnQ, context) != 1)
164                 FAIL(FATAL_ERROR_INTERNAL);
165
166             // if the space provided for the public modulus is large enough,
167             // save the created value
168             if(BN_num_bits(bnN) != (publicKey->size * 8))
169             {
170                 retVal = CRYPT_PARAMETER;
171                 goto Cleanup;
172             }
173             BN_bn2bin(bnN, publicKey->buffer);
174         }
175         else
176         {
177             // One prime provided so find the second prime by division
178             BN_bin2bn(publicKey->buffer, publicKey->size, bnN);
179
180             // Get q = n/p;
181             if(BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
182                 FAIL(FATAL_ERROR_INTERNAL);
183
184             // If there is a remainder, then this is not a valid n
185             if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
186             {
187                 retVal = CRYPT_PARAMETER;       // problem may be recoverable
188                 goto Cleanup;
189             }
190         }
191         // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
192         BN_copy(bnPhi, bnN);
193         BN_sub(bnPhi, bnPhi, bnP);
194         BN_sub(bnPhi, bnPhi, bnQ);
195         BN_add_word(bnPhi, 1);
196         // Compute the multiplicative inverse
197         BN_set_word(bnE, exponent);
198         if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
199         {
200             // Going to assume that the error is caused by a bad set of parameters.
201             // Specifically, an exponent that is not compatible with the primes.
202             // In an implementation that has better visibility to the error codes,
203             // this might be refined so that failures in the library would return
```

```
204             // a more informative value.
205             // Do not assume that the error codes will remain unchanged.
206             retVal = CRYPT_PARAMETER;
207             goto Cleanup;
208         }
209         // Return the private exponent.
210         // Make sure it is normalized to have the correct size.
211         d->size = publicKey->size;
212         fill = d->size - BN_num_bytes(bnD);
213         BN_bn2bin(bnD, &d->buffer[fill]);
214         memset(d->buffer, 0, fill);
215     Cleanup:
216         BN_CTX_end(context);
217         BN_CTX_free(context);
218         return retVal;
219     }
```

B.10.1.3.3. RSAEP()

This function performs the RSAEP operation defined in PKCS#1v2. 1. It is an exponentiation of a value *(m*) with the public exponent *(e*), modulo the public *(n*).

Return Value	Meaning
CRYPT_SUCCESS	encryption complete
CRYPT_PARAMETER	number to exponentiate is larger than the modulus

```
220     static CRYPT_RESULT
221     RSAEP (
222         UINT32       dInOutSize,  // OUT size of the encrypted block
223         BYTE        *dInOut,      // OUT: the encrypted data
224         RSA_KEY     *key          // IN: the key to use
225     )
226     {
227         UINT32       e;
228         BYTE         exponent[4];
229         CRYPT_RESULT retVal;
230
231         e = key->exponent;
232         if(e == 0)
233             e = RSA_DEFAULT_PUBLIC_EXPONENT;
234         UINT32_TO_BYTE_ARRAY(e, exponent);
235
236         //!!! Can put check for test of RSA here
237
238         retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut, 4, exponent,
239                                key->publicKey->size, key->publicKey->buffer);
240
241         // Exponentiation result is stored in-place, thus no space shortage is possible.
242         pAssert(retVal != CRYPT_UNDERFLOW);
243
244         return retVal;
245     }
```

B.10.1.3.4. RSADP()

This function performs the RSADP operation defined in PKCS#1v2. 1. It is an exponentiation of a value *(c*) with the private exponent *(d*), modulo the public modulus *(n*). The decryption is in place.

This function also checks the size of the private key. If the size indicates that only a prime value is present, the key is converted to being a private exponent.

Return Value	Meaning
CRYPT_SUCCESS	decryption succeeded
CRYPT_PARAMETER	the value to decrypt is larger than the modulus

```
246  static CRYPT_RESULT
247  RSADP (
248      UINT32           dInOutSize,     // IN/OUT: size of decrypted data
249      BYTE            *dInOut,         // IN/OUT: the decrypted data
250      RSA_KEY         *key             // IN: the key
251  )
252  {
253      CRYPT_RESULT retVal;
254
255      //!!! Can put check for RSA tested here
256
257      // Make sure that the pointers are provided and that the private key is present
258      // If the private key is present it is assumed to have been created by
259      // so is presumed good _cpri__PrivateExponent
260      pAssert(key != NULL && dInOut != NULL &&
261              key->publicKey->size == key->publicKey->size);
262
263      // make sure that the value to be decrypted is smaller than the modulus
264      // note: this check is redundant as is also performed by _math__ModExp()
265      // which is optimized for use in RSA operations
266      if(_math__uComp(key->publicKey->size, key->publicKey->buffer,
267                      dInOutSize, dInOut) <= 0)
268          return CRYPT_PARAMETER;
269
270      // _math__ModExp can return CRYPT_PARAMTER or CRYPT_UNDERFLOW but actual
271      // underflow is not possible because everything is in the same buffer.
272      retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut,
273                             key->privateKey->size, key->privateKey->buffer,
274                             key->publicKey->size, key->publicKey->buffer);
275
276      // Exponentiation result is stored in-place, thus no space shortage is possible.
277      pAssert(retVal != CRYPT_UNDERFLOW);
278
279      return retVal;
280  }
```

B.10.1.3.5. 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

```
281  static CRYPT_RESULT
282  OaepEncode(
283      UINT32       paddedSize,     // IN: pad value size
284      BYTE        *padded,         // OUT: the pad data
285      TPM_ALG_ID   hashAlg,        // IN: algorithm to use for padding
286      const char  *label,          // IN: null-terminated string (may be NULL)
287      UINT32       messageSize,    // IN: the message size
288      BYTE        *message         // IN: the message being padded
289  #ifdef  TEST_RSA                 //
```

```
290         , BYTE        *testSeed    // IN: optional seed used for testing.
291     #endif  // TEST_RSA             //
292     )
293     {
294         UINT32       padLen;
295         UINT32       dbSize;
296         UINT32       i;
297         BYTE         mySeed[MAX_DIGEST_SIZE];
298         BYTE        *seed = mySeed;
299         INT32        hLen = _cpri__GetDigestSize(hashAlg);
300         BYTE         mask[MAX_RSA_KEY_BYTES];
301         BYTE        *pp;
302         BYTE        *pm;
303         UINT32       lSize = 0;
304         CRYPT_RESULT retVal = CRYPT_SUCCESS;
305
306
307         pAssert(padded != NULL && message != NULL);
308
309         // A value of zero is not allowed because the KDF can't produce a result
310         // if the digest size is zero.
311         if(hLen <= 0)
312             return CRYPT_PARAMETER;
313
314         // If a label is provided, get the length of the string, including the
315         // terminator
316         if(label != NULL)
317             lSize = (UINT32)strlen(label) + 1;
318
319         // Basic size check
320         // messageSize <= k  2hLen  2
321         if(messageSize > paddedSize - 2 * hLen - 2)
322             return CRYPT_FAIL;
323
324         // Hash L even if it is null
325         // Offset into padded leaving room for masked seed and byte of zero
326         pp = &padded[hLen + 1];
327         retVal = _cpri__HashBlock(hashAlg, lSize, (BYTE *)label, hLen, pp);
328
329         // concatenate PS of k  mLen  2hLen  2
330         padLen = paddedSize - messageSize - (2 * hLen) - 2;
331         memset(&pp[hLen], 0, padLen);
332         pp[hLen+padLen] = 0x01;
333         padLen += 1;
334         memcpy(&pp[hLen+padLen], message, messageSize);
335
336         // The total size of db = hLen + pad + mSize;
337         dbSize = hLen+padLen+messageSize;
338
339         // If testing, then use the provided seed. Otherwise, use values
340         // from the RNG
341     #ifdef  TEST_RSA
342         if(testSeed != NULL)
343             seed = testSeed;
344         else
345     #endif  // TEST_RSA
346             _cpri__GenerateRandom(hLen, mySeed);
347
348         // mask = MGF1 (seed, nSize  hLen  1)
349         if((retVal = _cpri__MGF1(dbSize, mask,  hashAlg, hLen, seed)) < 0)
350             return retVal; // Don't expect an error because hash size is not zero
351                            // was detected in the call to _cpri__HashBlock() above.
352
353         // Create the masked db
354         pm = mask;
355         for(i = dbSize; i > 0; i--)
```

```
356             *pp++ ^= *pm++;
357         pp = &padded[hLen + 1];
358 
359         // Run the masked data through MGF1
360         if((retVal = _cpri__MGF1(hLen, &padded[1],  hashAlg, dbSize, pp)) < 0)
361             return retVal; // Don't expect zero here as the only case for zero
362                            // was detected in the call to _cpri__HashBlock() above.
363 
364         // Now XOR the seed to create masked seed
365         pp = &padded[1];
366         pm = seed;
367         for(i = hLen; i > 0; i--)
368             *pp++ ^= *pm++;
369 
370         // Set the first byte to zero
371         *padded = 0x00;
372         return CRYPT_SUCCESS;
373     }
```

B.10.1.3.6. OaepDecode()

This function performs OAEP padding checking. The size of the buffer to receive the recovered data. If the padding is not valid, the *dSize* size is set to zero and the function returns CRYPT_NO_RESULTS.

The *dSize* parameter is used as an input to indicate the size available in the buffer. If insufficient space is available, the size is not changed and the return code is CRYPT_FAIL.

Return Value	Meaning
CRYPT_SUCCESS	decode complete
CRYPT_PARAMETER	the value to decode was larger than the modulus
CRYPT_FAIL	the padding is wrong or the buffer to receive the results is too small

```
374 static CRYPT_RESULT
375 OaepDecode(
376     UINT32          *dataOutSize,   // IN/OUT: the recovered data size
377     BYTE            *dataOut,       // OUT: the recovered data
378     TPM_ALG_ID       hashAlg,       // IN: algorithm to use for padding
379     const char      *label,         // IN: null-terminated string (may be NULL)
380     UINT32           paddedSize,    // IN: the size of the padded data
381     BYTE            *padded         // IN: the padded data
382 )
383 {
384     UINT32           dSizeSave;
385     UINT32           i;
386     BYTE             seedMask[MAX_DIGEST_SIZE];
387     INT32            hLen = _cpri__GetDigestSize(hashAlg);
388 
389     BYTE             mask[MAX_RSA_KEY_BYTES];
390     BYTE            *pp;
391     BYTE            *pm;
392     UINT32           lSize = 0;
393     CRYPT_RESULT     retVal = CRYPT_SUCCESS;
394 
395     // Unknown hash
396     pAssert(hLen > 0 && dataOutSize != NULL && dataOut != NULL && padded != NULL);
397 
398     // If there is a label, get its size including the terminating 0x00
399     if(label != NULL)
400         lSize = (UINT32)strlen(label) + 1;
401 
402     // Set the return size to zero so that it doesn't have to be done on each
403     // failure
```

```
404         dSizeSave = *dataOutSize;
405         *dataOutSize = 0;
406 
407         // Strange size (anything smaller can't be an OAEP padded block)
408         // Also check for no leading 0
409         if(paddedSize < (<K>unsigned)((2 * hLen) + 2) || *padded != 0)
410             return CRYPT_FAIL;
411 
412         // Use the hash size to determine what to put through MGF1 in order
413         // to recover the seedMask
414         if((retVal = _cpri__MGF1(hLen, seedMask,  hashAlg,
415                                  paddedSize-hLen-1, &padded[hLen+1])) < 0)
416             return retVal;
417 
418         // Recover the seed into seedMask
419         pp = &padded[1];
420         pm = seedMask;
421         for(i = hLen; i > 0; i--)
422             *pm++ ^= *pp++;
423 
424         // Use the seed to generate the data mask
425         if((retVal = _cpri__MGF1(paddedSize-hLen-1, mask,  hashAlg,
426                                  hLen, seedMask)) < 0)
427             return retVal;
428 
429         // Use the mask generated from seed to recover the padded data
430         pp = &padded[hLen+1];
431         pm = mask;
432         for(i = paddedSize-hLen-1; i > 0; i--)
433             *pm++ ^= *pp++;
434 
435         // Make sure that the recovered data has the hash of the label
436         // Put trial value in the seed mask
437         if((retVal=_cpri__HashBlock(hashAlg, lSize,(BYTE *)label, hLen, seedMask)) < 0)
438             return retVal;
439 
440         if(memcmp(seedMask, mask, hLen) != 0)
441             return CRYPT_FAIL;
442 
443 
444         // find the start of the data
445         pm = &mask[hLen];
446         for(i = paddedSize-(2*hLen)-1; i > 0; i--)
447         {
448             if(*pm++ != 0)
449                 break;
450         }
451         if(i == 0)
452             return CRYPT_PARAMETER;
453 
454         // pm should be pointing at the first part of the data
455         // and i is one greater than the number of bytes to move
456         i--;
457         if(i > dSizeSave)
458         {
459             // Restore dSize
460             *dataOutSize = dSizeSave;
461             return CRYPT_FAIL;
462         }
463         memcpy(dataOut, pm, i);
464         *dataOutSize = i;
465         return CRYPT_SUCCESS;
466     }
```

B.10.1.3.7. PKSC1v1_5Encode()

This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2. 1

Return Value	Meaning
CRYPT_SUCCESS	data encoded
CRYPT_PARAMETER	message size is too large

```
467  static CRYPT_RESULT
468  RSAES_PKSC1v1_5Encode(
469      UINT32        paddedSize,     // IN: pad value size
470      BYTE         *padded,         // OUT: the pad data
471      UINT32        messageSize,    // IN: the message size
472      BYTE         *message         // IN: the message being padded
473  )
474  {
475      UINT32      ps = paddedSize - messageSize - 3;
476      if(messageSize > paddedSize - 11)
477          return CRYPT_PARAMETER;
478
479      // move the message to the end of the buffer
480      memcpy(&padded[paddedSize - messageSize], message, messageSize);
481
482      // Set the first byte to 0x00 and the second to 0x02
483      *padded = 0;
484      padded[1] = 2;
485
486      // Fill with random bytes
487      _cpri__GenerateRandom(ps, &padded[2]);
488
489      // Set the delimiter for the random field to 0
490      padded[2+ps] = 0;
491
492      // Now, the only messy part. Make sure that all the ps bytes are non-zero
493      // In this implementation, use the value of the current index
494      for(ps++; ps > 1; ps--)
495      {
496          if(padded[ps] == 0)
497              padded[ps] = 0x55;     // In the < 0.5% of the cases that the random
498                                     // value is 0, just pick a value to put into
499                                     // the spot.
500      }
501      return CRYPT_SUCCESS;
502  }
```

B.10.1.3.8. RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2. 1

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decoding error or results would no fit into provided buffer

```
503  static CRYPT_RESULT
504  RSAES_Decode(
505      UINT32       *messageSize,     // IN/OUT: recovered message size
506      BYTE         *message,         // OUT: the recovered message
507      UINT32        codedSize,       // IN: the encoded message size
508      BYTE         *coded            // IN: the encoded message
509  )
```

```
510 {
511     BOOL        fail = FALSE;
512     UINT32      ps;
513
514     fail = (codedSize < 11);
515     fail |= (coded[0] != 0x00) || (coded[1] != 0x02);
516     for(ps = 2; ps < codedSize; ps++)
517     {
518         if(coded[ps] == 0)
519             break;
520     }
521     ps++;
522
523     // Make sure that ps has not gone over the end and that there are at least 8
524     // bytes of pad data.
525     fail |= ((ps >= codedSize) || ((ps-2) < 8));
526     if((*messageSize < codedSize - ps) || fail)
527         return CRYPT_FAIL;
528
529     *messageSize = codedSize - ps;
530     memcpy(message, &coded[ps], codedSize - ps);
531     return CRYPT_SUCCESS;
532 }
```

B.10.1.3.9. PssEncode()

This function creates an encoded block of data that is the size of modulus. The function uses the maximum salt size that will fit in the encoded block.

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	hashAlg is not a supported hash algorithm

```
533 static CRYPT_RESULT
534 PssEncode    (
535     UINT32       eOutSize,       // IN: size of the encode data buffer
536     BYTE        *eOut,           // OUT: encoded data buffer
537     TPM_ALG_ID   hashAlg,        // IN: hash algorithm to use for the encoding
538     UINT32       hashInSize,     // IN: size of digest to encode
539     BYTE        *hashIn          // IN: the digest
540 #ifdef TEST_RSA                  //
541     ,  BYTE        *saltIn       // IN: optional parameter for testing
542 #endif // TEST_RSA               //
543 )
544 {
545     INT32               hLen = _cpri__GetDigestSize(hashAlg);
546     BYTE                salt[MAX_RSA_KEY_BYTES - 1];
547     UINT16              saltSize;
548     BYTE               *ps = salt;
549     CRYPT_RESULT        retVal;
550     UINT16              mLen;
551     CPRI_HASH_STATE     hashState;
552
553     // These are fatal errors indicating bad TPM firmware
554     pAssert(eOut != NULL && hLen > 0 && hashIn != NULL );
555
556     // Get the size of the mask
557     mLen = (UINT16)(eOutSize - hLen - 1);
558
559     // Use the maximum salt size
560     saltSize = mLen - 1;
561
```

```
562 //using eOut for scratch space
563     // Set the first 8 bytes to zero
564     memset(eOut, 0, 8);
565
566
567     // Get set the salt
568 #ifdef  TEST_RSA
569     if(saltIn != NULL)
570     {
571         saltSize = hLen;
572         memcpy(salt, saltIn, hLen);
573     }
574     else
575 #endif  // TEST_RSA
576         _cpri__GenerateRandom(saltSize, salt);
577
578     // Create the hash of the pad || input hash || salt
579     _cpri__StartHash(hashAlg, FALSE, &hashState);
580     _cpri__UpdateHash(&hashState, 8, eOut);
581     _cpri__UpdateHash(&hashState, hashInSize, hashIn);
582     _cpri__UpdateHash(&hashState, saltSize, salt);
583     _cpri__CompleteHash(&hashState, hLen, &eOut[eOutSize - hLen - 1]);
584
585     // Create a mask
586     if((retVal = _cpri__MGF1(mLen, eOut, hashAlg, hLen, &eOut[mLen])) < 0)
587     {
588         // Currently _cpri__MGF1 is not expected to return a CRYPT_RESULT error.
589         pAssert(0);
590         return retVal;
591     }
592     // Since this implementation uses key sizes that are all even multiples of
593     // 8, just need to make sure that the most significant bit is CLEAR
594     eOut[0] &= 0x7f;
595
596     // Before we mess up the eOut value, set the last byte to 0xbc
597     eOut[eOutSize - 1] = 0xbc;
598
599     // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
600     eOut = &eOut[mLen - saltSize - 1];
601     *eOut++ ^= 0x01;
602
603     // XOR the salt data into the buffer
604     for(; saltSize > 0; saltSize--)
605         *eOut++ ^= *ps++;
606
607     // and we are done
608     return CRYPT_SUCCESS;
609 }
```

B.10.1.3.10. 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

```
610 static CRYPT_RESULT
611 PssDecode(
612     TPM_ALG_ID   hashAlg,        // IN: hash algorithm to use for the encoding
613     UINT32       dInSize,        // IN: size of the digest to compare
614     BYTE        *dIn,            // In: the digest to compare
615     UINT32       eInSize,        // IN: size of the encoded data
616     BYTE        *eIn,            // IN: the encoded data
617     UINT32       saltSize        // IN: the expected size of the salt
618 )
619 {
620     INT32            hLen = _cpri__GetDigestSize(hashAlg);
621     BYTE             mask[MAX_RSA_KEY_BYTES];
622     BYTE            *pm = mask;
623     BYTE             pad[8] = {0};
624     UINT32           i;
625     UINT32           mLen;
626     BOOL             fail = FALSE;
627     CRYPT_RESULT     retVal;
628     CPRI_HASH_STATE  hashState;
629
630     // These errors are indicative of failures due to programmer error
631     pAssert(dIn != NULL && eIn != NULL);
632
633     // check the hash scheme
634     if(hLen == 0)
635         return CRYPT_SCHEME;
636
637     // most significant bit must be zero
638     fail = ((eIn[0] & 0x80) != 0);
639
640     // last byte must be 0xbc
641     fail |= (eIn[eInSize - 1] != 0xbc);
642
643     // Use the hLen bytes at the end of the buffer to generate a mask
644     // Doesn't start at the end which is a flag byte
645     mLen = eInSize - hLen - 1;
646     if((retVal = _cpri__MGF1(mLen, mask, hashAlg, hLen, &eIn[mLen])) < 0)
647         return retVal;
648     if(retVal == 0)
649         return CRYPT_FAIL;
650
651     // Clear the MSO of the mask to make it consistent with the encoding.
652     mask[0] &= 0x7F;
653
654     // XOR the data into the mask to recover the salt. This sequence
655     // advances eIn so that it will end up pointing to the seed data
656     // which is the hash of the signature data
657     for(i = mLen; i > 0; i--)
658         *pm++ ^= *eIn++;
659
660     // Find the first byte of 0x01 after a string of all 0x00
661     for(pm = mask, i = mLen; i > 0; i--)
662     {
663         if(*pm == 0x01)
664             break;
665         else
666             fail |= (*pm++ != 0);
667     }
```

```
668         fail |= (i == 0);
669
670         // if we have failed, will continue using the entire mask as the salt value so
671         // that the timing attacks will not disclose anything (I don't think that this
672         // is a problem for TPM applications but, usually, we don't fail so this
673         // doesn't cost anything).
674         if(fail)
675         {
676             i = mLen;
677             pm = mask;
678         }
679         else
680         {
681             pm++;
682             i--;
683         }
684         // If the salt size was provided, then the recovered size must match
685         fail |= (saltSize != 0 && i != saltSize);
686
687         // i contains the salt size and pm points to the salt. Going to use the input
688         // hash and the seed to recreate the hash in the lower portion of eIn.
689         _cpri__StartHash(hashAlg, FALSE, &hashState);
690
691         // add the pad of 8 zeros
692         _cpri__UpdateHash(&hashState, 8, pad);
693
694         // add the provided digest value
695         _cpri__UpdateHash(&hashState, dInSize, dIn);
696
697         // and the salt
698         _cpri__UpdateHash(&hashState, i, pm);
699
700         // get the result
701         retVal = _cpri__CompleteHash(&hashState, MAX_DIGEST_SIZE, mask);
702
703         // retVal will be the size of the digest or zero. If not equal to the indicated
704         // digest size, then the signature doesn't match
705         fail |= (retVal != hLen);
706         fail |= (memcmp(mask, eIn, hLen) != 0);
707         if(fail)
708             return CRYPT_FAIL;
709         else
710             return CRYPT_SUCCESS;
711     }
```

B.10.1.3.11. 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*

```
712     static CRYPT_RESULT
713     RSASSA_Encode(
714         UINT32           eOutSize,      // IN: the size of the resulting block
715         BYTE            *eOut,          // OUT: the encoded block
716         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for PKSC1v1_5
717         UINT32           hInSize,       // IN: size of hash to be signed
718         BYTE            *hIn            // IN: hash buffer
```

```
719 )
720 {
721     BYTE            *der;
722     INT32            derSize = _cpri__GetHashDER(hashAlg, &der);
723     INT32            fillSize;
724
725     pAssert(eOut != NULL && hIn != NULL);
726
727     // Can't use this scheme if the algorithm doesn't have a DER string defined.
728     if(derSize == 0 )
729         return CRYPT_SCHEME;
730
731     // If the digest size of 'hashAl' doesn't match the input digest size, then
732     // the DER will misidentify the digest so return an error
733     if((unsigned)_cpri__GetDigestSize(hashAlg) != hInSize)
734         return CRYPT_PARAMETER;
735
736     fillSize = eOutSize - derSize - hInSize - 3;
737
738     // Make sure that this combination will fit in the provided space
739     if(fillSize < 8)
740         return CRYPT_PARAMETER;
741     // Start filling
742     *eOut++ = 0; // initial byte of zero
743     *eOut++ = 1; // byte of 0x01
744     for(; fillSize > 0; fillSize--)
745         *eOut++ = 0xff; // bunch of 0xff
746     *eOut++ = 0; // another 0
747     for(; derSize > 0; derSize--)
748         *eOut++ = *der++;    // copy the DER
749     for(; hInSize > 0; hInSize--)
750         *eOut++ = *hIn++;    // copy the hash
751     return CRYPT_SUCCESS;
752 }
```

B.10.1.3.12. 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

```
753 static CRYPT_RESULT
754 RSASSA_Decode(
755     TPM_ALG_ID       hashAlg,        // IN: hash algorithm to use for the encoding
756     UINT32           hInSize,        // IN: size of the digest to compare
757     BYTE            *hIn,            // In: the digest to compare
758     UINT32           eInSize,        // IN: size of the encoded data
759     BYTE            *eIn             // IN: the encoded data
760 )
761 {
762     BOOL             fail = FALSE;
763     BYTE            *der;
764     INT32            derSize = _cpri__GetHashDER(hashAlg, &der);
765     INT32            hashSize = _cpri__GetDigestSize(hashAlg);
766     INT32            fillSize;
767
768     pAssert(hIn != NULL && eIn != NULL);
769
770     // Can't use this scheme if the algorithm doesn't have a DER string
```

```
771         // defined or if the provided hash isn't the right size
772         if(derSize == 0 || (unsigned)hashSize != hInSize)
773             return CRYPT_SCHEME;
774 
775         // Make sure that this combination will fit in the provided space
776         // Since no data movement takes place, can just walk though this
777         // and accept nearly random values. This can only be called from
778         // _cpri__ValidateSignature() so eInSize is known to be in range.
779         fillSize = eInSize - derSize - hashSize - 3;
780 
781         // Start checking
782         fail |= (*eIn++ != 0); // initial byte of zero
783         fail |= (*eIn++ != 1); // byte of 0x01
784         for(; fillSize > 0; fillSize--)
785             fail |= (*eIn++ != 0xff); // bunch of 0xff
786         fail |= (*eIn++ != 0); // another 0
787         for(; derSize > 0; derSize--)
788             fail |= (*eIn++ != *der++); // match the DER
789         for(; hInSize > 0; hInSize--)
790             fail |= (*eIn++ != *hIn++); // match the hash
791         if(fail)
792             return CRYPT_FAIL;
793         return CRYPT_SUCCESS;
794     }
```

B.10.1.4. Externally Accessible Functions

B.10.1.4.1. _cpri__RsaStartup()

Function that is called to initialize the hash service. In this implementation, this function does nothing but it is called by the CryptUtilStartup() function and must be present.

```
795     BOOL
796     _cpri__RsaStartup(
797         void
798     )
799     {
800         return TRUE;
801     }
```

B.10.1.4.2. _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

```
802 CRYPT_RESULT
803 _cpri__EncryptRSA(
804     UINT32      *cOutSize,      // OUT: the size of the encrypted data
805     BYTE        *cOut,          // OUT: the encrypted data
806     RSA_KEY     *key,           // IN: the key to use for encryption
807     TPM_ALG_ID   padType,       // IN: the type of padding
808     UINT32       dInSize,       // IN: the amount of data to encrypt
809     BYTE        *dIn,           // IN: the data to encrypt
810     TPM_ALG_ID   hashAlg,       // IN: in case this is needed
811     const char  *label          // IN: in case it is needed
812 )
813 {
814     CRYPT_RESULT    retVal = CRYPT_SUCCESS;
815
816     pAssert(cOutSize != NULL);
817
818     // All encryption schemes return the same size of data
819     if(*cOutSize < key->publicKey->size)
820         return CRYPT_PARAMETER;
821     *cOutSize = key->publicKey->size;
822
823     switch (padType)
824     {
825     case TPM_ALG_NULL:  // 'raw' encryption
826         {
827             // dIn can have more bytes than cOut as long as the extra bytes
828             // are zero
829             for(; dInSize > *cOutSize; dInSize--)
830             {
831                 if(*dIn++ != 0)
832                     return CRYPT_PARAMETER;
833
834             }
835             // If dIn is smaller than cOut, fill cOut with zeros
836             if(dInSize < *cOutSize)
837                 memset(cOut, 0, *cOutSize - dInSize);
838
839             // Copy the rest of the value
840             memcpy(&cOut[*cOutSize-dInSize], dIn, dInSize);
841             // If the size of dIn is the same as cOut dIn could be larger than
842             // the modulus. If it is, then RSAEP() will catch it.
843         }
844         break;
845     case TPM_ALG_RSAES:
846         retVal = RSAES_PKSC1v1_5Encode(*cOutSize, cOut, dInSize, dIn);
847         break;
848     case TPM_ALG_OAEP:
849         retVal = OaepEncode(*cOutSize, cOut, hashAlg, label, dInSize, dIn
850 #ifdef  TEST_RSA
851                             ,NULL
852 #endif
853                            );
854         break;
855     default:
856         return CRYPT_SCHEME;
857     }
858     // All the schemes that do padding will come here for the encryption step
859     // Check that the Encoding worked
```

```
860         if(retVal != CRYPT_SUCCESS)
861             return retVal;
862 
863         // Padding OK so do the encryption
864         return RSAEP(*cOutSize, cOut, key);
865     }
```

B.10.1.4.3. _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

```
866 CRYPT_RESULT
867 _cpri__DecryptRSA(
868     UINT32      *dOutSize,      // OUT: the size of the decrypted data
869     BYTE        *dOut,          // OUT: the decrypted data
870     RSA_KEY     *key,           // IN: the key to use for decryption
871     TPM_ALG_ID   padType,       // IN: the type of padding
872     UINT32       cInSize,       // IN: the amount of data to decrypt
873     BYTE        *cIn,           // IN: the data to decrypt
874     TPM_ALG_ID   hashAlg,       // IN: in case this is needed for the scheme
875     const char  *label          // IN: in case it is needed for the scheme
876 )
877 {
878     CRYPT_RESULT    retVal;
879 
880     // Make sure that the necessary parameters are provided
881     pAssert(cIn != NULL && dOut != NULL && dOutSize != NULL && key != NULL);
882 
883     // Size is checked to make sure that the decryption works properly
884     if(cInSize != key->publicKey->size)
885         return CRYPT_PARAMETER;
886 
887     // For others that do padding, do the decryption in place and then
888     // go handle the decoding.
889     if((retVal = RSADP(cInSize, cIn, key)) != CRYPT_SUCCESS)
890         return retVal;      // Decryption failed
891 
892     // Remove padding
893     switch (padType)
894     {
895     case TPM_ALG_NULL:
896         if(*dOutSize < key->publicKey->size)
897             return CRYPT_FAIL;
898         *dOutSize = key->publicKey->size;
899         memcpy(dOut, cIn, *dOutSize);
900         return CRYPT_SUCCESS;
901     case TPM_ALG_RSAES:
902         return RSAES_Decode(dOutSize, dOut, cInSize, cIn);
903         break;
904     case TPM_ALG_OAEP:
905         return OaepDecode(dOutSize, dOut, hashAlg, label, cInSize, cIn);
906         break;
907     default:
```

```
908             return CRYPT_SCHEME;
909             break;
910         }
911     }
```

B.10.1.4.4. _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)

```
912     CRYPT_RESULT
913     _cpri__SignRSA(
914         UINT32          *sigOutSize,    // OUT: size of signature
915         BYTE            *sigOut,        // OUT: signature
916         RSA_KEY         *key,           // IN: key to use
917         TPM_ALG_ID       scheme,        // IN: the scheme to use
918         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for PKSC1v1_5
919         UINT32           hInSize,       // IN: size of digest to be signed
920         BYTE            *hIn            // IN: digest buffer
921     )
922     {
923         CRYPT_RESULT    retVal;
924
925         // Parameter checks
926         pAssert(sigOutSize != NULL && sigOut != NULL && key != NULL && hIn != NULL);
927
928
929         // For all signatures the size is the size of the key modulus
930         *sigOutSize = key->publicKey->size;
931         switch (scheme)
932         {
933         case TPM_ALG_NULL:
934             *sigOutSize = 0;
935             return CRYPT_SUCCESS;
936         case TPM_ALG_RSAPSS:
937             // PssEncode can return CRYPT_PARAMETER
938             retVal = PssEncode(*sigOutSize, sigOut, hashAlg, hInSize, hIn
939     #ifdef  TEST_RSA
940                                , NULL
941     #endif
942                               );
943             break;
944         case TPM_ALG_RSASSA:
945             // RSASSA_Encode can return CRYPT_PARAMETER or CRYPT_SCHEME
946             retVal = RSASSA_Encode(*sigOutSize, sigOut, hashAlg, hInSize, hIn);
947             break;
948         default:
949             return CRYPT_SCHEME;
950         }
951         if(retVal != CRYPT_SUCCESS)
952             return retVal;
953         // Do the encryption using the private key
954         // RSADP can return CRYPT_PARAMETR
955         return RSADP(*sigOutSize,sigOut, key);
956     }
```

B.10.1.4.5. _cpri__ValidateSignatureRSA()

This function is used to validate an RSA signature. If the signature is valid CRYPT_SUCCESS is returned. If the signature is not valid, CRYPT_FAIL is returned. Other return codes indicate either parameter problems or fatal errors.

Return Value	Meaning
CRYPT_SUCCESS	the signature checks
CRYPT_FAIL	the signature does not check
CRYPT_SCHEME	unsupported scheme or hash algorithm

```
957  CRYPT_RESULT
958  _cpri__ValidateSignatureRSA(
959      RSA_KEY          *key,           // IN: key to use
960      TPM_ALG_ID        scheme,        // IN: the scheme to use
961      TPM_ALG_ID        hashAlg,       // IN: hash algorithm
962      UINT32            hInSize,       // IN: size of digest to be checked
963      BYTE             *hIn,           // IN: digest buffer
964      UINT32            sigInSize,     // IN: size of signature
965      BYTE             *sigIn,         // IN: signature
966      UINT16            saltSize       // IN: salt size for PSS
967  )
968  {
969      CRYPT_RESULT      retVal;
970
971      // Fatal programming errors
972      pAssert(key != NULL && sigIn != NULL && hIn != NULL);
973
974      // Errors that might be caused by calling parameters
975      if(sigInSize != key->publicKey->size)
976          return CRYPT_FAIL;
977      // Decrypt the block
978      if((retVal = RSAEP(sigInSize, sigIn, key)) != CRYPT_SUCCESS)
979          return CRYPT_FAIL;
980      switch (scheme)
981      {
982      case TPM_ALG_NULL:
983          return CRYPT_SCHEME;
984          break;
985      case TPM_ALG_RSAPSS:
986          return PssDecode(hashAlg, hInSize, hIn, sigInSize, sigIn, saltSize);
987          break;
988      case TPM_ALG_RSASSA:
989          return RSASSA_Decode(hashAlg, hInSize, hIn, sigInSize, sigIn);
990          break;
991      default:
992          break;
993      }
994      return CRYPT_SCHEME;
995  }
996  #ifndef RSA_KEY_SIEVE       //%
```

B.10.1.4.6. _cpri__GenerateKeyRSA()

Generate an RSA key from a provided seed

Return Value	Meaning
CRYPT_FAIL	exponent is not prime or is less than 3; or could not find a prime using the provided parameters
CRYPT_CANCEL	operation was cancelled

```
 997  CRYPT_RESULT
 998  _cpri__GenerateKeyRSA(
 999      TPM2B           *n,             // OUT: The public modulus
1000      TPM2B           *p,             // OUT: One of the prime factors of n
1001      UINT16           keySizeInBits, // IN: Size of the public modulus in bits
1002      UINT32           e,             // IN: The public exponent
1003      TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use in the key
1004                                      //     generation process
1005      TPM2B           *seed,          // IN: the seed to use
1006      const char      *label,         // IN: A label for the generation process.
1007      TPM2B           *extra,         // IN: Party 1 data for the KDF
1008      UINT32          *counter        // IN/OUT: Counter value to allow KFD iteration
1009                                      //         to be propagated across multiple
1010                                      //         routines
1011  )
1012  {
1013      UINT32           lLen;          // length of the label
1014                                      // (counting the terminating 0);
1015      UINT16           digestSize = _cpri__GetDigestSize(hashAlg);
1016
1017      TPM2B_HASH_BLOCK    oPadKey;
1018
1019      UINT32          outer;
1020      UINT32          inner;
1021      BYTE            swapped[4];
1022
1023      CRYPT_RESULT    retVal;
1024      int             i, fill;
1025      const static char     defaultLabel[] = "RSA key";
1026      BYTE            *pb;
1027
1028
1029      CPRI_HASH_STATE  h1;            // contains the hash of the
1030                                      //   HMAC key w/ iPad
1031      CPRI_HASH_STATE  h2;            // contains the hash of the
1032                                      //   HMAC key w/ oPad
1033      CPRI_HASH_STATE  h;             // the working hash context
1034
1035      BIGNUM          *bnP;
1036      BIGNUM          *bnQ;
1037      BIGNUM          *bnT;
1038      BIGNUM          *bnE;
1039      BIGNUM          *bnN;
1040      BN_CTX          *context;
1041      UINT32           rem;
1042
1043      // Make sure that hashAlg is valid hash
1044      pAssert(digestSize != 0);
1045
1046      // if present, use externally provided counter
1047      if(counter != NULL)
1048          outer = *counter;
1049      else
1050          outer = 1;
1051
1052      // Validate exponent
1053      UINT32_TO_BYTE_ARRAY(e, swapped);
1054
```

```
1055        // Need to check that the exponent is prime and not less than 3
1056        if( e != 0 && (e < 3  || !_math__IsPrime(e)))
1057            return CRYPT_FAIL;
1058
1059        // Get structures for the big number representations
1060        context = BN_CTX_new();
1061        if(context == NULL)
1062            FAIL(FATAL_ERROR_ALLOCATION);
1063        BN_CTX_start(context);
1064        bnP = BN_CTX_get(context);
1065        bnQ = BN_CTX_get(context);
1066        bnT = BN_CTX_get(context);
1067        bnE = BN_CTX_get(context);
1068        bnN = BN_CTX_get(context);
1069        if(bnN == NULL)
1070            FAIL(FATAL_ERROR_INTERNAL);
1071
1072        // Set Q to zero. This is used as a flag. The prime is computed in P. When a
1073        // new prime is found, Q is checked to see if it is zero.  If so, P is copied
1074        // to Q and a new P is found.  When both P and Q are non-zero, the modulus and
1075        // private exponent are computed and a trial encryption/decryption is
1076        // performed.  If the encrypt/decrypt fails, assume that at least one of the
1077        // primes is composite. Since we don't know which one, set Q to zero and start
1078        // over and find a new pair of primes.
1079        BN_zero(bnQ);
1080
1081        // Need to have some label
1082        if(label == NULL)
1083            label = (const char *)&defaultLabel;
1084        // Get the label size
1085        for(lLen = 0; label[lLen++] != 0;);
1086
1087
1088        // Start the hash using the seed and get the intermediate hash value
1089        _cpri__StartHMAC(hashAlg, FALSE, &h1, seed->size, seed->buffer, &oPadKey.b);
1090        _cpri__StartHash(hashAlg, FALSE, &h2);
1091        _cpri__UpdateHash(&h2, oPadKey.b.size, oPadKey.b.buffer);
1092
1093        n->size = keySizeInBits/8;
1094        pAssert(n->size <= MAX_RSA_KEY_BYTES);
1095        p->size = n->size / 2;
1096        if(e == 0)
1097            e = RSA_DEFAULT_PUBLIC_EXPONENT;
1098
1099        BN_set_word(bnE, e);
1100
1101        // The first test will increment the counter from zero.
1102        for(outer += 1; outer != 0; outer++)
1103        {
1104            if(_plat__IsCanceled())
1105            {
1106                retVal = CRYPT_CANCEL;
1107                goto Cleanup;
1108            }
1109
1110            // Need to fill in the candidate with the hash
1111            fill = digestSize;
1112            pb = p->buffer;
1113
1114            // Reset the inner counter
1115            inner = 0;
1116            for(i = p->size; i > 0; i -= digestSize)
1117            {
1118                inner++;
1119                // Initialize the HMAC with saved state
1120                _cpri__CopyHashState(&h, &h1);
```

```
1121
1122                // Hash the inner counter (the one that changes on each HMAC iteration)
1123                UINT32_TO_BYTE_ARRAY(inner, swapped);
1124                _cpri__UpdateHash(&h, 4, swapped);
1125                _cpri__UpdateHash(&h, lLen, (BYTE *)label);
1126
1127                // Is there any party 1 data
1128                if(extra != NULL)
1129                    _cpri__UpdateHash(&h, extra->size, extra->buffer);
1130
1131                // Include the outer counter (the one that changes on each prime
1132                // prime candidate generation
1133                UINT32_TO_BYTE_ARRAY(outer, swapped);
1134                _cpri__UpdateHash(&h, 4, swapped);
1135                _cpri__UpdateHash(&h, 2, (BYTE *)&keySizeInBits);
1136                if(i < fill)
1137                    fill = i;
1138                _cpri__CompleteHash(&h, fill, pb);
1139
1140                // Restart the oPad hash
1141                _cpri__CopyHashState(&h, &h2);
1142
1143                // Add the last hashed data
1144                _cpri__UpdateHash(&h, fill, pb);
1145
1146                // gives a completed HMAC
1147                _cpri__CompleteHash(&h, fill, pb);
1148                pb += fill;
1149            }
1150            // Set the Most significant 2 bits and the low bit of the candidate
1151            p->buffer[0] |= 0xC0;
1152            p->buffer[p->size - 1] |= 1;
1153
1154            // Convert the candidate to a BN
1155            BN_bin2bn(p->buffer, p->size, bnP);
1156
1157            // If this is the second prime, make sure that it differs from the
1158            // first prime by at least 2^100
1159            if(!BN_is_zero(bnQ))
1160            {
1161                // bnQ is non-zero if we already found it
1162                if(BN_ucmp(bnP, bnQ) < 0)
1163                    BN_sub(bnT, bnQ, bnP);
1164                else
1165                    BN_sub(bnT, bnP, bnQ);
1166                if(BN_num_bits(bnT) < 100)  <Q>// Difference has to be at least 100 bits
1167                    continue;
1168            }
1169            // Make sure that the prime candidate (p) is not divisible by the exponent
1170            // and that (p-1) is not divisible by the exponent
1171            // Get the remainder after dividing by the modulus
1172            rem = BN_mod_word(bnP, e);
1173            if(rem == 0) // evenly divisible so add two keeping the number odd and
1174                // making sure that 1 != p mod e
1175                BN_add_word(bnP, 2);
1176            else if(rem == 1) // leaves a remainder of 1 so subtract two keeping the
1177                // number odd and making (e-1) = p mod e
1178                BN_sub_word(bnP, 2);
1179
1180            // Have a candidate, check for primality
1181            if((retVal = (CRYPT_RESULT)BN_is_prime_ex(bnP,
1182                        BN_prime_checks, NULL, NULL)) < 0)
1183                FAIL(FATAL_ERROR_INTERNAL);
1184
1185            if(retVal != 1)
1186                continue;
```

```
1187
1188            // Found a prime, is this the first or second.
1189            if(BN_is_zero(bnQ))
1190            {
1191                // copy p to q and compute another prime in p
1192                BN_copy(bnQ, bnP);
1193                continue;
1194            }
1195            //Form the public modulus
1196            BN_mul(bnN, bnP, bnQ, context);
1197            if(BN_num_bits(bnN) != keySizeInBits)
1198                FAIL(FATAL_ERROR_INTERNAL);
1199
1200            // Save the public modulus
1201            BnTo2B(n, bnN, 0);  // Fills the buffer with the correct size
1202            pAssert(   (n->size == (keySizeInBits + 7) / 8) && ((n->buffer[0] & 0x80)
1203                    != 0));
1204
1205            // And one prime
1206            BnTo2B(p, bnP, 0);
1207            pAssert((p->size == n->size/2) && ((p->buffer[0] & 0x80) != 0));
1208
1209            // Finish by making sure that we can form the modular inverse of PHI
1210            // with respect to the public exponent
1211            // Compute PHI = (p - 1)(q - 1) = n - p - q + 1
1212            // Make sure that we can form the modular inverse
1213            BN_sub(bnT, bnN, bnP);
1214            BN_sub(bnT, bnT, bnQ);
1215            BN_add_word(bnT, 1);
1216
1217            // find d such that (Phi * d) mod e ==1
1218            // If there isn't then we are broken because we took the step
1219            // of making sure that the prime != 1 mod e so the modular inverse
1220            // must exist
1221            if(BN_mod_inverse(bnT, bnE, bnT, context) == NULL || BN_is_zero(bnT))
1222                FAIL(FATAL_ERROR_INTERNAL);
1223
1224            // And, finally, do a trial encryption decryption
1225            {
1226                TPM2B_TYPE(RSA_KEY, MAX_RSA_KEY_BYTES);
1227                TPM2B_RSA_KEY       r;
1228                r.t.size = sizeof(n->size);
1229
1230                // If we are using a seed, then results must be reproducible on each
1231                // call. Otherwise, just get a random number
1232                if(seed == NULL)
1233                    _cpri__GenerateRandom(n->size, r.t.buffer);
1234                else
1235                {
1236                    // this this version does not have a deterministic RNG, XOR the
1237                    // public key and private exponent to get a deterministic value
1238                    // for testing.
1239                    int         i;
1240
1241                    // Generate a random-ish number starting with the public modulus
1242                    // XORed with the MSO of the seed
1243                    for(i = 0; i < n->size; i++)
1244                        r.t.buffer[i] = n->buffer[i] ^ seed->buffer[0];
1245                }
1246                // Make sure that the number is smaller than the public modulus
1247                r.t.buffer[0] &= 0x7F;
1248                      // Convert
1249                if(   BN_bin2bn(r.t.buffer, r.t.size, bnP) == NULL
1250                      // Encrypt with the public exponent
1251                   || BN_mod_exp(bnQ, bnP, bnE, bnN, context) != 1
1252                      // Decrypt with the private exponent
```

```
1253                || BN_mod_exp(bnQ, bnQ, bnT, bnN, context) != 1)
1254                 FAIL(FATAL_ERROR_INTERNAL);
1255             // If the starting and ending values are not the same, start over )-;
1256             if(BN_ucmp(bnP, bnQ) != 0)
1257             {
1258                 BN_zero(bnQ);
1259                 continue;
1260             }
1261         }
1262         retVal = CRYPT_SUCCESS;
1263         goto Cleanup;
1264     }
1265     retVal = CRYPT_FAIL;
1266
1267
1268 Cleanup:
1269     // Close out the hash sessions
1270     _cpri__CompleteHash(&h2, 0, NULL);
1271     _cpri__CompleteHash(&h1, 0, NULL);
1272
1273     // Free up allocated BN values
1274     BN_CTX_end(context);
1275     BN_CTX_free(context);
1276     if(counter != NULL)
1277         *counter = outer;
1278     return retVal;
1279 }
1280 #endif      // RSA_KEY_SIEVE             //%
```

B.10.2. Alternative RSA Key Generation

B.10.2.1. 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.10.1 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.10.2.2. 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.

```
4   #define PRIME_DIFF_TABLE_512_BYTE_PAGES     13
```

This set of macros used the value above to set the table size.

```
5   #ifndef PRIME_DIFF_TABLE_512_BYTE_PAGES
6   #   define PRIME_DIFF_TABLE_512_BYTE_PAGES   4
7   #endif
8   #ifdef PRIME_DIFF_TABLE_512_BYTE_PAGES
9   #   if PRIME_DIFF_TABLE_512_BYTE_PAGES > 12
10  #       define PRIME_DIFF_TABLE_BYTES 6542
11  #   else
12  #       if  PRIME_DIFF_TABLE_512_BYTE_PAGES <= 0
13  #           define PRIME_DIFF_TABLE_BYTES 512
14  #       else
15  #           define PRIME_DIFF_TABLE_BYTES (PRIME_DIFF_TABLE_512_BYTE_PAGES * 512)
16  #       endif
17  #   endif
18  #endif
19  extern const BYTE primeDiffTable [PRIME_DIFF_TABLE_BYTES];
```

This determines the number of bits in the sieve field This must be a power of two.

```
20  #define FIELD_POWER     14  // This is the only value in this group that should be
21                              // changed
22  #define FIELD_BITS      (1 << FIELD_POWER)
23  #define MAX_FIELD_SIZE      ((FIELD_BITS / 8) + 1)
```

This is the pre-sieved table. It already has the bits for multiples of 3, 5, and 7 cleared.

```
24  #define SEED_VALUES_SIZE        105
25  const extern BYTE               seedValues[SEED_VALUES_SIZE];
```

This allows determination of the number of bits that are set in a byte without having to count them individually.

```
26  const extern BYTE               bitsInByte[256];
```

This is the iterator structure for accessing the compressed prime number table. The expectation is that values will need to be accesses sequentially. This tries to save some data access.

```
27  typedef struct {
28      UINT32      lastPrime;
29      UINT32      index;
30      UINT32      final;
31  } PRIME_ITERATOR;
32  #ifdef  RSA_INSTRUMENT
33  #   define INSTRUMENT_SET(a, b) ((a) = (b))
34  #   define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
35  #   define INSTRUMENT_INC(a)    (a) = (a) + 1
36  extern UINT32  failedAtIteration[10];
37  extern UINT32  MillerRabinTrials;
38  extern UINT32  totalFieldsSieved;
39  extern UINT32  emptyFieldsSieved;
40  extern UINT32  noPrimeFields;
41  extern UINT32  primesChecked;
42  extern UINT16   lastSievePrime;
```

 Copyright © TCG 2006-2013

```
43  #else
44  #   define INSTRUMENT_SET(a, b)
45  #   define INSTRUMENT_ADD(a, b)
46  #   define INSTRUMENT_INC(a)
47  #endif
48  #ifdef RSA_DEBUG
49  extern UINT16   defaultFieldSize;
50  #define NUM_PRIMES          2047
51  extern const __int16        primes[NUM_PRIMES];
52  #else
53  #define defaultFieldSize    MAX_FIELD_SIZE
54  #endif
55  #endif
```

B.10.2.3. RSAKeySieve.c

B.10.2.3.1. Includes and defines

```
#include    "OsslCryptoEngine.h"
```

This file produces no code unless the compile switch is set to cause it to generate code.

```
#ifdef      RSA_KEY_SIEVE               //%
#include    "RsaKeySieve.h"
```

This next line will show up in the header file for this code. It will make the local functions public when debugging.

```
//%#ifdef   RSA_DEBUG
```

B.10.2.3.2. Bit Manipulation Functions

Introduction

These functions operate on a bit array. A bit array is an array of bytes with the 0th byte being the byte with the lowest memory address. Within the byte, bit 0 is the least significant bit.

ClearBit()

This function will CLEAR a bit in a bit array.

```
void
ClearBit(
    unsigned char       *a,             // IN: A pointer to an array of bytes
    int                  i              // IN: the number of the bit to CLEAR
    )
{
    a[i >> 3] &= 0xff ^ (1 << (i & 7));
}
```

SetBit()

Function to SET a bit in a bit array.

```
void
SetBit(
    unsigned char       *a,             // IN: A pointer to an array of bytes
    int                  i              // IN: the number of the bit to SET
    )
{
    a[i >> 3] |= (1 << (i & 7));
}
```

IsBitSet()

Function to test if a bit in a bit array is SET.

Return Value	Meaning
0	bit is CLEAR
1	bit is SET

```
UINT32
IsBitSet(
```

```
23      unsigned char       *a,                 // IN: A pointer to an array of bytes
24      int                  i                  // IN: the number of the bit to test
25      )
26  {
27      return ((a[i >> 3] & (1 << (i & 7))) != 0);
28  }
```

BitsInArry()

This function counts the number of bits set in an array of bytes.

```
29  int
30  BitsInArray(
31      unsigned char       *a,                 // IN: A pointer to an array of bytes
32      int                  i                  // IN: the number of bytes to sum
33      )
34  {
35      int     j = 0;
36      for(; i ; i--)
37          j += bitsInByte[*a++];
38      return j;
39  }
```

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

```
40  UINT32
41  FindNthSetBit(
42      const UINT16     aSize,     // IN: the size of the array to check
43      const BYTE      *a,         // IN: the array to check
44      const UINT32     n          // IN, the number of the SET bit
45  )
46  {
47      UINT32       i;
48      const BYTE  *pA = a;
49      UINT32       retValue;
50      BYTE         sel;
51
52      (aSize);
53
54      //find the bit
55      for(i = 0; i < n; i += bitsInByte[*pA++]);
56
57      // The chosen bit is in the byte that was just accessed
58      // Compute the offset to the start of that byte
59      pA--;
60      retValue = (pA - a) * 8;
61
62      // Subtract the bits in the last byte added.
63      i -= bitsInByte[*pA];
64
65      // Now process the byte, one bit at a time.
66      for(sel = *pA; sel != 0 ; sel = sel >> 1)
67      {
68          if(sel & 1)
69          {
70              i += 1;
71              if(i == n)
72                  return retValue;
73          }
74          retValue += 1;
75      }
76      FAIL(FATAL_ERROR_INTERNAL);
```

```
77      return 0;   // This is just to keep the compiler from complaining
78  }
```

B.10.2.3.3. Miscellaneous Functions

RandomForRsa()

This function uses a special form of KDFa() to produces a pseudo random sequence. It's input is a structure that contains pointers to a pre-computed set of hash contexts that are set up for the HMAC computations using the seed.

This function will test that ktx. outer will not wrap to zero if incremented. If so, the function returns FALSE. Otherwise, the ktx. outer is incremented before each number is generated.

```
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 }
```

MillerRabinRounds()

Function returns the number of MillerRabin() rounds necessary to give an error probability equal to the security strength of the prime. These values are from FIPS 186-3.

```
139 UINT32
140 MillerRabinRounds(
141     UINT32      bits        // IN: Number of bits in the RSA prime
142     )
143 {
144     if(bits < 511) <K>return 8;     // don't really expect this
145     if(bits < 1536) <K>return 5;    // for 512 and 1K primes
146     return 4;                     // for 3K public modulus and greater
147 }
```

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 }
```

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 }
```

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 }
```

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 }
```

SetDefaultNumberOfPrimes()

This macro sets the default number of primes to the indicated value.

```
294 //%#define SetDefaultNumberOfPrimes(p) (primeTableBytes = AdjustNumberOfPrimes(p))
```

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];
```

 Copyright © TCG 2006-2013

DRAFT

```
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(b);
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}};
```

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     }
```

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
522             // Set this as the trial prime
523             BN_add_word(bnP, chosen * 2);
524
525             // Use MR to see if this is prime
526             if(MillerRabin(bnP, rounds, ktx, context))
527             {
528                 // Final check is to make sure that 0 != (p-1) mod e
529                 // This is the same as -1 != p mod e ; or
530                 // (e - 1) != p mod e
531                 if((e <= 3) || (BN_mod_word(bnP, e) != (e-1)))
532                     return TRUE;
533             }
534             // Back out the bit number
535             BN_sub_word(bnP, chosen * 2);
536
537             // Clear the bit just tested
538             ClearBit(field, chosen);
539     }
540         // Ran out of bits and couldn't find a prime in this field
541         INSTRUMENT_INC(noPrimeFields);
542         return FALSE;
543     }
```

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.

```
544     void
545     AdjustPrimeCandidate(
546        BYTE         *a,
547        UINT16        len
548        )
549     {
550         UINT16  highBytes;
551
552         highBytes = BYTE_ARRAY_TO_UINT16(a);
553         // This is fixed point arithmetic on 16-bit values
554         highBytes = ((UINT32)highBytes * (UINT32)0x4AFB) >> 16;
555         highBytes += 0xB505;
556         UINT16_TO_BYTE_ARRAY(highBytes, a);
557         a[len-1] |= 1;
558     }
```

GeneratateRamdomPrime()

 Copyright © TCG 2006-2013

```
559  void
560  GenerateRandomPrime(
561      TPM2B   *p,
562      BN_CTX  *ctx
563  #ifdef  RSA_DEBUG       //%
564     ,UINT16   field,
565      UINT16   primes
566  #endif                  //%
567      )
568  {
569      BIGNUM  *bnP;
570      BN_CTX  *context;
571
572      if(ctx == NULL) context = BN_CTX_new();
573      else context = ctx;
574      if(context == NULL)
575          FAIL(FATAL_ERROR_ALLOCATION);
576      BN_CTX_start(context);
577      bnP = BN_CTX_get(context);
578
579      while(TRUE)
580      {
581          _cpri__GenerateRandom(p->size, p->buffer);
582          p->buffer[p->size-1] |= 1;
583          p->buffer[0] |= 0x80;
584          BN_bin2bn(p->buffer, p->size, bnP);
585  #ifdef  RSA_DEBUG
586          if(PrimeSelectWithSieve(bnP, NULL, 0, context, field, primes))
587  #else
588          if(PrimeSelectWithSieve(bnP, NULL, 0, context))
589  #endif
590              break;
591      }
592      BnTo2B(p, bnP, (INT16)BN_num_bytes(bnP));
593      BN_CTX_end(context);
594      if(ctx == NULL)
595          BN_CTX_free(context);
596      return;
597  }
598  KDFa_CONTEXT *
599  KDFaContextStart(
600      KDFa_CONTEXT        *ktx,           // IN/OUT: the context structure to
601                                          //         initialize
602      TPM2B               *seed,          // IN: the seed for the digest process
603      TPM_ALG_ID           hashAlg,       // IN: the hash algorithm
604      TPM2B               *extra,         // IN: the extra data
605      UINT32              *outer,         // IN: the outer iteration counter
606      UINT16               keySizeInBits
607      )
608  {
609      UINT16               digestSize = _cpri__GetDigestSize(hashAlg);
610      TPM2B_HASH_BLOCK     oPadKey;
611
612      if(seed == NULL)
613          return NULL;
614
615      pAssert(ktx != NULL && outer != NULL && digestSize != 0);
616
617     // Start the hash using the seed and get the intermediate hash value
618     _cpri__StartHMAC(hashAlg, FALSE, &(ktx->iPadCtx), seed->size, seed->buffer,
619                      &oPadKey.b);
620     _cpri__StartHash(hashAlg, FALSE, &(ktx->oPadCtx));
621     _cpri__UpdateHash(&(ktx->oPadCtx), oPadKey.b.size, oPadKey.b.buffer);
622     ktx->extra = extra;
623     ktx->hashAlg = hashAlg;
624     ktx->outer = outer;
```

```
625     ktx->keySizeInBits = keySizeInBits;
626     return ktx;
627 }
628 void
629 KDFaContextEnd(
630     KDFa_CONTEXT        *ktx            // IN/OUT: the context structure to close
631     )
632 {
633     if(ktx != NULL)
634     {
635         // Close out the hash sessions
636         _cpri__CompleteHash(&(ktx->iPadCtx), 0, NULL);
637         _cpri__CompleteHash(&(ktx->oPadCtx), 0, NULL);
638     }
639 }
640 //%#endif
```

B.10.2.3.4. Public Function

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.

_cpri__GenerateKeyRSA()

Generate an RSA key from a provided seed

Return Value	Meaning
CRYPT_FAIL	exponent is not prime or is less than 3; or could not find a prime using the provided parameters
CRYPT_CANCEL	operation was cancelled

```
641 CRYPT_RESULT
642 _cpri__GenerateKeyRSA(
643     TPM2B           *n,             // OUT: The public modulus
644     TPM2B           *p,             // OUT: One of the prime factors of n
645     UINT16           keySizeInBits, // IN: Size of the public modulus in bits
646     UINT32           e,             // IN: The public exponent
647     TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use in the key
648                                     //     generation process
649     TPM2B           *seed,          // IN: the seed to use
650     const char      *label,         // IN: A label for the generation process.
651     TPM2B           *extra,         // IN: Party 1 data for the KDF
652     UINT32          *counter        // IN/OUT: Counter value to allow KDF
653                                     //         iteration to be propagated across
654                                     //         multiple routines
655 #ifdef  RSA_DEBUG                   //%
656    ,UINT16           primes,        // IN: number of primes to test
657     UINT16           fieldSize      // IN: the field size to use
658 #endif                              //%
659     )
660 {
661     CRYPT_RESULT         retVal;
662     UINT32               myCounter = 0;
663     UINT32              *pCtr = (counter == NULL) ? &myCounter : counter;
664
665     KDFa_CONTEXT         ktx;
666     KDFa_CONTEXT        *ktxPtr;
667     UINT32               i;
668     BIGNUM              *bnP;
669     BIGNUM              *bnQ;
```

```
670         BIGNUM              *bnT;
671         BIGNUM              *bnE;
672         BIGNUM              *bnN;
673         BN_CTX              *context;
674
675
676         // Make sure that the required pointers are provided
677         pAssert(n != NULL && p != NULL);
678
679         // If the seed is provided, then use KDFa for generation of the 'random'
680         // values
681         ktxPtr = KDFaContextStart(&ktx, seed, hashAlg, extra, pCtr, keySizeInBits);
682
683         n->size = keySizeInBits/8;
684         p->size = n->size / 2;
685
686         // Validate exponent
687         if(e == 0 || e == RSA_DEFAULT_PUBLIC_EXPONENT)
688             e = RSA_DEFAULT_PUBLIC_EXPONENT;
689         else
690             if(!IsPrimeWord(e))
691                 return CRYPT_FAIL;
692
693         // Get structures for the big number representations
694         context = BN_CTX_new();
695         BN_CTX_start(context);
696         bnP = BN_CTX_get(context);
697         bnQ = BN_CTX_get(context);
698         bnT = BN_CTX_get(context);
699         bnE = BN_CTX_get(context);
700         bnN = BN_CTX_get(context);
701         if(bnN == NULL)
702             FAIL(FATAL_ERROR_INTERNAL);
703
704         // Set Q to zero. This is used as a flag. The prime is computed in P. When a
705         // new prime is found, Q is checked to see if it is zero.  If so, P is copied
706         // to Q and a new P is found.  When both P and Q are non-zero, the modulus and
707         // private exponent are computed and a trial encryption/decryption is
708         // performed.  If the encrypt/decrypt fails, assume that at least one of the
709         // primes is composite. Since we don't know which one, set Q to zero and start
710         // over and find a new pair of primes.
711         BN_zero(bnQ);
712         BN_set_word(bnE, e);
713
714         // Each call to generate a random value will increment ktx.outer
715         // it doesn't matter if ktx.outer wraps. This lets the caller
716         // use the initial value of the counter for additional entropy.
717         for(i = 0; i < UINT32_MAX; i++)
718         {
719             if(_plat__IsCanceled())
720             {
721                 retVal = CRYPT_CANCEL;
722                 goto end;
723             }
724             // Get a random prime candidate.
725             if(seed == NULL)
726                 _cpri__GenerateRandom(p->size, p->buffer);
727             else
728                 RandomForRsa(&ktx, label, p);
729             AdjustPrimeCandidate(p->buffer, p->size);
730
731             // Convert the candidate to a BN
732             if(BN_bin2bn(p->buffer, p->size, bnP) == NULL)
733                 FAIL(FATAL_ERROR_INTERNAL);
734             // If this is the second prime, make sure that it differs from the
735             // first prime by at least 2^100. Since BIGNUMS use words, the check
```

```
736             // below will make sure they are different by at least 128 bits
737             if(!BN_is_zero(bnQ))
738             { // bnQ is non-zero, we have a first value
739                 UINT32       *pP = (UINT32 *)(&bnP->d[4]);
740                 UINT32       *pQ = (UINT32 *)(&bnQ->d[4]);
741                 INT32        k = ((INT32)bnP->top) - 4;
742                 for(;k > 0; k--)
743                     if(*pP++ != *pQ++)
744                         break;
745                 // Didn't find any difference so go get a new value
746                 if(k == 0)
747                     continue;
748             }
749             // If PrimeSelectWithSieve   returns success, bnP is a prime,
750     #ifdef  RSA_DEBUG
751             if(!PrimeSelectWithSieve(bnP, ktxPtr, e, context, fieldSize, primes))
752     #else
753             if(!PrimeSelectWithSieve(bnP, ktxPtr, e, context))
754     #endif
755                 continue;   // If not, get another
756
757             // Found a prime, is this the first or second.
758             if(BN_is_zero(bnQ))
759             {   // copy p to q and compute another prime in p
760                 BN_copy(bnQ, bnP);
761                 continue;
762             }
763             //Form the public modulus
764            if(   BN_mul(bnN, bnP, bnQ, context) != 1
765               || BN_num_bits(bnN) != keySizeInBits)
766                 FAIL(FATAL_ERROR_INTERNAL);
767             // Save the public modulus
768             BnTo2B(n, bnN, n->size);
769             // And one prime
770             BnTo2B(p, bnP, p->size);
771
772     #ifdef EXTENDED_CHECKS
773             // Finish by making sure that we can form the modular inverse of PHI
774             // with respect to the public exponent
775             // Compute PHI = (p - 1)(q - 1) = n - p - q + 1
776             // Make sure that we can form the modular inverse
777             if(   BN_sub(bnT, bnN, bnP) != 1
778                || BN_sub(bnT, bnT, bnQ) != 1
779                || BN_add_word(bnT, 1) != 1)
780                 FAIL(FATAL_ERROR_INTERNAL);
781
782             // find d such that (Phi * d) mod e ==1
783             // If there isn't then we are broken because we took the step
784             // of making sure that the prime != 1 mod e so the modular inverse
785             // must exist
786             if(   BN_mod_inverse(bnT, bnE, bnT, context) == NULL
787                || BN_is_zero(bnT))
788                 FAIL(FATAL_ERROR_INTERNAL);
789
790             // And, finally, do a trial encryption decryption
791             {
792                 TPM2B_TYPE(RSA_KEY, MAX_RSA_KEY_BYTES);
793                 TPM2B_RSA_KEY        r;
794                 r.t.size = sizeof(r.t.buffer);
795                 // If we are using a seed, then results must be reproducible on each
796                 // call. Otherwise, just get a random number
797                 if(seed == NULL)
798                     _cpri__GenerateRandom(keySizeInBits/8, r.t.buffer);
799                 else
800                     RandomForRsa(&ktx, label, &r.b);
801
```

```
802             // Make sure that the number is smaller than the public modulus
803             r.t.buffer[0] &= 0x7F;
804                   // Convert
805             if(   BN_bin2bn(r.t.buffer, r.t.size, bnP) == NULL
806                   // Encrypt with the public exponent
807                || BN_mod_exp(bnQ, bnP, bnE, bnN, context) != 1
808                   // Decrypt with the private exponent
809                || BN_mod_exp(bnQ, bnQ, bnT, bnN, context) != 1)
810                 FAIL(FATAL_ERROR_INTERNAL);
811             // If the starting and ending values are not the same, start over )-;
812             if(BN_ucmp(bnP, bnQ) != 0)
813             {
814                 BN_zero(bnQ);
815                 continue;
816             }
817         }
818 #endif // EXTENDED_CHECKS
819         retVal = CRYPT_SUCCESS;
820         goto end;
821     }
822     retVal = CRYPT_FAIL;
823
824 end:
825     KDFaContextEnd(&ktx);
826
827     // Free up allocated BN values
828     BN_CTX_end(context);
829     BN_CTX_free(context);
830     return retVal;
831 }
832 #else
833 static void noFuntion(void)
834 {
835     pAssert(1);
836 }
837 #endif              //%
```

B.10.2.4. 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
};
```

DRAFT

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,
```

```
180       0x04,0x1E,0x02,0x0A,0x02,0x12,0x04,0x12,0x06,0x0E,0x0A,0x02,0x04,0x14,0x24,0x06,
181       0x04,0x06,0x0E,0x04,0x14,0x0A,0x0E,0x16,0x06,0x02,0x1E,0x0C,0x0A,0x12,0x02,0x04,
182       0x0E,0x06,0x16,0x12,0x02,0x0C,0x06,0x04,0x08,0x04,0x08,0x06,0x0A,0x02,0x0C,0x12,
183       0x0A,0x0E,0x10,0x0E,0x04,0x06,0x06,0x02,0x06,0x04,0x02,0x1C,0x02,0x1C,0x06,0x02,
184       0x04,0x06,0x0E,0x04,0x0C,0x0E,0x10,0x0E,0x04,0x06,0x08,0x06,0x04,0x06,0x06,0x06,
185       0x08,0x04,0x08,0x04,0x0E,0x10,0x08,0x06,0x04,0x0C,0x08,0x10,0x02,0x0A,0x08,0x04,
186       0x06,0x1A,0x06,0x0A,0x08,0x04,0x06,0x0C,0x0E,0x1E,0x04,0x0E,0x16,0x08,0x0C,0x04,
187       0x06,0x08,0x0A,0x06,0x0E,0x0A,0x06,0x02,0x0A,0x0C,0x0C,0x0E,0x06,0x06,0x12,0x0A,
188       0x06,0x08,0x12,0x04,0x06,0x02,0x06,0x0A,0x02,0x0A,0x08,0x06,0x06,0x0A,0x02,0x12
189   #endif
190   // 1792
191   #if PRIME_DIFF_TABLE_BYTES > 1792
192     ,0x0A,0x02,0x0C,0x04,0x06,0x08,0x0A,0x0C,0x0E,0x0C,0x04,0x08,0x0A,0x06,0x06,0x14,
193       0x04,0x0E,0x10,0x0E,0x0A,0x08,0x0A,0x0C,0x02,0x12,0x06,0x0C,0x0A,0x0C,0x02,0x04,
194       0x02,0x0C,0x06,0x04,0x08,0x04,0x2C,0x04,0x02,0x04,0x02,0x0A,0x0C,0x06,0x06,0x0E,
195       0x04,0x06,0x06,0x06,0x08,0x06,0x24,0x12,0x04,0x06,0x02,0x0C,0x06,0x06,0x06,0x04,
196       0x0E,0x16,0x0C,0x02,0x12,0x0A,0x06,0x1A,0x18,0x04,0x02,0x04,0x02,0x04,0x0E,0x04,
197       0x06,0x06,0x08,0x10,0x0C,0x02,0x2A,0x04,0x02,0x04,0x18,0x06,0x06,0x02,0x12,0x04,
198       0x0E,0x06,0x1C,0x12,0x0E,0x06,0x0A,0x0C,0x02,0x06,0x0C,0x1E,0x06,0x04,0x06,0x06,
199       0x0E,0x04,0x02,0x18,0x04,0x06,0x06,0x1A,0x0A,0x12,0x06,0x08,0x06,0x06,0x1E,0x04,
200       0x0C,0x0C,0x02,0x10,0x02,0x06,0x04,0x0C,0x12,0x02,0x06,0x04,0x1A,0x0C,0x06,0x0C,
201       0x04,0x18,0x18,0x0C,0x06,0x02,0x0C,0x1C,0x08,0x04,0x06,0x0C,0x02,0x12,0x06,0x04,
202       0x06,0x06,0x14,0x10,0x02,0x06,0x06,0x12,0x0A,0x06,0x02,0x04,0x08,0x06,0x06,0x18,
203       0x10,0x06,0x08,0x0A,0x06,0x0E,0x16,0x08,0x10,0x06,0x02,0x0C,0x04,0x02,0x16,0x08,
204       0x12,0x22,0x02,0x06,0x12,0x04,0x06,0x06,0x08,0x0A,0x08,0x12,0x06,0x04,0x02,0x04,
205       0x08,0x10,0x02,0x0C,0x0C,0x06,0x12,0x04,0x06,0x06,0x06,0x02,0x06,0x0C,0x0A,0x14,
206       0x0C,0x12,0x04,0x06,0x02,0x10,0x02,0x0A,0x0E,0x04,0x1E,0x02,0x0A,0x0C,0x02,0x18,
207       0x06,0x10,0x08,0x0A,0x02,0x0C,0x16,0x06,0x02,0x10,0x14,0x0A,0x02,0x0C,0x0C,0x00
208   #endif
209   // 2048
210   #if PRIME_DIFF_TABLE_BYTES > 2048
211     ,0x12,0x0A,0x0C,0x06,0x02,0x0A,0x02,0x06,0x0A,0x12,0x02,0x0C,0x06,0x04,0x06,0x02,
212       0x18,0x1C,0x02,0x04,0x02,0x0A,0x02,0x10,0x0C,0x08,0x16,0x02,0x06,0x04,0x02,0x0A,
213       0x06,0x14,0x0C,0x0A,0x08,0x0C,0x06,0x06,0x06,0x04,0x12,0x02,0x04,0x0C,0x12,0x02,
214       0x0C,0x06,0x04,0x02,0x10,0x0C,0x0C,0x0E,0x04,0x08,0x12,0x04,0x0C,0x0E,0x06,0x06,
215       0x04,0x08,0x06,0x04,0x14,0x0C,0x0A,0x0E,0x04,0x02,0x10,0x02,0x0C,0x1E,0x04,0x06,
216       0x18,0x14,0x18,0x0A,0x08,0x0C,0x0A,0x0C,0x06,0x0C,0x0C,0x06,0x08,0x10,0x0E,0x06,
217       0x04,0x06,0x24,0x14,0x0A,0x1E,0x0C,0x02,0x04,0x02,0x1C,0x0C,0x0E,0x06,0x16,0x08,
218       0x04,0x12,0x06,0x0E,0x12,0x04,0x06,0x02,0x06,0x22,0x12,0x02,0x10,0x06,0x12,0x02,
219       0x18,0x04,0x02,0x06,0x0C,0x06,0x0C,0x0A,0x08,0x06,0x10,0x0C,0x08,0x0A,0x0E,0x28,
220       0x06,0x02,0x06,0x04,0x0C,0x0E,0x04,0x02,0x04,0x02,0x04,0x08,0x06,0x0A,0x06,0x06,
221       0x02,0x06,0x06,0x06,0x0C,0x06,0x18,0x0A,0x02,0x0A,0x06,0x0C,0x06,0x06,0x0E,0x06,
222       0x06,0x34,0x14,0x06,0x0A,0x02,0x0A,0x08,0x0A,0x0C,0x0C,0x02,0x06,0x04,0x0E,0x10,
223       0x08,0x0C,0x06,0x16,0x02,0x0A,0x08,0x06,0x16,0x02,0x16,0x06,0x08,0x0A,0x0C,0x0C,
224       0x02,0x0A,0x06,0x0C,0x02,0x04,0x0E,0x0A,0x02,0x06,0x12,0x04,0x0C,0x08,0x12,0x0C,
225       0x06,0x06,0x04,0x06,0x06,0x0E,0x04,0x02,0x0C,0x0C,0x04,0x06,0x12,0x12,0x0C,0x02,
226       0x10,0x0C,0x08,0x12,0x0A,0x1A,0x04,0x06,0x08,0x06,0x06,0x04,0x02,0x0A,0x14,0x04
227   #endif
228   // 2304
229   #if PRIME_DIFF_TABLE_BYTES > 2304
230     ,0x06,0x08,0x04,0x14,0x0A,0x02,0x22,0x02,0x04,0x18,0x02,0x0C,0x0C,0x0A,0x06,0x02,
231       0x0C,0x1E,0x06,0x0C,0x10,0x0C,0x02,0x16,0x12,0x0C,0x0E,0x0A,0x02,0x0C,0x0C,0x04,
232       0x02,0x04,0x06,0x0C,0x02,0x10,0x12,0x02,0x28,0x08,0x10,0x06,0x08,0x0A,0x02,0x04,
233       0x12,0x08,0x0A,0x08,0x0C,0x04,0x12,0x02,0x12,0x0A,0x02,0x04,0x02,0x04,0x08,0x1C,
234       0x02,0x06,0x16,0x0C,0x06,0x0E,0x12,0x04,0x06,0x08,0x06,0x06,0x0A,0x08,0x04,0x02,
235       0x12,0x0A,0x06,0x14,0x16,0x08,0x06,0x1E,0x04,0x02,0x04,0x12,0x06,0x1E,0x02,0x04,
236       0x08,0x06,0x04,0x06,0x0C,0x0E,0x22,0x0E,0x06,0x04,0x02,0x06,0x04,0x0E,0x04,0x02,
237       0x06,0x1C,0x02,0x04,0x06,0x08,0x0A,0x02,0x0A,0x02,0x0A,0x02,0x04,0x1E,0x02,0x0C,
238       0x0C,0x0A,0x12,0x0C,0x0E,0x0A,0x02,0x0C,0x06,0x0A,0x06,0x0E,0x0C,0x04,0x0E,0x04,
239       0x12,0x02,0x0A,0x08,0x04,0x08,0x0A,0x0C,0x12,0x12,0x08,0x06,0x12,0x10,0x0E,0x06,
240       0x06,0x0A,0x0E,0x04,0x06,0x02,0x0C,0x0C,0x04,0x06,0x06,0x0C,0x02,0x10,0x02,0x0C,
241       0x06,0x04,0x0E,0x06,0x04,0x02,0x0C,0x12,0x04,0x24,0x12,0x0C,0x0C,0x02,0x04,0x02,
242       0x04,0x08,0x0C,0x04,0x24,0x06,0x12,0x02,0x0C,0x0A,0x06,0x0C,0x18,0x08,0x06,0x06,
243       0x10,0x0C,0x02,0x12,0x0A,0x14,0x0A,0x02,0x06,0x12,0x04,0x02,0x28,0x06,0x02,0x10,
244       0x02,0x04,0x08,0x12,0x0A,0x0C,0x06,0x02,0x0A,0x08,0x04,0x06,0x0C,0x02,0x0A,0x12,
245       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,
```

```
    0x0E,0x0C,0x0C,0x10,0x06,0x06,0x02,0x12,0x04,0x06,0x1E,0x0E,0x04,0x06,0x06,0x02
#endif
// 4352
#if PRIME_DIFF_TABLE_BYTES > 4352
  ,0x06,0x04,0x06,0x0E,0x06,0x04,0x08,0x0A,0x0C,0x06,0x20,0x0A,0x08,0x16,0x02,0x0A,
   0x06,0x18,0x08,0x04,0x1E,0x06,0x02,0x0C,0x10,0x08,0x06,0x04,0x06,0x08,0x10,0x0E,
   0x06,0x06,0x04,0x02,0x0A,0x0C,0x02,0x10,0x0E,0x04,0x02,0x04,0x14,0x12,0x0A,0x02,
   0x0A,0x06,0x0C,0x1E,0x08,0x12,0x0C,0x0A,0x02,0x06,0x06,0x04,0x0C,0x0C,0x02,0x04,
   0x0C,0x12,0x18,0x02,0x0A,0x06,0x08,0x10,0x08,0x06,0x0C,0x0A,0x0E,0x06,0x0C,0x06,
   0x06,0x04,0x02,0x18,0x04,0x06,0x08,0x06,0x04,0x02,0x04,0x06,0x0E,0x04,0x08,0x0A,
   0x18,0x18,0x0C,0x02,0x06,0x0C,0x16,0x1E,0x02,0x06,0x12,0x0A,0x06,0x06,0x08,0x04,
   0x02,0x06,0x0A,0x08,0x0A,0x06,0x08,0x10,0x06,0x0E,0x06,0x04,0x18,0x08,0x0A,0x02,
   0x0C,0x06,0x04,0x24,0x02,0x16,0x06,0x08,0x06,0x0A,0x08,0x06,0x0C,0x0A,0x0E,0x0A,
   0x06,0x12,0x0C,0x02,0x0C,0x04,0x1A,0x0A,0x0E,0x10,0x12,0x08,0x12,0x0C,0x0C,0x06,
   0x10,0x0E,0x18,0x0A,0x0C,0x08,0x16,0x06,0x02,0x0A,0x3C,0x06,0x02,0x04,0x08,0x10,
   0x0E,0x0A,0x06,0x18,0x06,0x0C,0x12,0x18,0x02,0x1E,0x04,0x02,0x0C,0x06,0x0A,0x02,
   0x04,0x0E,0x06,0x10,0x02,0x0A,0x08,0x16,0x14,0x06,0x04,0x20,0x06,0x12,0x04,0x02,
   0x04,0x02,0x04,0x08,0x34,0x0E,0x16,0x02,0x16,0x14,0x0A,0x08,0x0A,0x02,0x06,0x04,
   0x0E,0x04,0x06,0x14,0x04,0x06,0x02,0x0C,0x0C,0x06,0x0C,0x10,0x02,0x0C,0x0A,0x08,
   0x04,0x06,0x02,0x1C,0x0C,0x08,0x0A,0x0C,0x02,0x04,0x0E,0x1C,0x08,0x06,0x04,0x02
#endif
// 4608
#if PRIME_DIFF_TABLE_BYTES > 4608
  ,0x04,0x06,0x02,0x0C,0x3A,0x06,0x0E,0x0A,0x02,0x06,0x1C,0x20,0x04,0x1E,0x08,0x06,
   0x04,0x06,0x0C,0x0C,0x02,0x04,0x06,0x06,0x0E,0x10,0x08,0x1E,0x04,0x02,0x0A,0x08,
   0x06,0x04,0x06,0x1A,0x04,0x0C,0x02,0x0A,0x12,0x0C,0x0C,0x12,0x02,0x04,0x0C,0x08,
   0x0C,0x0A,0x14,0x04,0x08,0x10,0x0C,0x08,0x06,0x10,0x08,0x0A,0x0C,0x0E,0x06,0x04,
   0x08,0x0C,0x04,0x14,0x06,0x28,0x08,0x10,0x06,0x24,0x02,0x06,0x04,0x06,0x02,0x16,
   0x12,0x02,0x0A,0x06,0x24,0x0E,0x0C,0x04,0x12,0x08,0x04,0x0E,0x0A,0x02,0x0A,0x08,
   0x04,0x02,0x12,0x10,0x0C,0x0E,0x0A,0x0E,0x06,0x06,0x2A,0x0A,0x06,0x06,0x14,0x0A,
   0x08,0x0C,0x04,0x0C,0x12,0x02,0x0A,0x0E,0x12,0x0A,0x12,0x08,0x06,0x04,0x0E,0x06,
   0x0A,0x1E,0x0E,0x06,0x06,0x04,0x0C,0x26,0x04,0x02,0x04,0x06,0x08,0x0C,0x0A,0x06,
   0x12,0x06,0x32,0x06,0x04,0x06,0x0C,0x08,0x0A,0x20,0x06,0x16,0x02,0x0A,0x0C,0x12,
   0x02,0x06,0x04,0x1E,0x08,0x06,0x06,0x12,0x0A,0x02,0x04,0x0C,0x14,0x0A,0x08,0x18,
   0x0A,0x02,0x06,0x16,0x06,0x02,0x12,0x0A,0x0C,0x02,0x1E,0x12,0x0C,0x1C,0x02,0x06,
   0x04,0x06,0x0E,0x06,0x0C,0x0A,0x08,0x04,0x0C,0x1A,0x0A,0x08,0x06,0x10,0x02,0x0A,
   0x12,0x0E,0x06,0x04,0x06,0x0E,0x10,0x02,0x06,0x04,0x0C,0x14,0x04,0x14,0x04,0x06,
   0x0C,0x02,0x24,0x04,0x06,0x02,0x0A,0x02,0x16,0x08,0x06,0x0A,0x0C,0x0C,0x12,0x0E,
   0x18,0x24,0x04,0x14,0x18,0x0A,0x06,0x02,0x1C,0x06,0x12,0x08,0x04,0x06,0x08,0x06
#endif
// 4864
#if PRIME_DIFF_TABLE_BYTES > 4864
  ,0x04,0x02,0x0C,0x1C,0x12,0x0E,0x10,0x0E,0x12,0x0A,0x08,0x06,0x04,0x06,0x06,0x08,
   0x16,0x0C,0x02,0x0A,0x12,0x06,0x02,0x12,0x0A,0x02,0x0C,0x0A,0x12,0x20,0x06,0x04,
   0x06,0x06,0x08,0x06,0x06,0x0A,0x14,0x06,0x0C,0x0A,0x08,0x0A,0x0E,0x06,0x0A,0x0E,
   0x04,0x02,0x16,0x12,0x02,0x0A,0x02,0x04,0x14,0x04,0x02,0x22,0x02,0x0C,0x06,0x0A,
   0x02,0x0A,0x12,0x06,0x0E,0x0C,0x0C,0x16,0x08,0x06,0x10,0x06,0x08,0x04,0x0C,0x06,
   0x08,0x04,0x24,0x06,0x06,0x14,0x18,0x06,0x0C,0x12,0x0A,0x02,0x0A,0x1A,0x06,0x10,
   0x08,0x06,0x04,0x18,0x12,0x08,0x0C,0x0C,0x0A,0x12,0x0C,0x02,0x18,0x04,0x0C,0x12,
   0x0C,0x0E,0x0A,0x02,0x04,0x18,0x0C,0x0E,0x0A,0x06,0x02,0x06,0x04,0x06,0x1A,0x04,
   0x06,0x06,0x02,0x16,0x08,0x12,0x04,0x12,0x08,0x04,0x18,0x02,0x0C,0x0C,0x04,0x02,
   0x34,0x02,0x12,0x06,0x04,0x06,0x0C,0x02,0x06,0x0C,0x0A,0x08,0x04,0x02,0x18,0x0A,
   0x02,0x0A,0x02,0x0C,0x06,0x12,0x28,0x06,0x14,0x10,0x02,0x0C,0x06,0x0A,0x0C,0x02,
   0x04,0x06,0x0E,0x0C,0x0C,0x16,0x06,0x08,0x04,0x02,0x10,0x12,0x0C,0x02,0x06,0x10,
   0x06,0x02,0x06,0x04,0x0C,0x1E,0x08,0x10,0x02,0x12,0x0A,0x18,0x02,0x06,0x18,0x04,
   0x02,0x16,0x02,0x10,0x02,0x06,0x0C,0x04,0x12,0x08,0x04,0x0E,0x04,0x12,0x18,0x06,
   0x02,0x06,0x0A,0x02,0x0A,0x26,0x06,0x0A,0x0E,0x06,0x06,0x18,0x04,0x02,0x0C,0x10,
   0x0E,0x10,0x0C,0x02,0x06,0x0A,0x1A,0x04,0x02,0x0C,0x06,0x04,0x0C,0x08,0x0C,0x0A
#endif
// 5120
#if PRIME_DIFF_TABLE_BYTES > 5120
  ,0x12,0x06,0x0E,0x1C,0x02,0x06,0x0A,0x02,0x04,0x0E,0x22,0x02,0x06,0x16,0x02,0x0A,
   0x0E,0x04,0x02,0x10,0x08,0x0A,0x06,0x08,0x0A,0x08,0x04,0x06,0x02,0x10,0x06,0x06,
   0x12,0x1E,0x0E,0x06,0x04,0x1E,0x02,0x0A,0x0E,0x04,0x14,0x0A,0x08,0x04,0x08,0x12,
   0x04,0x0E,0x06,0x04,0x18,0x06,0x06,0x12,0x12,0x02,0x24,0x06,0x0A,0x0E,0x0C,0x04,
   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,
```

```
    1051,1061,1063,1069,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,1163,
    1171,1181,1187,1193,1201,1213,1217,1223,1229,1231,1237,1249,1259,1277,1279,1283,
    1289,1291,1297,1301,1303,1307,1319,1321,1327,1361,1367,1373,1381,1399,1409,1423,
    1427,1429,1433,1439,1447,1451,1453,1459,1471,1481,1483,1487,1489,1493,1499,1511,
    1523,1531,1543,1549,1553,1559,1567,1571,1579,1583,1597,1601,1607,1609,1613,1619,
    1621,1627,1637,1657,1663,1667,1669,1693,1697,1699,1709,1721,1723,1733,1741,1747,
    1753,1759,1777,1783,1787,1789,1801,1811,1823,1831,1847,1861,1867,1871,1873,1877,
    1879,1889,1901,1907,1913,1931,1933,1949,1951,1973,1979,1987,1993,1997,1999,2003,
    2011,2017,2027,2029,2039,2053,2063,2069,2081,2083,2087,2089,2099,2111,2113,2129,
    2131,2137,2141,2143,2153,2161,2179,2203,2207,2213,2221,2237,2239,2243,2251,2267,
    2269,2273,2281,2287,2293,2297,2309,2311,2333,2339,2341,2347,2351,2357,2371,2377,
    2381,2383,2389,2393,2399,2411,2417,2423,2437,2441,2447,2459,2467,2473,2477,2503,
    2521,2531,2539,2543,2549,2551,2557,2579,2591,2593,2609,2617,2621,2633,2647,2657,
    2659,2663,2671,2677,2683,2687,2689,2693,2699,2707,2711,2713,2719,2729,2731,2741,
    2749,2753,2767,2777,2789,2791,2797,2801,2803,2819,2833,2837,2843,2851,2857,2861,
    2879,2887,2897,2903,2909,2917,2927,2939,2953,2957,2963,2969,2971,2999,3001,3011,
    3019,3023,3037,3041,3049,3061,3067,3079,3083,3089,3109,3119,3121,3137,3163,3167,
    3169,3181,3187,3191,3203,3209,3217,3221,3229,3251,3253,3257,3259,3271,3299,3301,
    3307,3313,3319,3323,3329,3331,3343,3347,3359,3361,3371,3373,3389,3391,3407,3413,
    3433,3449,3457,3461,3463,3467,3469,3491,3499,3511,3517,3527,3529,3533,3539,3541,
    3547,3557,3559,3571,3581,3583,3593,3607,3613,3617,3623,3631,3637,3643,3659,3671,
    3673,3677,3691,3697,3701,3709,3719,3727,3733,3739,3761,3767,3769,3779,3793,3797,
    3803,3821,3823,3833,3847,3851,3853,3863,3877,3881,3889,3907,3911,3917,3919,3923,
    3929,3931,3943,3947,3967,3989,4001,4003,4007,4013,4019,4021,4027,4049,4051,4057,
    4073,4079,4091,4093,4099,4111,4127,4129,4133,4139,4153,4157,4159,4177,4201,4211,
    4217,4219,4229,4231,4241,4243,4253,4259,4261,4271,4273,4283,4289,4297,4327,4337,
    4339,4349,4357,4363,4373,4391,4397,4409,4421,4423,4441,4447,4451,4457,4463,4481,
    4483,4493,4507,4513,4517,4519,4523,4547,4549,4561,4567,4583,4591,4597,4603,4621,
    4637,4639,4643,4649,4651,4657,4663,4673,4679,4691,4703,4721,4723,4729,4733,4751,
    4759,4783,4787,4789,4793,4799,4801,4813,4817,4831,4861,4871,4877,4889,4903,4909,
    4919,4931,4933,4937,4943,4951,4957,4967,4969,4973,4987,4993,4999,5003,5009,5011,
    5021,5023,5039,5051,5059,5077,5081,5087,5099,5101,5107,5113,5119,5147,5153,5167,
    5171,5179,5189,5197,5209,5227,5231,5233,5237,5261,5273,5279,5281,5297,5303,5309,
    5323,5333,5347,5351,5381,5387,5393,5399,5407,5413,5417,5419,5431,5437,5441,5443,
    5449,5471,5477,5479,5483,5501,5503,5507,5519,5521,5527,5531,5557,5563,5569,5573,
    5581,5591,5623,5639,5641,5647,5651,5653,5657,5659,5669,5683,5689,5693,5701,5711,
    5717,5737,5741,5743,5749,5779,5783,5791,5801,5807,5813,5821,5827,5839,5843,5849,
    5851,5857,5861,5867,5869,5879,5881,5897,5903,5923,5927,5939,5953,5981,5987,6007,
    6011,6029,6037,6043,6047,6053,6067,6073,6079,6089,6091,6101,6113,6121,6131,6133,
    6143,6151,6163,6173,6197,6199,6203,6211,6217,6221,6229,6247,6257,6263,6269,6271,
    6277,6287,6299,6301,6311,6317,6323,6329,6337,6343,6353,6359,6361,6367,6373,6379,
    6389,6397,6421,6427,6449,6451,6469,6473,6481,6491,6521,6529,6547,6551,6553,6563,
    6569,6571,6577,6581,6599,6607,6619,6637,6653,6659,6661,6673,6679,6689,6691,6701,
    6703,6709,6719,6733,6737,6761,6763,6779,6781,6791,6793,6803,6823,6827,6829,6833,
    6841,6857,6863,6869,6871,6883,6899,6907,6911,6917,6947,6949,6959,6961,6967,6971,
    6977,6983,6991,6997,7001,7013,7019,7027,7039,7043,7057,7069,7079,7103,7109,7121,
    7127,7129,7151,7159,7177,7187,7193,7207,7211,7213,7219,7229,7237,7243,7247,7253,
    7283,7297,7307,7309,7321,7331,7333,7349,7351,7369,7393,7411,7417,7433,7451,7457,
    7459,7477,7481,7487,7489,7499,7507,7517,7523,7529,7537,7541,7547,7549,7559,7561,
    7573,7577,7583,7589,7591,7603,7607,7621,7639,7643,7649,7669,7673,7681,7687,7691,
    7699,7703,7717,7723,7727,7741,7753,7757,7759,7789,7793,7817,7823,7829,7841,7853,
    7867,7873,7877,7879,7883,7901,7907,7919,7927,7933,7937,7949,7951,7963,7993,8009,
    8011,8017,8039,8053,8059,8069,8081,8087,8089,8093,8101,8111,8117,8123,8147,8161,
    8167,8171,8179,8191,8209,8219,8221,8231,8233,8237,8243,8263,8269,8273,8287,8291,
    8293,8297,8311,8317,8329,8353,8363,8369,8377,8387,8389,8419,8423,8429,8431,8443,
    8447,8461,8467,8501,8513,8521,8527,8537,8539,8543,8563,8573,8581,8597,8599,8609,
    8623,8627,8629,8641,8647,8663,8669,8677,8681,8689,8693,8699,8707,8713,8719,8731,
    8737,8741,8747,8753,8761,8779,8783,8803,8807,8819,8821,8831,8837,8839,8849,8861,
    8863,8867,8887,8893,8923,8929,8933,8941,8951,8963,8969,8971,8999,9001,9007,9011,
    9013,9029,9041,9043,9049,9059,9067,9091,9103,9109,9127,9133,9137,9151,9157,9161,
    9173,9181,9187,9199,9203,9209,9221,9227,9239,9241,9257,9277,9281,9283,9293,9311,
    9319,9323,9337,9341,9343,9349,9371,9377,9391,9397,9403,9413,9419,9421,9431,9433,
    9437,9439,9461,9463,9467,9473,9479,9491,9497,9511,9521,9533,9539,9547,9551,9587,
    9601,9613,9619,9623,9629,9631,9643,9649,9661,9677,9679,9689,9697,9719,9721,9733,
    9739,9743,9749,9767,9769,9781,9787,9791,9803,9811,9817,9829,9833,9839,9851,9857,
    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.11 Elliptic Curve Files

B.11.1. 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
```

DRAFT

B.11.2. CpriDataEcc.c

B.11.2.1.1. Introduction

The curve parameters in this section replicate the information that is in the TCG Algorithm Registry. This curve data should be removed when the data in the registry is extracted into a data file (CryptDataEcc.c) and a header file (CryptDataEcc.h). The header file should be shared between CryptEcc.c and CyrptUtil.c

NOTE: This file should be included by the Ecc module in the library.

B.11.2.1.2. NIST Prime 256-bit Curve

```
1   static const TPM2B_32_BYTE_VALUE NIST_P256_P = {32,{
2                                       0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x01,
3                                       0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
4                                       0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
5                                       0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};
6   static const TPM2B_32_BYTE_VALUE NIST_P256_A = {32,{
7                                       0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x01,
8                                       0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
9                                       0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
10                                      0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xfc}};
11  static const TPM2B_32_BYTE_VALUE NIST_P256_B = {32,{
12                                      0x5a,0xc6,0x35,0xd8,0xaa,0x3a,0x93,0xe7,
13                                      0xb3,0xeb,0xbd,0x55,0x76,0x98,0x86,0xbc,
14                                      0x65,0x1d,0x06,0xb0,0xcc,0x53,0xb0,0xf6,
15                                      0x3b,0xce,0x3c,0x3e,0x27,0xd2,0x60,0x4b}};
16  static const TPM2B_32_BYTE_VALUE NIST_P256_X = {32,{
17                                      0x6b,0x17,0xd1,0xf2,0xe1,0x2c,0x42,0x47,
18                                      0xf8,0xbc,0xe6,0xe5,0x63,0xa4,0x40,0xf2,
19                                      0x77,0x03,0x7d,0x81,0x2d,0xeb,0x33,0xa0,
20                                      0xf4,0xa1,0x39,0x45,0xd8,0x98,0xc2,0x96}};
21  static const TPM2B_32_BYTE_VALUE NIST_P256_Y = {32,{
22                                      0x4f,0xe3,0x42,0xe2,0xfe,0x1a,0x7f,0x9b,
23                                      0x8e,0xe7,0xeb,0x4a,0x7c,0x0f,0x9e,0x16,
24                                      0x2b,0xce,0x33,0x57,0x6b,0x31,0x5e,0xce,
25                                      0xcb,0xb6,0x40,0x68,0x37,0xbf,0x51,0xf5}};
26  static const TPM2B_32_BYTE_VALUE NIST_P256_N = {32,{
27                                      0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
28                                      0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
29                                      0xbc,0xe6,0xfa,0xad,0xa7,0x17,0x9e,0x84,
30                                      0xf3,0xb9,0xca,0xc2,0xfc,0x63,0x25,0x51}};
31  static const TPM2B_1_BYTE_VALUE   NIST_P256_H = {1, {1}};
32  const ECC_CURVE_DATA NIST_P256 = {&NIST_P256_P.b, &NIST_P256_A.b, &NIST_P256_B.b,
33                                    &NIST_P256_X.b, &NIST_P256_Y.b, &NIST_P256_N.b,
34                                    &NIST_P256_H.b};
```

B.11.2.1.3. BN Prime 256-bit Curve

```
35  static const TPM2B_32_BYTE_VALUE BN_P256_P = {32,{
36                                      0xff,0xff,0xff,0xff,0xff,0xfc,0xf0,0xcd,
37                                      0x46,0xe5,0xf2,0x5e,0xee,0x71,0xa4,0x9f,
38                                      0x0c,0xdc,0x65,0xfb,0x12,0x98,0x0a,0x82,
39                                      0xd3,0x29,0x2d,0xdb,0xae,0xd3,0x30,0x13}};
40  static const TPM2B_1_BYTE_VALUE  BN_P256_A = {1,{0}};
41  static const TPM2B_1_BYTE_VALUE  BN_P256_B = {1,{3}};
42  static const TPM2B_1_BYTE_VALUE  BN_P256_X = {1,{1}};
43  static const TPM2B_1_BYTE_VALUE  BN_P256_Y = {1,{2}};
44  static const TPM2B_32_BYTE_VALUE BN_P256_N = {32,{
45                                      0xff,0xff,0xff,0xff,0xff,0xfc,0xf0,0xcd,
46                                      0x46,0xe5,0xf2,0x5e,0xee,0x71,0xa4,0x9e,
```

```
                                        0x0c,0xdc,0x65,0xfb,0x12,0x99,0x92,0x1a,
                                        0xf6,0x2d,0x53,0x6c,0xd1,0x0b,0x50,0x0d}};
static const TPM2B_1_BYTE_VALUE  BN_P256_H = {1,{1}};
const ECC_CURVE_DATA BN_P256 = {&BN_P256_P.b, &BN_P256_A.b, &BN_P256_B.b,
                                &BN_P256_X.b, &BN_P256_Y.b, &BN_P256_N.b,
                                &BN_P256_H.b};
#ifdef TPM_ECC_SM2_P256
#ifndef   _SM2_SIGN_DEBUG
```

These are the actual values for SM2 curve

```
static const TPM2B_32_BYTE_VALUE SM2_P256_P = {32,{
                                        0xff,0xff,0xff,0xfe,0xff,0xff,0xff,0xff,
                                        0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
                                        0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
                                        0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff}};
static const TPM2B_32_BYTE_VALUE SM2_P256_A = {32,{
                                        0xff,0xff,0xff,0xfe,0xff,0xff,0xff,0xff,
                                        0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
                                        0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
                                        0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xfc}};
static const TPM2B_32_BYTE_VALUE SM2_P256_B = {32,{
                                        0x28,0xe9,0xfa,0x9e,0x9d,0x9f,0x5e,0x34,
                                        0x4d,0x5a,0x9e,0x4b,0xcf,0x65,0x09,0xa7,
                                        0xf3,0x97,0x89,0xf5,0x15,0xab,0x8f,0x92,
                                        0xDD,0xBC,0xBD,0x41,0x4D,0x94,0x0E,0x93}};
static const TPM2B_32_BYTE_VALUE SM2_P256_X = {32,{
                                        0x32,0xC4,0xAE,0x2C,0x1F,0x19,0x81,0x19,
                                        0x5F,0x99,0x04,0x46,0x6A,0x39,0xC9,0x94,
                                        0x8F,0xE3,0x0B,0xBF,0xF2,0x66,0x0B,0xE1,
                                        0x71,0x5A,0x45,0x89,0x33,0x4C,0x74,0xC7}};
static const TPM2B_32_BYTE_VALUE SM2_P256_Y = {32,{
                                        0xBC,0x37,0x36,0xA2,0xF4,0xF6,0x77,0x9C,
                                        0x59,0xBD,0xCE,0xE3,0x6B,0x69,0x21,0x53,
                                        0xD0,0xA9,0x87,0x7C,0xC6,0x2A,0x47,0x40,
                                        0x02,0xDF,0x32,0xE5,0x21,0x39,0xF0,0xA0}};
static const TPM2B_32_BYTE_VALUE SM2_P256_N = {32,{
                                        0xFF,0xFF,0xFF,0xFE,0xFF,0xFF,0xFF,0xFF,
                                        0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
                                        0x72,0x03,0xDF,0x6B,0x21,0xC6,0x05,0x2B,
                                        0x53,0xBB,0xF4,0x09,0x39,0xD5,0x41,0x23}};
#else //_SM2_SIGN_DEBUG
```

These are the values for debug of SM2 sign

```
static const TPM2B_32_BYTE_VALUE SM2_P256_P = {32,{
                                        0x85,0x42,0xD6,0x9E,0x4C,0x04,0x4F,0x18,
                                        0xE8,0xB9,0x24,0x35,0xBF,0x6F,0xF7,0xDE,
                                        0x45,0x72,0x83,0x91,0x5C,0x45,0x51,0x7D,
                                        0x72,0x2E,0xDB,0x8B,0x08,0xF1,0xDF,0xC3}};
static const TPM2B_32_BYTE_VALUE SM2_P256_A = {32,{
                                        0x78,0x79,0x68,0xB4,0xFA,0x32,0xC3,0xFD,
                                        0x24,0x17,0x84,0x2E,0x73,0xBB,0xFE,0xFF,
                                        0x2F,0x3C,0x84,0x8B,0x68,0x31,0xD7,0xE0,
                                        0xEC,0x65,0x22,0x8B,0x39,0x37,0xE4,0x98}};
static const TPM2B_32_BYTE_VALUE SM2_P256_B = {32,{
                                        0x63,0xE4,0xC6,0xD3,0xB2,0x3B,0x0C,0x84,
                                        0x9C,0xF8,0x42,0x41,0x48,0x4B,0xFE,0x48,
                                        0xF6,0x1D,0x59,0xA5,0xB1,0x6B,0xA0,0x6E,
                                        0x6E,0x12,0xD1,0xDA,0x27,0xC5,0x24,0x9A}};
static const TPM2B_32_BYTE_VALUE SM2_P256_X = {32,{
                                        0x42,0x1D,0xEB,0xD6,0x1B,0x62,0xEA,0xB6,
                                        0x74,0x64,0x34,0xEB,0xC3,0xCC,0x31,0x5E,
                                        0x32,0x22,0x0B,0x3B,0xAD,0xD5,0x0B,0xDC,
                                        0x4C,0x4E,0x6C,0x14,0x7F,0xED,0xD4,0x3D}};
```

```
106   static const TPM2B_32_BYTE_VALUE SM2_P256_Y = {32,{
107                                         0x06,0x80,0x51,0x2B,0xCB,0xB4,0x2C,0x07,
108                                         0xD4,0x73,0x49,0xD2,0x15,0x3B,0x70,0xC4,
109                                         0xE5,0xD7,0xFD,0xFC,0xBF,0xA3,0x6E,0xA1,
110                                         0xA8,0x58,0x41,0xB9,0xE4,0x6E,0x09,0xA2}};
111   static const TPM2B_32_BYTE_VALUE SM2_P256_N = {32,{
112                                         0x85,0x42,0xD6,0x9E,0x4C,0x04,0x4F,0x18,
113                                         0xE8,0xB9,0x24,0x35,0xBF,0x6F,0xF7,0xDD,
114                                         0x29,0x77,0x20,0x63,0x04,0x85,0x62,0x8D,
115                                         0x5A,0xE7,0x4E,0xE7,0xC3,0x2E,0x79,0xB7}};
116   #endif
117   static const TPM2B_1_BYTE_VALUE   SM2_P256_H = {1, {1}};
118   const ECC_CURVE_DATA SM2_P256 = {&SM2_P256_P.b, &SM2_P256_A.b, &SM2_P256_B.b,
119                                    &SM2_P256_X.b, &SM2_P256_Y.b, &SM2_P256_N.b,
120                                    &SM2_P256_H.b};
121   #endif
```

Make sure that this table has algorithms in the same order as the *eccCurveValues*[] table in CryptUtil.c

```
122   const ECC_CURVE   eccCurves[] =
123   {
124       {TPM_ECC_NIST_P256,             // curveId
125        256,                           // key size in bits
126       {TPM_ALG_NULL, {TPM_ALG_NULL}}, // default KDF and hash
127       {TPM_ALG_NULL, {TPM_ALG_NULL}}, // default signing scheme and hash
128       &NIST_P256}                     // curve values
129   #ifdef  TPM_ECC_SM2_P256
130      ,{TPM_ECC_SM2_P256,
131        256,
132       {TPM_ALG_NULL, {TPM_ALG_NULL}},
133       {TPM_ALG_NULL, {TPM_ALG_NULL}},
134       &SM2_P256}
135   #endif
136   #ifdef  TPM_ALG_ECDAA
137      ,{TPM_ECC_BN_P256,
138        256,
139       {TPM_ALG_NULL, {TPM_ALG_NULL}},
140       {TPM_ALG_ECDAA, {TPM_ALG_NULL}},
141       &BN_P256}
142   #endif
143   };
144   const UINT16    ECC_CURVE_COUNT = sizeof(eccCurves) / sizeof(ECC_CURVE);
```

B.11.3. CpriECC.c

B.11.3.1. Includes and Defines

```
#include    "OsslCryptoEngine.h"
```

B.11.3.2. Functions

B.11.3.2.1. _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.

```
BOOL
_cpri__EccStartup(
    void
)
{
    return TRUE;
}
```

B.11.3.2.2. _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.

```
TPM_ECC_CURVE
_cpri__GetCurveIdByIndex(
    UINT16      i
)
{
    if(i >= ECC_CURVE_COUNT)
       return TPM_ECC_NONE;
    return eccCurves[i].curveId;
}
UINT32
_cpri__EccGetCurveCount(
    void
    )
{
    return ECC_CURVE_COUNT;
}
```

B.11.3.2.3. _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

```
25  const ECC_CURVE *
26  _cpri__EccGetParametersByCurveId(
27      TPM_ECC_CURVE       curveId     // IN: the curveID
28  )
29  {
30      int          i;
31      for(i = 0; i < ECC_CURVE_COUNT; i++)
32      {
33          if(eccCurves[i].curveId == curveId)
34              return &eccCurves[i];
35      }
36      FAIL(FATAL_ERROR_INTERNAL);
37      return NULL;    // unreachable code to avoid compiler warning.
38  }
39  static const ECC_CURVE_DATA *
40  GetCurveData(
41      TPM_ECC_CURVE        curveId     // IN: the curveID
42      )
43  {
44      const ECC_CURVE     *curve = _cpri__EccGetParametersByCurveId(curveId);
45      return curve->curveData;
46  }
```

B.11.3.2.4. Point2B()

This function makes a TPMS_ECC_POINT from a BIGNUM EC_POINT.

```
47  static BOOL
48  Point2B(
49      EC_GROUP        *group,         // IN: group for the point
50      TPMS_ECC_POINT  *p,             // OUT: receives the converted point
51      EC_POINT        *ecP,           // IN: the point to convert
52      UINT16           size,          // IN: size of the coordinates
53      BN_CTX          *context        // IN: working context
54      )
55  {
56      BIGNUM          *bnX;
57      BIGNUM          *bnY;
58
59      BN_CTX_start(context);
60      bnX = BN_CTX_get(context);
61      bnY = BN_CTX_get(context);
62
63      if(     bnY == NULL
64
65          // Get the coordinate values
66         ||  EC_POINT_get_affine_coordinates_GFp(group, ecP, bnX, bnY, context) != 1
67
68         // Convert x
69         ||  (!BnTo2B(&p->x.b, bnX, size))
70
71         // Convert y
72         ||  (!BnTo2B(&p->y.b, bnY, size))
73        )
74              FAIL(FATAL_ERROR_INTERNAL);
75
76      BN_CTX_end(context);
```

```
77      return TRUE;
78  }
```

B.11.3.2.5. 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 *

```
79  static EC_GROUP *
80  EccCurveInit(
81      TPM_ECC_CURVE            curveId,          // IN: the ID of the curve
82      BN_CTX                  *groupContext      // IN: the context in which the
83                                                 //     group is to be created
84  )
85  {
86      const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
87      EC_GROUP                *group = NULL;
88      EC_POINT                *P = NULL;
89      BN_CTX                  *context;
90      BIGNUM                  *bnP;
91      BIGNUM                  *bnA;
92      BIGNUM                  *bnB;
93      BIGNUM                  *bnX;
94      BIGNUM                  *bnY;
95      BIGNUM                  *bnN;
96      BIGNUM                  *bnH;
97      int                      ok = FALSE;
98
99      // Context must be provided and curve selector must be valid
100     pAssert(groupContext != NULL &&  curveData != NULL);
101
102     context = BN_CTX_new();
103     if(context == NULL)
104         FAIL(FATAL_ERROR_ALLOCATION);
105
106     BN_CTX_start(context);
107     bnP = BN_CTX_get(context);
108     bnA = BN_CTX_get(context);
109     bnB = BN_CTX_get(context);
110     bnX = BN_CTX_get(context);
111     bnY = BN_CTX_get(context);
112     bnN = BN_CTX_get(context);
113     bnH = BN_CTX_get(context);
114
115     if (bnH == NULL)
116         goto Cleanup;
117
118     // Convert the number formats
119
120     BnFrom2B(bnP, curveData->p);
121     BnFrom2B(bnA, curveData->a);
122     BnFrom2B(bnB, curveData->b);
123     BnFrom2B(bnX, curveData->x);
124     BnFrom2B(bnY, curveData->y);
125     BnFrom2B(bnN, curveData->n);
126     BnFrom2B(bnH, curveData->h);
```

```
127
128         // initialize EC group, associate a generator point and initialize the point
129         // from the parameter data
130         ok = (   (group = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, groupContext)) != NULL
131               && (P = EC_POINT_new(group)) != NULL
132               && EC_POINT_set_affine_coordinates_GFp(group, P, bnX, bnY, groupContext)
133               && EC_GROUP_set_generator(group, P, bnN, bnH)
134              );
135     Cleanup:
136         if (!ok && group != NULL)
137         {
138             EC_GROUP_free(group);
139             group = NULL;
140         }
141         if(P != NULL)
142             EC_POINT_free(P);
143         BN_CTX_end(context);
144         BN_CTX_free(context);
145         return group;
146     }
```

B.11.3.2.6. PointFrom2B()

This function sets the coordinates of an existing BN Point from a TPMS_ECC_POINT.

```
147     static EC_POINT *
148     PointFrom2B(
149         EC_GROUP        *group,         // IN: the group for the point
150         EC_POINT        *ecP,           // IN: an existing BN point in the group
151         TPMS_ECC_POINT  *p,             // IN: the 2B coordinates of the point
152         BN_CTX          *context        // IN: the BIGNUM context
153         )
154     {
155         BIGNUM          *bnX;
156         BIGNUM          *bnY;
157
158         // If the point is not allocated then just return a NULL
159         if(ecP == NULL)
160             return NULL;
161
162         BN_CTX_start(context);
163         bnX = BN_CTX_get(context);
164         bnY = BN_CTX_get(context);
165         if( // Set the coordinates of the point
166               bnY == NULL
167            || BN_bin2bn(p->x.t.buffer, p->x.t.size, bnX) == NULL
168            || BN_bin2bn(p->y.t.buffer, p->y.t.size, bnY) == NULL
169            || !EC_POINT_set_affine_coordinates_GFp(group, ecP, bnX, bnY, context)
170            )
171            FAIL(FATAL_ERROR_INTERNAL);
172
173         BN_CTX_end(context);
174         return ecP;
175     }
```

B.11.3.2.7. EccInitPoint2B()

This function allocates a point in the provided group and initializes it with the values in a TPMS_ECC_POINT.

```
176     static EC_POINT *
177     EccInitPoint2B(
178         EC_GROUP        *group,         // IN: group for the point
```

```
179         TPMS_ECC_POINT  *p,             // IN: the coordinates for the point
180         BN_CTX          *context        // IN: the BIGNUM context
181         )
182     {
183         EC_POINT        *ecP;
184
185         BN_CTX_start(context);
186         ecP = EC_POINT_new(group);
187
188         if(PointFrom2B(group, ecP, p, context) == NULL)
189             FAIL(FATAL_ERROR_INTERNAL);
190
191         BN_CTX_end(context);
192         return ecP;
193     }
```

B.11.3.2.8. 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

```
194     static CRYPT_RESULT
195     PointMul(
196         EC_GROUP        *group,         // IN: group curve
197         EC_POINT        *ecpQ,          // OUT: result
198         BIGNUM          *bnA,           // IN: scalar for [A]G
199         EC_POINT        *ecpP,          // IN: point for [B]P
200         BIGNUM          *bnB,           // IN: scalar for [B]P
201         BN_CTX          *context        // IN: working context
202     )
203     {
204           if(EC_POINT_mul(group, ecpQ, bnA, ecpP, bnB, context) != 1)
205               FAIL(FATAL_ERROR_INTERNAL);
206           if(EC_POINT_is_at_infinity(group, ecpQ))
207               return CRYPT_NO_RESULT;
208           return CRYPT_SUCCESS;
209     }
```

B.11.3.2.9. 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)

```
210     static void
211     GetRandomPrivate(
212         TPM2B_ECC_PARAMETER     *dOut,      // OUT: the qualified random value
213         const TPM2B             *pIn        // IN: the maximum value for the key
214     )
215     {
216         int          i;
217         BYTE        *pb;
218
219         pAssert(pIn != NULL && dOut != NULL && pIn->size <= MAX_ECC_KEY_BYTES);
220
221         // Set the size of the output
```

```
222         dOut->t.size = pIn->size;
223         // Get some random bits
224         while(TRUE)
225         {
226             _cpri__GenerateRandom(dOut->t.size, dOut->t.buffer);
227             // See if the d < n
228             if(memcmp(dOut->t.buffer, pIn->buffer, pIn->size) < 0)
229             {
230                 // dOut < n so make sure that 0 < dOut
231                 for(pb = dOut->t.buffer, i = dOut->t.size; i > 0; i--)
232                 {
233                     if(*pb++ != 0)
234                         return;
235                 }
236             }
237         }
238     }
```

B.11.3.2.10. Mod2B()

Function does modular reduction of TPM2B values.

```
239     static CRYPT_RESULT
240     Mod2B(
241         TPM2B           *x,     // IN/OUT: value to reduce
242         const TPM2B     *n      // IN: mod
243         )
244     {
245         int         compare;
246         compare = _math__uComp(x->size, x->buffer, n->size, n->buffer);
247         if(compare < 0)
248             // if x < n, then mod is x
249             return CRYPT_SUCCESS;
250         if(compare == 0)
251         {
252             // if x == n then mod is 0
253             x->size = 0;
254             x->buffer[0] = 0;
255             return CRYPT_SUCCESS;
256         }
257        return _math__Div(x, n, NULL, x);
258     }
```

B.11.3.2.11. _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

```
259 CRYPT_RESULT
260 _cpri__EccPointMultiply(
261     TPMS_ECC_POINT      *Rout,              // OUT: the product point R
262     TPM_ECC_CURVE        curveId,           // IN: the curve to use
263     TPM2B_ECC_PARAMETER *dIn,               // IN: value to multiply against
264                                             //     the curve generator
265     TPMS_ECC_POINT      *Qin,               // IN: point Q
266     TPM2B_ECC_PARAMETER *uIn                // IN: scalar value for the multiplier
267                                             //     of Q
268 )
269 {
270     BN_CTX              *context;
271     BIGNUM              *bnD;
272     BIGNUM              *bnU;
273     EC_GROUP            *group;
274     EC_POINT            *R = NULL;
275     EC_POINT            *Q = NULL;
276     CRYPT_RESULT         retVal = CRYPT_SUCCESS;
277
278
279     // Validate that the required parameters are provided.
280     pAssert((dIn != NULL || uIn != NULL) && (Qin != NULL || dIn != NULL));
281
282     // If a point is provided for the multiply, make sure that it is on the curve
283     if(Qin != NULL && !_cpri__EccIsPointOnCurve(curveId, Qin))
284         return CRYPT_POINT;
285
286     context = BN_CTX_new();
287     if(context == NULL)
288         FAIL(FATAL_ERROR_ALLOCATION);
289
290     BN_CTX_start(context);
291     bnU = BN_CTX_get(context);
292     bnD = BN_CTX_get(context);
293     group = EccCurveInit(curveId, context);
294
295     // There should be no path for getting a bad curve ID into this function.
296     pAssert(group != NULL);
297
298     // check allocations should have worked and allocate R
299     if(   bnD == NULL
300        || (R = EC_POINT_new(group)) == NULL)
301         FAIL(FATAL_ERROR_ALLOCATION);
302
303     // If Qin is present, create the point
304     if(Qin != NULL)
305     {
306         // Assume the size variables do not overflow. This should not happen in
307         // the contexts in which this function will be called.
308         assert2Bsize(Qin->x.t);
309         assert2Bsize(Qin->x.t);
310         Q = EccInitPoint2B(group, Qin, context);
311
312     }
313     if(dIn != NULL)
314     {
315         // Assume the size variables do not overflow, which should not happen in
316         // the contexts that this function will be called.
```

```
317             assert2Bsize(dIn->t);
318             BnFrom2B(bnD, &dIn->b);
319         }
320         else
321             bnD = NULL;
322 
323         // If uIn is specified, initialize its BIGNUM
324         if(uIn != NULL)
325         {
326             // Assume the size variables do not overflow, which should not happen in
327             // the contexts that this function will be called.
328             assert2Bsize(uIn->t);
329             BnFrom2B(bnU, &uIn->b);
330         }
331         // If uIn is not specified but Q is, then we are going to
332         // do R = [d]Q
333         else if(Qin != NULL)
334         {
335             bnU = bnD;
336             bnD = NULL;
337         }
338         // If neither Q nor u is specified, then null this pointer
339         else
340             bnU = NULL;
341 
342         // Use the generator of the curve
343         if((retVal = PointMul(group, R, bnD, Q, bnU, context)) == CRYPT_SUCCESS)
344             Point2B(group, Rout, R, (UINT16) BN_num_bytes(&group->field), context);
345 
346         if (Q)
347             EC_POINT_free(Q);
348         if(R)
349             EC_POINT_free(R);
350         if(group)
351             EC_GROUP_free(group);
352         BN_CTX_end(context);
353         BN_CTX_free(context);
354         return retVal;
355     }
```

B.11.3.2.12. ClearPoint2B()

Initialize the size values of a point

```
356     static void
357     ClearPoint2B(
358         TPMS_ECC_POINT      *p          // IN: the point
359         )
360     {
361         if(p != NULL) {
362             p->x.t.size = 0;
363             p->y.t.size = 0;
364         }
365     }
366     #if defined TPM_ALG_ECDAA || defined TPM_ALG_SM2 //%
```

B.11.3.2.13. _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

```
367 CRYPT_RESULT
368 _cpri__EccCommitCompute(
369     TPMS_ECC_POINT      *K,         // OUT: [d]B or [r]Q
370     TPMS_ECC_POINT      *L,         // OUT: [r]B
371     TPMS_ECC_POINT      *E,         // OUT: [r]M
372     TPM_ECC_CURVE        curveId,   // IN: the curve for the computations
373     TPMS_ECC_POINT      *M,         // IN: M (optional)
374     TPMS_ECC_POINT      *B,         // IN: B (optional)
375     TPM2B_ECC_PARAMETER *d,         // IN: d (required)
376     TPM2B_ECC_PARAMETER *r          // IN: the computed r value (required)
377 )
378 {
379     BN_CTX              *context;
380     BIGNUM              *bnX, *bnY, *bnR, *bnD;
381     EC_GROUP            *group;
382     EC_POINT            *pK = NULL, *pL = NULL, *pE = NULL, *pM = NULL, *pB = NULL;
383     UINT16               keySizeInBytes;
384     CRYPT_RESULT         retVal = CRYPT_SUCCESS;
385
386     // Validate that the required parameters are provided.
387     // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
388     // E := [r]Q if both M and B are NULL.
389     pAssert(   r != NULL && (K != NULL || B == NULL) && (L != NULL || B == NULL)
390             || (E != NULL || (M == NULL && B != NULL)));
391
392     context = BN_CTX_new();
393     if(context == NULL)
394         FAIL(FATAL_ERROR_ALLOCATION);
395     BN_CTX_start(context);
396     bnR = BN_CTX_get(context);
397     bnD = BN_CTX_get(context);
398     bnX = BN_CTX_get(context);
399     bnY = BN_CTX_get(context);
400     if(bnY == NULL)
401         FAIL(FATAL_ERROR_ALLOCATION);
402
403     // Initialize the output points in case they are not computed
404     ClearPoint2B(K);
405     ClearPoint2B(L);
406     ClearPoint2B(E);
407
408     if((group = EccCurveInit(curveId, context)) == NULL)
409     {
410         retVal = CRYPT_PARAMETER;
411         goto Cleanup2;
412     }
413     keySizeInBytes = (UINT16) BN_num_bytes(&group->field);
414
415     // Sizes of the r and d parameters may not be zero
416     pAssert(((int) r->t.size > 0) && ((int) d->t.size > 0));
417
418     // Convert scalars to BIGNUM
419     BnFrom2B(bnR, &r->b);
420     BnFrom2B(bnD, &d->b);
421
```

```
422         // If B is provided, compute K=[d]B and L=[r]B
423         if(B != NULL)
424         {
425             // Allocate the points to receive the value
426             if(     (pK = EC_POINT_new(group)) == NULL
427                 || (pL = EC_POINT_new(group)) == NULL)
428             FAIL(FATAL_ERROR_ALLOCATION);
429             // need to compute K = [d]B
430             // Allocate and initialize BIGNUM version of B
431             pB = EccInitPoint2B(group, B, context);
432 
433             // do the math for K = [d]B
434             if((retVal = PointMul(group, pK, NULL, pB, bnD, context)) != CRYPT_SUCCESS)
435                 goto Cleanup;
436 
437             // Convert BN K to TPM2B K
438             Point2B(group, K, pK, keySizeInBytes, context);
439 
440             //  compute L= [r]B after checking for cancel
441             if(_plat__IsCanceled())
442             {
443                 retVal = CRYPT_CANCEL;
444                 goto Cleanup;
445             }
446             // compute L = [r]B
447             if((retVal = PointMul(group, pL, NULL, pB, bnR, context)) != CRYPT_SUCCESS)
448                 goto Cleanup;
449 
450             // Convert BN L to TPM2B L
451             Point2B(group, L, pL, keySizeInBytes, context);
452         }
453         if(M != NULL || B == NULL)
454         {
455             // if this is the third point multiply, check for cancel first
456             if(B != NULL && _plat__IsCanceled())
457             {
458                 retVal = CRYPT_CANCEL;
459                 goto Cleanup;
460             }
461 
462             // Allocate E
463             if((pE = EC_POINT_new(group)) == NULL)
464                 FAIL(FATAL_ERROR_ALLOCATION);
465 
466             // Create BIGNUM version of M unless M is NULL
467             if(M != NULL)
468             {
469                 // M provided so initialize a BIGNUM M and compute E = [r]M
470                 pM = EccInitPoint2B(group, M, context);
471                 retVal = PointMul(group, pE, NULL, pM, bnR, context);
472             }
473             else
474                 // compute E = [r]Q (this is only done if M and B are both NULL
475                 retVal = PointMul(group, pE, bnR, NULL, NULL, context);
476 
477             if(retVal == CRYPT_SUCCESS)
478                 // Convert E to 2B format
479                 Point2B(group, E, pE, keySizeInBytes, context);
480         }
481     Cleanup:
482         EC_GROUP_free(group);
483         if(pK != NULL) EC_POINT_free(pK);
484         if(pL != NULL) EC_POINT_free(pL);
485         if(pE != NULL) EC_POINT_free(pE);
486         if(pM != NULL) EC_POINT_free(pM);
487         if(pB != NULL) EC_POINT_free(pB);
```

```
488     Cleanup2:
489         BN_CTX_end(context);
490         BN_CTX_free(context);
491         return retVal;
492     }
493     #endif  //%
```

B.11.3.2.14. _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

```
494     BOOL
495     _cpri__EccIsPointOnCurve(
496         TPM_ECC_CURVE            curveId,       // IN: the curve selector
497         TPMS_ECC_POINT          *Q              // IN: the point.
498     )
499     {
500         BN_CTX                  *context;
501         BIGNUM                  *bnX;
502         BIGNUM                  *bnY;
503         BIGNUM                  *bnA;
504         BIGNUM                  *bnB;
505         BIGNUM                  *bnP;
506         BIGNUM                  *bn3;
507         const  ECC_CURVE_DATA   *curveData = GetCurveData(curveId);
508         BOOL                     retVal;
509
510         pAssert(Q != NULL && curveData != NULL);
511
512         if((context = BN_CTX_new()) == NULL)
513             FAIL(FATAL_ERROR_ALLOCATION);
514         BN_CTX_start(context);
515         bnX = BN_CTX_get(context);
516         bnY = BN_CTX_get(context);
517         bnA = BN_CTX_get(context);
518         bnB = BN_CTX_get(context);
519         bn3 = BN_CTX_get(context);
520         bnP = BN_CTX_get(context);
521         if(bnP == NULL)
522             FAIL(FATAL_ERROR_ALLOCATION);
523
524         // Convert values
525         if (   !BN_bin2bn(Q->x.t.buffer, Q->x.t.size, bnX)
526             || !BN_bin2bn(Q->y.t.buffer, Q->y.t.size, bnY)
527             || !BN_bin2bn(curveData->p->buffer, curveData->p->size, bnP)
528             || !BN_bin2bn(curveData->a->buffer, curveData->a->size, bnA)
529             || !BN_set_word(bn3, 3)
530             || !BN_bin2bn(curveData->b->buffer, curveData->b->size, bnB)
531            )
532             FAIL(FATAL_ERROR_INTERNAL);
533
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.11.3.2.15. _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     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 process.
564         TPM2B                   *extra,     // IN: Party 1 data for the KDF
565         UINT32                  *counter    // IN/OUT: Counter value to allow KDF
566                                             //         iteration to be propagated
567                                             //         across multiple functions
568     )
569     {
570         const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
571         UINT16                   keySizeInBytes;
572         UINT32                   count = 0;
573         CRYPT_RESULT             retVal;
574         UINT16                   hLen = _cpri__GetDigestSize(hashAlg);
575         BIGNUM                  *bnNm1;         // Order of the curve minus one
576         BIGNUM                  *bnD;           // the private scalar
577         BN_CTX                  *context;       // the context for the BIGNUM values
578         BYTE                     withExtra[MAX_ECC_KEY_BYTES + 8];  // trial key with
579                                                                     //extra bits
580         TPM2B_4_BYTE_VALUE       marshaledCounter = {4, {0}};
581         UINT32                   totalBits;
582
583         // Validate parameters (these are fatal)
```

```
584         pAssert(   seed != NULL && dOut != NULL && Qout != NULL && curveData != NULL);
585
586         // Non-fatal parameter checks.
587         if(hLen <= 0)
588             return CRYPT_PARAMETER;
589
590         // allocate the local BN values
591         context = BN_CTX_new();
592         if(context == NULL)
593             FAIL(FATAL_ERROR_ALLOCATION);
594         BN_CTX_start(context);
595         bnNm1 = BN_CTX_get(context);
596         bnD = BN_CTX_get(context);
597
598         // The size of the input scalars is limited by the size of the size of a
599         // TPM2B_ECC_PARAMETER. Make sure that it is not irrational.
600         pAssert((int) curveData->n->size <= MAX_ECC_KEY_BYTES);
601
602         if(   bnD == NULL
603            || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnNm1) == NULL
604            || (keySizeInBytes = (UINT16) BN_num_bytes(bnNm1)) > MAX_ECC_KEY_BYTES)
605             FAIL(FATAL_ERROR_INTERNAL);
606
607         // get the total number of bits
608         totalBits = BN_num_bits(bnNm1) + 64;
609
610         // Reduce bnNm1 from 'n' to 'n' - 1
611         BN_sub_word(bnNm1, 1);
612
613         // Initialize the count value
614         if(counter != NULL)
615             count = *counter;
616         if(count == 0)
617             count = 1;
618
619         // Start search for key (should be quick)
620         for(; count != 0; count++)
621         {
622
623             UINT32_TO_BYTE_ARRAY(count, marshaledCounter.t.buffer);
624             _cpri__KDFa(hashAlg, seed, label, extra, &marshaledCounter.b,
625                         totalBits, withExtra, NULL, FALSE);
626
627             // Convert the result and modular reduce
628             // Assume the size variables do not overflow, which should not happen in
629             // the contexts that this function will be called.
630             pAssert(keySizeInBytes <= MAX_ECC_KEY_BYTES);
631             if (   BN_bin2bn(withExtra, keySizeInBytes+8, bnD) == NULL
632                 || BN_mod(bnD, bnD, bnNm1, context) != 1)
633                 FAIL(FATAL_ERROR_INTERNAL);
634
635             // Add one to get 0 < d < n
636             BN_add_word(bnD, 1);
637             if(BnTo2B(&dOut->b, bnD, keySizeInBytes) != 1)
638                     FAIL(FATAL_ERROR_INTERNAL);
639
640             // Do the point multiply to create the public portion of the key. If
641             // the multiply generates the point at infinity (unlikely), do another
642             // iteration.
643             if(    (retVal = _cpri__EccPointMultiply(Qout, curveId, dOut, NULL, NULL))
644                 != CRYPT_NO_RESULT)
645                 break;
646         }
647
648         if(count == 0) // if counter wrapped, then the TPM should go into failure mode
649             FAIL(FATAL_ERROR_INTERNAL);
```

```
650
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.11.3.2.16. _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     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.11.3.2.17. 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     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 
783         // Set z = leftmost bits of the digest
784         // NOTE: This is implemented such that the key size needs to be
785         //       an even number of bytes in length.
786         if(digest->size > curveData->n->size)
787         {
788             // Assume the size variables do not overflow, which should not happen
789             // in the contexts that this function will be called.
790             pAssert(curveData->n->size <= MAX_ECC_KEY_BYTES);
791             // digest is larger than n so truncate
792             BN_bin2bn(digest->buffer, curveData->n->size, bnZ);
793         }
794         else
795         {
796             // Assume the size variables do not overflow, which should not happen
797             // in the contexts that this function will be called.
798             pAssert(digest->size <= MAX_DIGEST_SIZE);
799             // digest is same or smaller than n so use it all
800             BN_bin2bn(digest->buffer, digest->size, bnZ);
801         }
802 
803         // Assume the size variables do not overflow, which should not happen in
804         // the contexts that this function will be called.
805         assert2Bsize(dIn->t);
806         if(   bnZ == NULL
807 
808            // need the private scalar of the signing key
809            || BN_bin2bn(dIn->t.buffer, dIn->t.size, bnD) == NULL)
810             FAIL(FATAL_ERROR_INTERNAL);
811 
812 
813         // NOTE: When the result of an operation is going to be reduced mod x
814         // any modular multiplication is done so that the intermediate values
815         // don't get too large.
816         //
817         // now have inverse of K (bnIk), z (bnZ), r (bnR),  d (bnD) and n (bnN)
818         // Compute s = k^-1 (z + r*d)(mod n)
819             //  first do d = r*d mod n
820         if(  !BN_mod_mul(bnD, bnR, bnD, bnN, context)
821 
822            // d = z + r * d
823            || !BN_add(bnD, bnZ, bnD)
824 
825            // d = k^(-1)(z + r * d)(mod n)
826            || !BN_mod_mul(bnD, bnIk, bnD, bnN, context)
827 
828            // convert to TPM2B format
829            || !BnTo2B(&sOut->b, bnD, curveData->n->size)
830 
831            // and write the modular reduced version of r
```

```
832             // NOTE: this was deferred to reduce the number of
833             // error checks.
834             || !BnTo2B(&rOut->b, bnR, curveData->n->size))
835              FAIL(FATAL_ERROR_INTERNAL);
836
837         if(!BN_is_zero(bnD))
838             break;  // signature not zero so done
839
840         // if the signature value was zero, start over
841     }
842
843     // Free up allocated BN values
844     BN_CTX_end(context);
845     BN_CTX_free(context);
846     return retVal;
847 }
848 #endif  //%
849 #if defined TPM_ALG_ECDAA || defined TPM_ALG_ECSCHNORR  //%
```

B.11.3.2.18. 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

```
850 static CRYPT_RESULT
851 EcDaa(
852     TPM2B_ECC_PARAMETER     *tOut,      // OUT: T component of the signature
853     TPM2B_ECC_PARAMETER     *sOut,      // OUT: s component of the signature
854     TPM_ECC_CURVE            curveId,   // IN: the curve used in signing
855     TPM2B_ECC_PARAMETER     *dIn,       // IN: the private key
856     TPM2B                   *tIn,       // IN: the value to sign
857     TPM2B_ECC_PARAMETER     *kIn        // IN: a random value from commit
858 )
859 {
860     BIGNUM                  *bnN, *bnK, *bnT, *bnD;
861     BN_CTX                  *context;
862     const TPM2B             *n;
863     const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
864     BOOL                     OK = TRUE;
865
866     // Parameter checks
867      pAssert(   sOut != NULL && dIn != NULL && tOut != NULL
868              && kIn != NULL && curveData != NULL);
869
870     // this just saves key strokes
871     n = curveData->n;
872
873     if(tIn != NULL)
874         Copy2B(&tOut->b, tIn);
875
876     // The size of dIn and kIn input scalars is limited by the size of the size
877     // of a TPM2B_ECC_PARAMETER and tIn can be no larger than a digest.
```

```
878         // Make sure they are within range.
879         pAssert(    (int) dIn->t.size <= MAX_ECC_KEY_BYTES
880                 && (int) kIn->t.size <= MAX_ECC_KEY_BYTES
881                 && (int) tOut->t.size <= MAX_DIGEST_SIZE
882                );
883 
884         context = BN_CTX_new();
885         if(context == NULL)
886             FAIL(FATAL_ERROR_ALLOCATION);
887         BN_CTX_start(context);
888         bnN = BN_CTX_get(context);
889         bnK = BN_CTX_get(context);
890         bnT = BN_CTX_get(context);
891         bnD = BN_CTX_get(context);
892 
893         // Check for allocation problems
894         if(bnD == NULL)
895             FAIL(FATAL_ERROR_ALLOCATION);
896 
897         // Convert values
898         if(   BN_bin2bn(n->buffer, n->size, bnN) == NULL
899            || BN_bin2bn(kIn->t.buffer, kIn->t.size, bnK) ==  NULL
900            || BN_bin2bn(dIn->t.buffer, dIn->t.size, bnD) == NULL
901            || BN_bin2bn(tOut->t.buffer, tOut->t.size, bnT) == NULL)
902 
903             FAIL(FATAL_ERROR_INTERNAL);
904         // Compute T = T mod n
905         OK = OK && BN_mod(bnT, bnT, bnN, context);
906 
907         // compute (s = k + T * d mod n)
908                 //   d = T * d mod n
909         OK = OK && BN_mod_mul(bnD, bnT, bnD, bnN, context) == 1;
910                 //   d = k + T * d mod n
911         OK = OK && BN_mod_add(bnD, bnK, bnD, bnN, context) == 1;
912                 //   s = d
913         OK = OK && BnTo2B(&sOut->b, bnD, n->size);
914                 //   r = T
915         OK = OK && BnTo2B(&tOut->b, bnT, n->size);
916         if(!OK)
917             FAIL(FATAL_ERROR_INTERNAL);
918 
919         // Cleanup
920         BN_CTX_end(context);
921         BN_CTX_free(context);
922 
923         return CRYPT_SUCCESS;
924     }
925     #endif  //%
926     #ifdef TPM_ALG_ECSCHNORR //%
```

B.11.3.2.19. 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

```
927  static CRYPT_RESULT
928  SchnorrEcc(
929      TPM2B_ECC_PARAMETER     *rOut,      // OUT: r component of the signature
930      TPM2B_ECC_PARAMETER     *sOut,      // OUT: s component of the signature
931      TPM_ALG_ID               hashAlg,   // IN: hash algorithm used
932      TPM_ECC_CURVE            curveId,   // IN: the curve used in signing
933      TPM2B_ECC_PARAMETER     *dIn,       // IN: the private key
934      TPM2B                   *digest,    // IN: the digest to sign
935      TPM2B_ECC_PARAMETER     *kIn        // IN: for testing
936  )
937  {
938      TPM2B_ECC_PARAMETER      k;
939      BIGNUM                  *bnR, *bnN, *bnK, *bnT, *bnD;
940      BN_CTX                  *context;
941      const TPM2B             *n;
942      EC_POINT                *pR = NULL;
943      EC_GROUP                *group = NULL;
944      CPRI_HASH_STATE          hashState;
945      UINT16                   digestSize = _cpri__GetDigestSize(hashAlg);
946      const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
947      TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_PARAMETER_BYTES));
948      TPM2B_T                  T2b;
949      BOOL                     OK = TRUE;
950
951
952      // Parameter checks
953
954      // Must have a place for the 'r' and 's' parts of the signature, a private
955      // key ('d')
956      pAssert(   rOut != NULL && sOut != NULL && dIn != NULL
957              && digest != NULL && curveData != NULL);
958
959      // to save key strokes
960      n = curveData->n;
961
962      // If the digest does not produce a hash, then null the signature and return
963      // a failure.
964      if(digestSize == 0)
965      {
966          rOut->t.size = 0;
967          sOut->t.size = 0;
968          return CRYPT_SCHEME;
969      }
970
971      // Allocate big number values
972      context = BN_CTX_new();
973      if(context == NULL)
974          FAIL(FATAL_ERROR_ALLOCATION);
975      BN_CTX_start(context);
976      bnR = BN_CTX_get(context);
977      bnN = BN_CTX_get(context);
978      bnK = BN_CTX_get(context);
979      bnT = BN_CTX_get(context);
980      bnD = BN_CTX_get(context);
981      if(   bnD == NULL
982              // initialize the group parameters
983         || (group = EccCurveInit(curveId, context)) == NULL
984             // allocate a local point
985         || (pR = EC_POINT_new(group)) == NULL
```

```
986         )
987           FAIL(FATAL_ERROR_ALLOCATION);
988
989       if(BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
990           FAIL(FATAL_ERROR_INTERNAL);
991
992       while(OK)
993       {
994   // a) set k to a random value such that 1  k  n-1
995           if(kIn != NULL)
996           {
997               Copy2B(&k.b, &kIn->b);  // copy input k if testing
998               OK = FALSE;             // not OK to loop
999           }
1000          else
1001          // If get a random value in the correct range
1002              GetRandomPrivate(&k, n);
1003
1004          // Convert 'k' and generate pR = ['k']G
1005          BnFrom2B(bnK, &k.b);
1006
1007  // b) compute E  (xE, yE)  [k]G
1008          if(PointMul(group, pR, bnK, NULL, NULL, context) == CRYPT_NO_RESULT)
1009  // c) if E is the point at infinity, go to a)
1010              continue;
1011
1012  // d) compute e  xE (mod n)
1013          // Get the x coordinate of the point
1014          EC_POINT_get_affine_coordinates_GFp(group, pR, bnR, NULL, context);
1015
1016          // make (mod n)
1017          BN_mod(bnR, bnR, bnN, context);
1018
1019  // e) if e is zero, go to a)
1020          if(BN_is_zero(bnR))
1021              continue;
1022
1023          // Convert xR to a string (use T as a temp)
1024          BnTo2B(&T2b.b, bnR, (UINT16)(BN_num_bits(bnR)+7)/8);
1025
1026  // f) compute r  HschemeHash(P || e) (mod n)
1027          _cpri__StartHash(hashAlg, FALSE, &hashState);
1028          _cpri__UpdateHash(&hashState, digest->size, digest->buffer);
1029          _cpri__UpdateHash(&hashState, T2b.t.size, T2b.t.buffer);
1030          if(_cpri__CompleteHash(&hashState, digestSize, T2b.b.buffer) != digestSize)
1031              FAIL(FATAL_ERROR_INTERNAL);
1032          T2b.t.size = digestSize;
1033          BnFrom2B(bnT, &T2b.b);
1034          BN_div(NULL, bnT, bnT, bnN, context);
1035          BnTo2B(&rOut->b, bnT, (UINT16)BN_num_bytes(bnT));
1036
1037          // We have a value and we are going to exit the loop successfully
1038          OK = TRUE;
1039          break;
1040      }
1041      // Cleanup
1042      EC_POINT_free(pR);
1043      EC_GROUP_free(group);
1044      BN_CTX_end(context);
1045      BN_CTX_free(context);
1046
1047      // If we have a value, finish the signature
1048      if(OK)
1049          return EcDaa(rOut, sOut, curveId, dIn, NULL, &k);
1050      else
1051          return CRYPT_NO_RESULT;
```

```
1052    }
1053    #endif  //%
1054    #ifdef TPM_ALG_SM2 //%
1055    #ifdef  _SM2_SIGN_DEBUG //%
1056    static int
1057    cmp_bn2hex(
1058        BIGNUM      *bn,        //IN: big number value
1059        const char  *c          //IN: character string number
1060        )
1061    {
1062        int         result;
1063        BIGNUM      *bnC = BN_new();
1064        pAssert(bnC != NULL);
1065
1066        BN_hex2bn(&bnC, c);
1067        result = BN_ucmp(bn, bnC);
1068        BN_free(bnC);
1069        return result;
1070    }
1071    static int
1072    cmp_2B2hex(
1073        TPM2B       *a,         // IN: TPM2B number to compare
1074        const char  *c          // IN: character string
1075        )
1076    {
1077        int         result;
1078        int         sl = strlen(c);
1079        BIGNUM      *bnA;
1080
1081        result = (a->size * 2) - sl;
1082        if(result != 0)
1083            return result;
1084        pAssert((bnA = BN_bin2bn(a->buffer, a->size, NULL)) != NULL);
1085        result = cmp_bn2hex(bnA, c);
1086        BN_free(bnA);
1087        return result;
1088    }
1089    static void
1090    cpy_hexTo2B(
1091        TPM2B       *b,     // OUT: receives value
1092        const char  *c      // IN: source string
1093        )
1094    {
1095        BIGNUM      *bnB = BN_new();
1096        pAssert((strlen(c) & 1) == 0);  // must have an even number of digits
1097        b->size = strlen(c) / 2;
1098        BN_hex2bn(&bnB, c);
1099        pAssert(bnB != NULL);
1100        BnTo2B(b, bnB, b->size);
1101        BN_free(bnB);
1102
1103    }
1104    #endif //% _SM2_SIGN_DEBUG
```

B.11.3.2.20. 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

```
1105    static CRYPT_RESULT
1106    SignSM2(
1107        TPM2B_ECC_PARAMETER     *rOut,      // OUT: r component of the signature
1108        TPM2B_ECC_PARAMETER     *sOut,      // OUT: s component of the signature
1109        TPM_ECC_CURVE            curveId,   // IN: the curve used in signing
1110        TPM2B_ECC_PARAMETER     *dIn,       // IN: the private key
1111        TPM2B                   *digest     // IN: the digest to sign
1112        )
1113    {
1114        BIGNUM                  *bnR;
1115        BIGNUM                  *bnS;
1116        BIGNUM                  *bnN;
1117        BIGNUM                  *bnK;
1118        BIGNUM                  *bnX1;
1119        BIGNUM                  *bnD;
1120        BIGNUM                  *bnT;   // temp
1121        BIGNUM                  *bnE;
1122
1123        BN_CTX                  *context;
1124        TPM2B_TYPE(DIGEST, MAX_DIGEST_SIZE);
1125        TPM2B_ECC_PARAMETER      k;
1126        TPMS_ECC_POINT           p2Br;
1127        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1128
1129        pAssert(curveData != NULL);
1130        context = BN_CTX_new();
1131        BN_CTX_start(context);
1132        bnK = BN_CTX_get(context);
1133        bnR = BN_CTX_get(context);
1134        bnS = BN_CTX_get(context);
1135        bnX1 = BN_CTX_get(context);
1136        bnN = BN_CTX_get(context);
1137        bnD = BN_CTX_get(context);
1138        bnT = BN_CTX_get(context);
1139        bnE = BN_CTX_get(context);
1140        if(bnE == NULL)
1141            FAIL(FATAL_ERROR_ALLOCATION);
1142
1143        BnFrom2B(bnE, digest);
1144        BnFrom2B(bnN, curveData->n);
1145        BnFrom2B(bnD, &dIn->b);
1146
1147    #ifdef _SM2_SIGN_DEBUG
1148    BN_hex2bn(&bnE, "B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76");
1149    BN_hex2bn(&bnD, "128B2FA8BD433C6C068C8D803DFF79792A519A55171B1B650C23661D15897263");
1150    #endif
1151    // A3: Use random number generator to generate random number 1 <= k <= n-1;
1152    // NOTE: Ax: numbers are from the SM2 standard
1153        k.t.size = curveData->n->size;
1154    loop:
1155        {
1156            // Get a random number
1157            _cpri__GenerateRandom(k.t.size, k.t.buffer);
1158
1159    #ifdef _SM2_SIGN_DEBUG
1160    BN_hex2bn(&bnK, "6CB28D99385C175C94F94E934817663FC176D925DD72B727260DBAAE1FB2F96F");
1161    BnTo2B(&k.b,bnK, 32);
1162    k.t.size = 32;
1163    #endif
1164            //make sure that the number is 0 < k < n
1165            BnFrom2B(bnK, &k.b);
```

```
1166            if(    BN_ucmp(bnK, bnN) >= 0
1167                || BN_is_zero(bnK))
1168                goto loop;
1169
1170    // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
1171    // to details specified in 4.2.7 in Part 1 of this document, transform the
1172    // data type of x1 into an integer;
1173            if(   _cpri__EccPointMultiply(&p2Br, curveId, &k, NULL, NULL)
1174               == CRYPT_NO_RESULT)
1175                goto loop;
1176
1177            BnFrom2B(bnX1, &p2Br.x.b);
1178
1179     // A5: Figure out r = (e + x1) mod n,
1180            if(!BN_mod_add(bnR, bnE, bnX1, bnN, context))
1181                FAIL(FATAL_ERROR_INTERNAL);
1182    #ifdef _SM2_SIGN_DEBUG
1183    pAssert(cmp_bn2hex(bnR,
1184                       "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1")
1185            == 0);
1186    #endif
1187
1188               // if r=0 or r+k=n, return to A3;
1189             if(!BN_add(bnT, bnK, bnR))
1190                FAIL(FATAL_ERROR_INTERNAL);
1191
1192            if(BN_is_zero(bnR) || BN_ucmp(bnT, bnN) == 0)
1193                goto loop;
1194
1195    // A6: Figure out s = ((1 + dA)^-1  (k - r  dA)) mod n, if s=0, return to A3;
1196            // compute t = (1+d)-1
1197            BN_copy(bnT, bnD);
1198            if(   !BN_add_word(bnT, 1)
1199               || !BN_mod_inverse(bnT, bnT, bnN, context) // (1 + dA)^-1 mod n
1200               )
1201                FAIL(FATAL_ERROR_INTERNAL);
1202    #ifdef _SM2_SIGN_DEBUG
1203    pAssert(cmp_bn2hex(bnT,
1204                       "79BFCF3052C80DA7B939E0C6914A18CBB2D96D8555256E83122743A7D4F5F956")
1205            == 0);
1206    #endif
1207           // compute s = t * (k - r * dA) mod n
1208            if(   !BN_mod_mul(bnS, bnD, bnR, bnN, context) // (r * dA) mod n
1209               || !BN_mod_sub(bnS, bnK, bnS, bnN, context) // (k - (r * dA) mod n
1210               || !BN_mod_mul(bnS, bnT, bnS, bnN, context))// t * (k - (r * dA) mod n
1211               FAIL(FATAL_ERROR_INTERNAL);
1212    #ifdef _SM2_SIGN_DEBUG
1213    pAssert(cmp_bn2hex(bnS,
1214                       "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7")
1215            == 0);
1216    #endif
1217
1218            if(BN_is_zero(bnS))
1219                goto loop;
1220        }
1221
1222    // A7: According to details specified in 4.2.1 in Part 1 of this document, transform
1223    // the data type of r, s into bit strings, signature of message M is (r, s).
1224
1225        BnTo2B(&rOut->b, bnR, curveData->n->size);
1226        BnTo2B(&sOut->b, bnS, curveData->n->size);
1227    #ifdef _SM2_SIGN_DEBUG
1228    pAssert(cmp_2B2hex(&rOut->b,
1229                       "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1")
1230            == 0);
1231    pAssert(cmp_2B2hex(&sOut->b,
```

```
1232                 "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7")
1233         == 0);
1234 #endif
1235     BN_CTX_end(context);
1236     BN_CTX_free(context);
1237     return CRYPT_SUCCESS;
1238 }
1239 #endif  //% TMP_ALG_SM2
```

B.11.3.2.21. _cpri__SignEcc()

This function is the dispatch function for the various ECC-based signing schemes.

Return Value	Meaning
CRYPT_SCHEME	*scheme* is not supported

```
1240 CRYPT_RESULT
1241 _cpri__SignEcc(
1242     TPM2B_ECC_PARAMETER *rOut,      // OUT: r component of the signature
1243     TPM2B_ECC_PARAMETER *sOut,      // OUT: s component of the signature
1244     TPM_ALG_ID           scheme,    // IN: the scheme selector
1245     TPM_ALG_ID           hashAlg,   // IN: the hash algorithm if need
1246     TPM_ECC_CURVE        curveId,   // IN: the curve used in the signature process
1247     TPM2B_ECC_PARAMETER *dIn,       // IN: the private key
1248     TPM2B               *digest,    // IN: the digest to sign
1249     TPM2B_ECC_PARAMETER *kIn        // IN: k for input
1250 )
1251 {
1252     switch  (scheme)
1253     {
1254         case TPM_ALG_ECDSA:
1255             // SignEcdsa always works
1256             return SignEcdsa(rOut, sOut, curveId, dIn, digest);
1257             break;
1258 #ifdef TPM_ALG_ECDAA
1259         case TPM_ALG_ECDAA:
1260             if(rOut != NULL)
1261                 rOut->b.size = 0;
1262             return EcDaa(rOut, sOut, curveId, dIn, digest, kIn);
1263             break;
1264 #endif
1265 #ifdef TPM_ALG_ECSCHNORR
1266         case TPM_ALG_ECSCHNORR:
1267             return SchnorrEcc(rOut, sOut, hashAlg, curveId, dIn, digest, kIn);
1268             break;
1269 #endif
1270 #ifdef TPM_ALG_SM2
1271         case TPM_ALG_SM2:
1272             return SignSM2(rOut, sOut, curveId, dIn, digest);
1273             break;
1274 #endif
1275         default:
1276             return CRYPT_SCHEME;
1277     }
1278 }
1279 #ifdef TPM_ALG_ECDSA //%
```

B.11.3.2.22. ValidateSignatureEcdsa()

This function validates an ECDSA signature.

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid

```
1280    static CRYPT_RESULT
1281    ValidateSignatureEcdsa(
1282        TPM2B_ECC_PARAMETER     *rIn,       // IN: r component of the signature
1283        TPM2B_ECC_PARAMETER     *sIn,       // IN: s component of the signature
1284        TPM_ECC_CURVE            curveId,   // IN: the curve used in the signature
1285                                            //     process
1286        TPMS_ECC_POINT          *Qin,       // IN: the public point of the key
1287        TPM2B                   *digest     // IN: the digest that was signed
1288    )
1289    {
1290        TPM2B_ECC_PARAMETER      U1;
1291        TPM2B_ECC_PARAMETER      U2;
1292        TPMS_ECC_POINT           R;
1293        const TPM2B             *n;
1294        BN_CTX                  *context;
1295        EC_POINT                *pQ = NULL;
1296        EC_GROUP                *group = NULL;
1297        BIGNUM                  *bnU1;
1298        BIGNUM                  *bnU2;
1299        BIGNUM                  *bnR;
1300        BIGNUM                  *bnS;
1301        BIGNUM                  *bnW;
1302        BIGNUM                  *bnV;
1303        BIGNUM                  *bnN;
1304        BIGNUM                  *bnE;
1305        BIGNUM                  *bnGx;
1306        BIGNUM                  *bnGy;
1307        BIGNUM                  *bnQx;
1308        BIGNUM                  *bnQy;
1309        CRYPT_RESULT             retVal = CRYPT_FAIL;
1310        int                      t;
1311
1312        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1313
1314        // The curve selector should have been filtered by the unmarshaling process
1315        pAssert (curveData != NULL);
1316        n = curveData->n;
1317
1318    // 1. If r and s are not both integers in the interval [1, n - 1], output
1319    //    INVALID.
1320        if(   _math__uComp(rIn->t.size, rIn->t.buffer, n->size, n->buffer) >= 0
1321           || _math__uComp(sIn->t.size, sIn->t.buffer, n->size, n->buffer) >= 0
1322          )
1323           return CRYPT_FAIL;
1324
1325        context = BN_CTX_new();
1326        if(context == NULL)
1327            FAIL(FATAL_ERROR_ALLOCATION);
1328        BN_CTX_start(context);
1329        bnR = BN_CTX_get(context);
1330        bnS = BN_CTX_get(context);
1331        bnN = BN_CTX_get(context);
1332        bnE = BN_CTX_get(context);
1333        bnV = BN_CTX_get(context);
1334        bnW = BN_CTX_get(context);
1335        bnGx = BN_CTX_get(context);
1336        bnGy = BN_CTX_get(context);
1337        bnQx = BN_CTX_get(context);
1338        bnQy = BN_CTX_get(context);
```

```
1339        bnU1 = BN_CTX_get(context);
1340        bnU2 = BN_CTX_get(context);
1341
1342        // Assume the size variables do not overflow, which should not happen in
1343        // the contexts that this function will be called.
1344        assert2Bsize(Qin->x.t);
1345        assert2Bsize(rIn->t);
1346        assert2Bsize(sIn->t);
1347
1348        // BN_CTX_get() is sticky so only need to check the last value to know that
1349        // all worked.
1350        if(   bnU2 == NULL
1351
1352            // initialize the group parameters
1353           || (group = EccCurveInit(curveId, context)) == NULL
1354
1355           // allocate a local point
1356           || (pQ = EC_POINT_new(group)) == NULL
1357
1358           // use the public key values (QxIn and QyIn) to initialize Q
1359           || BN_bin2bn(Qin->x.t.buffer, Qin->x.t.size, bnQx) == NULL
1360           || BN_bin2bn(Qin->x.t.buffer, Qin->x.t.size, bnQy) == NULL
1361           || !EC_POINT_set_affine_coordinates_GFp(group, pQ, bnQx, bnQy, context)
1362
1363           // convert the signature values
1364           || BN_bin2bn(rIn->t.buffer, rIn->t.size, bnR) == NULL
1365           || BN_bin2bn(sIn->t.buffer, sIn->t.size, bnS) == NULL
1366
1367           // convert the curve order
1368           || BN_bin2bn(curveData->n->buffer, curveData->n->size, bnN) == NULL)
1369            FAIL(FATAL_ERROR_INTERNAL);
1370
1371
1372    // 2. Use the selected hash function to compute H0 = Hash(M0).
1373        // This is an input parameter
1374
1375    // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
1376        t = (digest->size > rIn->t.size) ? rIn->t.size : digest->size;
1377        if(BN_bin2bn(digest->buffer, t, bnE) == NULL)
1378            FAIL(FATAL_ERROR_INTERNAL);
1379
1380    // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
1381        if (BN_mod_inverse(bnW, bnS, bnN, context) == NULL)
1382            FAIL(FATAL_ERROR_INTERNAL);
1383
1384    // 5. Compute u1 = (e' *   w) mod n, and compute u2 = (r' *  w) mod n.
1385        if(   !BN_mod_mul(bnU1, bnE, bnW, bnN, context)
1386           || !BN_mod_mul(bnU2, bnR, bnW, bnN, context))
1387            FAIL(FATAL_ERROR_INTERNAL);
1388
1389        BnTo2B(&U1.b, bnU1, (UINT16) BN_num_bytes(bnU1));
1390        BnTo2B(&U2.b, bnU2, (UINT16) BN_num_bytes(bnU2));
1391
1392    // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
1393    //    scalar multiplication and EC addition (see [Routines]). If R is equal to
1394    //    the point at infinity O, output INVALID.
1395        if(_cpri__EccPointMultiply(&R, curveId, &U1, Qin, &U2) == CRYPT_SUCCESS)
1396        {
1397            // 7. Compute v = Rx mod n.
1398            if(   BN_bin2bn(R.x.t.buffer, R.x.t.size, bnV) == NULL
1399               || !BN_mod(bnV, bnV, bnN, context))
1400                FAIL(FATAL_ERROR_INTERNAL);
1401
1402        // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
1403            if(BN_cmp(bnV, bnR) == 0)
1404                retVal = CRYPT_SUCCESS;
```

```
1405        }
1406
1407        if(pQ != NULL) EC_POINT_free(pQ);
1408        if(group != NULL) EC_GROUP_free(group);
1409        BN_CTX_end(context);
1410        BN_CTX_free(context);
1411
1412        return retVal;
1413    }
1414    #endif      //% TPM_ALG_ECDSA
1415    #ifdef TPM_ALG_ECSCHNORR //%
```

B.11.3.2.23. ValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid
CRYPT_SCHEME	*hashAlg* is not supported

```
1416    static CRYPT_RESULT
1417    ValidateSignatureEcSchnorr(
1418        TPM2B_ECC_PARAMETER     *rIn,       // IN: r component of the signature
1419        TPM2B_ECC_PARAMETER     *sIn,       // IN: s component of the signature
1420        TPM_ALG_ID               hashAlg,   // IN: hash algorithm of the signature
1421        TPM_ECC_CURVE            curveId,   // IN: the curve used in the signature
1422                                            //     process
1423        TPMS_ECC_POINT          *Qin,       // IN: the public point of the key
1424        TPM2B                   *digest     // IN: the digest that was signed
1425    )
1426    {
1427        TPMS_ECC_POINT           pE;
1428        const TPM2B             *n;
1429        CPRI_HASH_STATE          hashState;
1430        TPM2B_DIGEST             rPrime;
1431        TPM2B_ECC_PARAMETER      minusR;
1432        UINT16                   digestSize = _cpri__GetDigestSize(hashAlg);
1433        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1434
1435        // The curve parameter should have been filtered by unmarshaling code
1436        pAssert(curveData != NULL);
1437
1438        if(digestSize == 0)
1439            return CRYPT_SCHEME;
1440
1441        // Input parameter validation
1442        pAssert(rIn != NULL && sIn != NULL && Qin != NULL && digest != NULL);
1443
1444        n = curveData->n;
1445
1446        // if sIn or rIn are not between 1 and N-1, signature check fails
1447        if(   _math__uComp(sIn->b.size, sIn->b.buffer, n->size, n->buffer) >= 0
1448           || _math__uComp(rIn->b.size, rIn->b.buffer, n->size, n->buffer) >= 0
1449          )
1450            return CRYPT_FAIL;
1451
1452        //E = [s]InG - [r]InQ
1453        _math__sub(n->size, n->buffer,
1454                   rIn->t.size, rIn->t.buffer,
1455                   &minusR.t.size, minusR.t.buffer);
1456        if(_cpri__EccPointMultiply(&pE, curveId, sIn, Qin, &minusR) != CRYPT_SUCCESS)
```

```
1457            return CRYPT_FAIL;
1458
1459        // Ex = Ex mod N
1460        if(Mod2B(&pE.x.b, n) != CRYPT_SUCCESS)
1461            FAIL(FATAL_ERROR_INTERNAL);
1462
1463        _math__Normalize2B(&pE.x.b);
1464
1465        // rPrime = h(digest || pE.x) mod n;
1466        _cpri__StartHash(hashAlg, FALSE, &hashState);
1467        _cpri__UpdateHash(&hashState, digest->size, digest->buffer);
1468        _cpri__UpdateHash(&hashState, pE.x.t.size, pE.x.t.buffer);
1469        if(_cpri__CompleteHash(&hashState, digestSize, rPrime.t.buffer) != digestSize)
1470            FAIL(FATAL_ERROR_INTERNAL);
1471
1472        rPrime.t.size = digestSize;
1473
1474        // rPrime = rPrime (mod n)
1475        if(Mod2B(&rPrime.b, n) != CRYPT_SUCCESS)
1476            FAIL(FATAL_ERROR_INTERNAL);
1477
1478        // If rIn and rPrime are not the same size, denormalize rPrime.
1479        if(rIn->t.size > rPrime.t.size)
1480            _math__Denormalize2B(&rPrime.b, rIn->t.size);
1481
1482        // see if the values match
1483        if ( rIn->t.size == rPrime.t.size
1484            && (memcmp(rIn->t.buffer, rPrime.t.buffer, rIn->t.size) == 0))
1485            return CRYPT_SUCCESS;
1486        else
1487            return CRYPT_FAIL;
1488    }
1489    #endif  //% TPM_ALG_ECSCHNORR
1490    #ifdef TPM_ALG_SM2  //%
```

B.11.3.2.24. ValidateSignatueSM2Dsa()

This function is used to validate an SM2 signature.

Return Value	Meaning
CRYPT_SUCCESS	signature valid
CRYPT_FAIL	signature not valid

```
1491    static CRYPT_RESULT
1492    ValidateSignatureSM2Dsa(
1493        TPM2B_ECC_PARAMETER     *rIn,       // IN: r component of the signature
1494        TPM2B_ECC_PARAMETER     *sIn,       // IN: s component of the signature
1495        TPM_ECC_CURVE            curveId,   // IN: the curve used in the signature
1496                                            //     process
1497        TPMS_ECC_POINT          *Qin,       // IN: the public point of the key
1498        TPM2B                   *digest     // IN: the digest that was signed
1499    )
1500    {
1501        BIGNUM                  *bnR;
1502        BIGNUM                  *bnRp;
1503        BIGNUM                  *bnT;
1504        BIGNUM                  *bnS;
1505        BIGNUM                  *bnE;
1506        EC_POINT                *pQ;
1507        BN_CTX                  *context;
1508        EC_GROUP                *group = NULL;
1509        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
```

```
1510        BOOL                    fail = FALSE;
1511
1512
1513        if((context = BN_CTX_new()) == NULL || curveData == NULL)
1514            FAIL(FATAL_ERROR_INTERNAL);
1515        bnR = BN_CTX_get(context);
1516        bnRp= BN_CTX_get(context);
1517        bnE = BN_CTX_get(context);
1518        bnT = BN_CTX_get(context);
1519        bnS = BN_CTX_get(context);
1520        if(   bnS == NULL
1521           || (group = EccCurveInit(curveId, context)) == NULL)
1522            FAIL(FATAL_ERROR_INTERNAL);
1523
1524    #ifdef  _SM2_SIGN_DEBUG
1525        cpy_hexTo2B(&Qin->x.b,
1526                "0AE4C7798AA0F119471BEE11825BE46202BB79E2A5844495E97C04FF4DF2548A");
1527        cpy_hexTo2B(&Qin->y.b,
1528                "7C0240F88F1CD4E16352A73C17B7F16F07353E53A176D684A9FE0C6BB798E857");
1529        cpy_hexTo2B(digest,
1530                "B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76");
1531    #endif
1532        pQ = EccInitPoint2B(group, Qin, context);
1533
1534    #ifdef _SM2_SIGN_DEBUG
1535        pAssert(EC_POINT_get_affine_coordinates_GFp(group, pQ, bnT, bnS, context));
1536        pAssert(cmp_bn2hex(bnT,
1537                    "0AE4C7798AA0F119471BEE11825BE46202BB79E2A5844495E97C04FF4DF2548A")
1538                == 0);
1539        pAssert(cmp_bn2hex(bnS,
1540                    "7C0240F88F1CD4E16352A73C17B7F16F07353E53A176D684A9FE0C6BB798E857")
1541                == 0);
1542    #endif
1543
1544        BnFrom2B(bnR, &rIn->b);
1545        BnFrom2B(bnS, &sIn->b);
1546        BnFrom2B(bnE, digest);
1547
1548    #ifdef _SM2_SIGN_DEBUG
1549    // Make sure that the input signature is the test signature
1550    pAssert(cmp_2B2hex(&rIn->b,
1551            "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1") == 0);
1552    pAssert(cmp_2B2hex(&sIn->b,
1553            "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7") == 0);
1554    #endif
1555
1556    // a) verify that r and s are in the inclusive interval 1 to (n  1)
1557        fail = BN_is_zero(bnR) || (BN_ucmp(bnR, &group->order) >= 0);
1558
1559        fail = BN_is_zero(bnS) || (BN_ucmp(bnS, &group->order) >= 0) || fail;
1560        if(fail)
1561        // There is no reason to continue. Since r and s are inputs from the caller,
1562        // they can know that the values are not in the proper range. So, exiting here
1563        // does not disclose any information.
1564            goto Cleanup;
1565
1566    // b) compute t  := (r + s) mod n
1567        if(!BN_mod_add(bnT, bnR, bnS, &group->order, context))
1568            FAIL(FATAL_ERROR_INTERNAL);
1569    #ifdef _SM2_SIGN_DEBUG
1570        pAssert(cmp_bn2hex(bnT,
1571                    "2B75F07ED7ECE7CCC1C8986B991F441AD324D6D619FE06DD63ED32E0C997C801")
1572                == 0);
1573    #endif
1574
1575    // c) verify that t > 0
```

```
1576        if(BN_is_zero(bnT)) {
1577            fail = TRUE;
1578            // set to a value that should allow rest of the computations to run without
1579            // trouble
1580            BN_copy(bnT, bnS);
1581        }
1582    // d) compute (x, y) := [s]G + [t]Q
1583        if(!EC_POINT_mul(group, pQ, bnS, pQ, bnT, context))
1584            FAIL(FATAL_ERROR_INTERNAL);
1585        // Get the x coordinate of the point
1586        if(!EC_POINT_get_affine_coordinates_GFp(group, pQ, bnT, NULL, context))
1587            FAIL(FATAL_ERROR_INTERNAL);
1588
1589    #ifdef  _SM2_SIGN_DEBUG
1590        pAssert(cmp_bn2hex(bnT,
1591                    "110FCDA57615705D5E7B9324AC4B856D23E6D9188B2AE47759514657CE25D112")
1592                    == 0);
1593    #endif
1594
1595    // e) compute r' := (e + x) mod n (the x coordinate is in bnT)
1596        if(!BN_mod_add(bnRp, bnE, bnT, &group->order, context))
1597            FAIL(FATAL_ERROR_INTERNAL);
1598
1599    // f) verify that r' = r
1600        fail = BN_ucmp(bnR, bnRp) != 0 || fail;
1601
1602    Cleanup:
1603        if(pQ) EC_POINT_free(pQ);
1604        if(group) EC_GROUP_free(group);
1605        BN_CTX_end(context);
1606        BN_CTX_free(context);
1607
1608        if(fail)
1609            return CRYPT_FAIL;
1610        else
1611            return CRYPT_SUCCESS;
1612    }
1613    #endif //% TMP_ALG_SM2
```

B.11.3.2.25. _cpri__ValidateSignatureEcc()

This function validates

Return Value	Meaning
CRYPT_SUCCESS	signature is valid
CRYPT_FAIL	not a valid signature
CRYPT_SCHEME	unsupported scheme or hash algorithm

```
1614    CRYPT_RESULT
1615    _cpri__ValidateSignatureEcc(
1616        TPM2B_ECC_PARAMETER     *rIn,       // IN: r component of the signature
1617        TPM2B_ECC_PARAMETER     *sIn,       // IN: s component of the signature
1618        TPM_ALG_ID               scheme,    // IN: the scheme selector
1619        TPM_ALG_ID               hashAlg,   // IN: the hash algorithm used (not
1620                                            //     used in all schemes)
1621        TPM_ECC_CURVE            curveId,   // IN: the curve used in the signature
1622                                            //     process
1623        TPMS_ECC_POINT          *Qin,       // IN: the public point of the key
1624        TPM2B                   *digest     // IN: the digest that was signed
1625    )
1626    {
1627        switch (scheme)
```

```
1628        {
1629            case TPM_ALG_ECDSA:
1630                return ValidateSignatureEcdsa(rIn, sIn, curveId, Qin, digest);
1631                break;
1632
1633    #ifdef  TPM_ALG_ECSCHNORR
1634            case TPM_ALG_ECSCHNORR:
1635                return ValidateSignatureEcSchnorr(rIn, sIn, hashAlg, curveId, Qin,
1636                                                  digest);
1637                break;
1638    #endif
1639
1640    #ifdef TPM_ALG_SM2
1641            case TPM_ALG_SM2:
1642                return ValidateSignatureSM2Dsa(rIn, sIn, curveId, Qin, digest);
1643    #endif
1644            default:
1645                break;
1646        }
1647        return CRYPT_SCHEME;
1648    }
1649    #if CC_ZGen_2Phase == YES //%
1650    #ifdef TPM_ALG_ECMQV
```

B.11.3.2.26. 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)

```
1651    static BOOL
1652    avf1(
1653        BIGNUM                  *bnX,           // IN/OUT: the reduced value
1654        BIGNUM                  *bnN            // IN: the order of the curve
1655        )
1656    {
1657    // compute f = 2^(ceil(ceil(log2(n)) / 2))
1658        int                      f = (BN_num_bits(bnN) + 1) / 2;
1659    // x' = 2^f + (x mod 2^f)
1660        BN_mask_bits(bnX, f);   // This is mod 2*2^f but it doesn't matter because
1661                                // the next operation will SET the extra bit anyway
1662        BN_set_bit(bnX, f);
1663        return TRUE;
1664    }
```

B.11.3.2.27. 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

```
1665    static CRYPT_RESULT
1666    C_2_2_MQV(
1667        TPMS_ECC_POINT        *outZ,         // OUT: the computed point
1668        TPM_ECC_CURVE          curveId,      // IN: the curve for the computations
1669        TPM2B_ECC_PARAMETER   *dsA,          // IN: static private TPM key
1670        TPM2B_ECC_PARAMETER   *deA,          // IN: ephemeral private TPM key
1671        TPMS_ECC_POINT        *QsB,          // IN: static public party B key
1672        TPMS_ECC_POINT        *QeB           // IN: ephemeral public party B key
1673        )
1674    {
1675        BN_CTX                  *context;
1676        EC_POINT                *pQeA = NULL;
1677        EC_POINT                *pQeB = NULL;
1678        EC_POINT                *pQsB = NULL;
1679        EC_GROUP                *group = NULL;
1680        BIGNUM                  *bnTa;
1681        BIGNUM                  *bnDeA;
1682        BIGNUM                  *bnDsA;
1683        BIGNUM                  *bnXeA;         // x coordinate of ephemeral party A key
1684        BIGNUM                  *bnH;
1685        BIGNUM                  *bnN;
1686        BIGNUM                  *bnXeB;
1687        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1688        CRYPT_RESULT            retVal;
1689
1690        pAssert(    curveData != NULL && outZ != NULL && dsA != NULL
1691                &&        deA != NULL &&  QsB != NULL && QeB != NULL);
1692
1693        context = BN_CTX_new();
1694        if(context == NULL || curveData == NULL)
1695            FAIL(FATAL_ERROR_ALLOCATION);
1696        BN_CTX_start(context);
1697        bnTa = BN_CTX_get(context);
1698        bnDeA = BN_CTX_get(context);
1699        bnDsA = BN_CTX_get(context);
1700        bnXeA = BN_CTX_get(context);
1701        bnH = BN_CTX_get(context);
1702        bnN = BN_CTX_get(context);
1703        bnXeB = BN_CTX_get(context);
1704        if(bnXeB == NULL)
1705            FAIL(FATAL_ERROR_ALLOCATION);
1706
1707    // Process:
1708    //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
1709    //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
1710    //  3. If P = O, output an error indicator.
1711    //  4. Z=xP, where xP is the x-coordinate of P.
1712
1713        // Initialize group parameters and local values of input
1714        if((group = EccCurveInit(curveId, context)) == NULL)
1715            FAIL(FATAL_ERROR_INTERNAL);
1716
1717        if((pQeA = EC_POINT_new(group)) == NULL)
1718            FAIL(FATAL_ERROR_ALLOCATION);
1719
1720        BnFrom2B(bnDeA, &deA->b);
1721        BnFrom2B(bnDsA, &dsA->b);
1722        BnFrom2B(bnH, curveData->h);
1723        BnFrom2B(bnN, curveData->n);
```

```
1724        BnFrom2B(bnXeB, &QeB->x.b);
1725        pQeB = EccInitPoint2B(group, QeB, context);
1726        pQsB = EccInitPoint2B(group, QsB, context);
1727
1728        // Compute the public ephemeral key pQeA = [de,A]G
1729        if(     (retVal = PointMul(group, pQeA, bnDeA, NULL, NULL, context))
1730           !=  CRYPT_SUCCESS)
1731            goto Cleanup;
1732
1733        if(EC_POINT_get_affine_coordinates_GFp(group, pQeA, bnXeA, NULL, context) != 1)
1734                FAIL(FATAL_ERROR_INTERNAL);
1735
1736    //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
1737    // tA := (ds,A + de,A  avf(Xe,A)) mod n (3)
1738    // Compute 'tA' = ('deA' +  'dsA'  avf('XeA')) mod n
1739        // Ta = avf(XeA);
1740        BN_copy(bnTa, bnXeA);
1741        avf1(bnTa, bnN);
1742        if(// do Ta = ds,A * Ta mod n = dsA * avf(XeA) mod n
1743              !BN_mod_mul(bnTa, bnDsA, bnTa, bnN, context)
1744
1745           // now Ta = deA + Ta mod n =  deA + dsA * avf(XeA) mod n
1746           || !BN_mod_add(bnTa, bnDeA, bnTa, bnN, context)
1747          )
1748                FAIL(FATAL_ERROR_INTERNAL);
1749
1750    //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
1751    // Put this in because almost every case of h is == 1 so skip the call when
1752        // not necessary.
1753        if(!BN_is_one(bnH))
1754        {
1755            // Cofactor is not 1 so compute Ta := Ta * h mod n
1756            if(!BN_mul(bnTa, bnTa, bnH, context))
1757                FAIL(FATAL_ERROR_INTERNAL);
1758        }
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, (UINT16)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    }
1791    #endif // TPM_ALG_ECMQV
1792    #ifdef TPM_ALG_SM2 //%
```

B.11.3.2.28. 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.

```
1793    static BOOL
1794    avfSm2(
1795        BIGNUM                  *bnX,           // IN/OUT: the reduced value
1796        BIGNUM                  *bnN            // IN: the order of the curve
1797        )
1798    {
1799    // a) set w := ceil(ceil(log2(n)) / 2) - 1
1800        int                      w = ((BN_num_bits(bnN) + 1) / 2) - 1;
1801
1802    // b) set x' := 2^w + ( x & (2^w - 1))
1803    // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
1804        BN_mask_bits(bnX, w);   // as wiht avf1, this is too big by a factor of 2 but
1805                                // it doesn't matter becasue we SET the extra bit anyway
1806        BN_set_bit(bnX, w);
1807        return TRUE;
1808    }
```

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

```
1809    static CRYPT_RESULT
1810    SM2KeyExchange(
1811        TPMS_ECC_POINT          *outZ,          // OUT: the computed point
1812        TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
1813        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1814        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1815        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1816        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1817        )
1818    {
1819        BN_CTX                  *context;
1820        EC_POINT                *pQeA = NULL;
1821        EC_POINT                *pQeB = NULL;
1822        EC_POINT                *pQsB = NULL;
1823        EC_GROUP                *group = NULL;
1824        BIGNUM                  *bnTa;
1825        BIGNUM                  *bnDeA;
1826        BIGNUM                  *bnDsA;
1827        BIGNUM                  *bnXeA;         // x coordinate of ephemeral party A key
1828        BIGNUM                  *bnH;
1829        BIGNUM                  *bnN;
```

```
1830        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
1898        // Now that 'tA' is (h * 'tA' mod n)
1899        // 'outZ' = ['tA'](QsB + [avf(QeB.x)](QeB)).
1900
1901        // first, compute XeB = avf(XeB)
1902        avfSm2(bnXeB, bnN);
1903
1904        // QeB := [XeB]QeB
1905        if(     !EC_POINT_mul(group, pQeB, NULL, pQeB, bnXeB, context)
1906
1907            // QeB := QsB + QeB
1908            ||  !EC_POINT_add(group, pQeB, pQeB, pQsB, context)
1909           )
1910            FAIL(FATAL_ERROR_INTERNAL);
1911
1912        // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
1913        if(PointMul(group, pQeB, NULL, pQeB, bnTa, context) == CRYPT_SUCCESS)
1914            // Convert BIGNUM E to TPM2B E
1915            Point2B(group, outZ, pQeB, (UINT16)BN_num_bytes(bnN), context);
1916
1917    Cleanup:
1918        if(pQeA != NULL) EC_POINT_free(pQeA);
1919        if(pQeB != NULL) EC_POINT_free(pQeB);
1920        if(pQsB != NULL) EC_POINT_free(pQsB);
1921        if(group != NULL) EC_GROUP_free(group);
1922        BN_CTX_end(context);
1923        BN_CTX_free(context);
1924
1925        return retVal;
1926
1927    }
1928    #endif   //% TPM_ALG_SM2
```

B.11.3.2.29. 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).

```
1929    static CRYPT_RESULT
1930    C_2_2_ECDH(
1931        TPMS_ECC_POINT          *outZ1,         // OUT: Zs
1932        TPMS_ECC_POINT          *outZ2,         // OUT: Ze
1933        TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
1934        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1935        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1936        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1937        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1938        )
1939    {
1940        BN_CTX                  *context;
1941        EC_POINT                *pQ = NULL;
1942        EC_GROUP                *group = NULL;
1943        BIGNUM                  *bnD;
1944        UINT16                   size;
1945        const ECC_CURVE_DATA    *curveData = GetCurveData(curveId);
1946
1947        context = BN_CTX_new();
1948        if(context == NULL || curveData == NULL)
1949            FAIL(FATAL_ERROR_ALLOCATION);
1950        BN_CTX_start(context);
1951        if((bnD = BN_CTX_get(context)) == NULL)
1952            FAIL(FATAL_ERROR_INTERNAL);
1953
```

```
1954        // Initialize group parameters and local values of input
1955        if((group = EccCurveInit(curveId, context)) == NULL)
1956            FAIL(FATAL_ERROR_INTERNAL);
1957        size = (UINT16)BN_num_bytes(&group->order);
1958
1959        // Get the static private key of A
1960        BnFrom2B(bnD, &dsA->b);
1961
1962        // Initialize the static public point from B
1963        pQ = EccInitPoint2B(group, QsB, context);
1964
1965        // Do the point multiply for the Zs value
1966        if(PointMul(group, pQ, NULL, pQ, bnD, context) != CRYPT_NO_RESULT)
1967            // Convert the Zs value
1968            Point2B(group, outZ1, pQ, size, context);
1969
1970        // Get the ephemeral private key of A
1971        BnFrom2B(bnD, &deA->b);
1972
1973        // Initalize the ephemeral public point from B
1974        PointFrom2B(group, pQ, QeB, context);
1975
1976        // Do the point multiply for the Ze value
1977        if(PointMul(group, pQ, NULL, pQ, bnD, context) != CRYPT_NO_RESULT)
1978            // Convert the Ze value.
1979            Point2B(group, outZ2, pQ, size, context);
1980
1981        if(pQ != NULL) EC_POINT_free(pQ);
1982        if(group != NULL) EC_GROUP_free(group);
1983        BN_CTX_end(context);
1984        BN_CTX_free(context);
1985        return CRYPT_SUCCESS;
1986    }
```

B.11.3.2.30. _cpri__C_2_2_KeyExchange()

This function is the dispatch routine for the EC key exchange funtions that use two ephemeral and two static keys.

Return Value	Meaning
CRYPT_SCHEME	scheme is not defined

```
1987    CRYPT_RESULT
1988    _cpri__C_2_2_KeyExchange(
1989        TPMS_ECC_POINT          *outZ1,         // OUT: a comptuted computed point
1990        TPMS_ECC_POINT          *outZ2,         // OUT: and optional second point
1991        TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
1992        TPM_ALG_ID               scheme,        // IN: the key exchange scheme
1993        TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
1994        TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
1995        TPMS_ECC_POINT          *QsB,           // IN: static public party B key
1996        TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
1997        )
1998    {
1999        pAssert(   outZ1 != NULL
2000                && dsA != NULL && deA != NULL
2001                && QsB != NULL && QeB != NULL);
2002
2003        // Initalize the output points so that they are empty until one of the
2004        // functions decides otherwise
2005        outZ1->x.b.size = 0;
2006        outZ1->y.b.size = 0;
2007        if(outZ2 != NULL)
```

```
2008        {
2009            outZ2->x.b.size = 0;
2010            outZ2->y.b.size = 0;
2011        }
2012
2013        switch (scheme)
2014        {
2015            case TPM_ALG_ECDH:
2016                return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
2017                break;
2018    #ifdef  TPM_ALG_ECMQV
2019            case TPM_ALG_ECMQV:
2020                return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
2021                break;
2022    #endif
2023    #ifdef  TPM_ALG_SM2
2024            case TPM_ALG_SM2:
2025                return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
2026                break;
2027    #endif
2028            default:
2029                return CRYPT_SCHEME;
2030        }
2031    }
2032    #else       //%
```

Stub used when the 2-phase key exchagne is not defined so that the linker has something to associate with the value in the . def file.

```
2033    CRYPT_RESULT
2034    _cpri__C_2_2_KeyExchange()
2035    {
2036        return CRYPT_FAIL;
2037    }
2038    #endif //% CC_ZGen_2Phase
```

Annex C
(informative)
Simulation Environment

C.1 Introduction

These files are used to simulate some of the implementation-dependent hardware of a TPM. These files are provided to allow creation of a simulation environment for the TPM. These files are not expected to be part of a hardware TPM implementation.

C.2 Cancel.c

C.2.1. Introduction

This module simulates the cancel pins on the TPM.

C.2.2. Includes, Typedefs, Structures, and Defines

```
1   #include "PlatformData.h"
```

C.2.3. Functions

C.2.3.1. _plat__IsCanceled()

Check if the cancel flag is set

Return Value	Meaning
TRUE	if cancel flag is set
FALSE	if cancel flag is not set

```
2   BOOL
3   _plat__IsCanceled(void)
4   {
5       // return cancel flag
6       return s_isCanceled;
7   }
```

C.2.3.2. _plat__SetCancel()

Set cancel flag.

```
8   void
9   _plat__SetCancel(void)
10  {
11      s_isCanceled = TRUE;
12      return;
13  }
```

C.2.3.3. _plat__ClearCancel()

Clear cancel flag

 Copyright © TCG 2006-2013

DRAFT

```
14  void
15  _plat__ClearCancel( void)
16  {
17      s_isCanceled = FALSE;
18      return;
19  }
```

C.3 Clock.c

C.3.1. Introduction

This file contains the routines that are used by the simulator to mimic a hardware clock on a TPM. In this implementation, all the time values are measured in millisecond. However, the precision of the clock functions may be implementation dependent.

C.3.2. Includes and Data Definitions

```
#include <time.h>
#include "TpmError.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

```
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

```
unsigned long long
_plat__ClockTimeElapsed(void)
{
    unsigned long long  elapsed;
```

```
27      unsigned long long  currentClock = clock();
28      elapsed = ((currentClock - s_initClock) * CLOCK_NOMINAL) / s_adjustRate;
29      s_initClock += (elapsed * s_adjustRate) / CLOCK_NOMINAL;
30
31  #ifdef  DEBUGGING_TIME
32      // Put this in so that TPM time will pass much faster than real time when
33      // doing debug.
34      // A value of 1000 for DEBUG_TIME_MULTIPLER will make each ms into a second
35      // A good value might be 100
36      elapsed *= DEBUG_TIME_MULTIPLIER
37  #endif
38                  return elapsed;
39  }
```

C.3.3.4. _plat__ClockAdjustRate()

Adjust the clock rate

```
40  void
41  _plat__ClockAdjustRate(
42      int         adjust              // IN: the adjust number.  It could be
43                                      //     positive or negative
44  )
45  {
46      // We expect the caller should only use a fixed set of constant values to
47      // adjust the rate
48      switch(adjust)
49      {
50          case CLOCK_ADJUST_COARSE:
51              s_adjustRate += CLOCK_ADJUST_COARSE;
52              break;
53          case -CLOCK_ADJUST_COARSE:
54              s_adjustRate -= CLOCK_ADJUST_COARSE;
55              break;
56          case CLOCK_ADJUST_MEDIUM:
57              s_adjustRate += CLOCK_ADJUST_MEDIUM;
58              break;
59          case -CLOCK_ADJUST_MEDIUM:
60              s_adjustRate -= CLOCK_ADJUST_MEDIUM;
61              break;
62          case CLOCK_ADJUST_FINE:
63              s_adjustRate += CLOCK_ADJUST_FINE;
64              break;
65          case -CLOCK_ADJUST_FINE:
66              s_adjustRate -= CLOCK_ADJUST_FINE;
67              break;
68          default:
69              pAssert(FALSE);
70              break;
71      }
72
73
74      if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
75          s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
76      if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
77          s_adjustRate = CLOCK_NOMINAL-CLOCK_ADJUST_LIMIT;
78
79      return;
80  }
```

C.4 Entropy.c

C.4.1. Includes

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <stdint.h>
#include <memory.h>
#include "TpmError.h"
```

C.4.2. Local values

This is the last 32-bits of hadware 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.

```
uint32_t        lastEntropy;
int             firstValue;
```

C.4.3. _plat__GetEntropy()

This funtion is used to get available hardare 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	hardare failure of the entropy generator, this is sticky
>= 0	the returned amount of entropy (bytes)

```
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);
```

 Copyright © TCG 2006-2013

```
33          }
34          if(OK)
35          {
36              lastEntropy = rndNum;
37              if(amount > sizeof(rndNum))
38                  amount = sizeof(rndNum);
39              memcpy(entropy, &rndNum, amount);
40          }
41          return (OK) ? (int32_t)amount : -1;
42      }
```

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 stucture The locality can be 0-4 or 32-255. 5-31 is not allowed.

```
unsigned char
_plat__LocalityGet(void)
{
    return s_locality;
}
```

C.5.2.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
void
_plat__LocalitySet(
    unsigned char   locality
)
{
    pAssert(locality <= 4 || locality > 31);
    s_locality = locality;
    return;
}
```

C.5.2.3. _plat__IsRsaKeyCacheEnabled()

This funciton is used to check if the RSA key cache is enabled or not.

```
int
_plat__IsRsaKeyCacheEnabled(
    void
    )
{
    return s_RsaKeyCacheEnabled;
}
```

DRAFT

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"
```

C.6.3. Functions

C.6.3.1. _plat__NVEnable()

Enable NV memory

Return Value	Meaning
0	if success
non-0	if fail

```
int
_plat__NVEnable(
    void    *platParameter              // IN: platform specific parameters
)
{
    platParameter = 0;      // to try to satisfy the compiler and remove warning

#ifdef FILE_BACKED_NV

    if(s_NVFile != NULL) return 0;

    // Try to open an exist NVChip file for read/write
    if(0 != fopen_s(&s_NVFile, "NVChip", "r+b"))
        s_NVFile = NULL;

    if(NULL != s_NVFile)
    {
        // See if the NVChip file is empty
        fseek(s_NVFile, 0, SEEK_END);
        if(0 == ftell(s_NVFile))
            s_NVFile = NULL;
    }

    if(s_NVFile == NULL)
    {
        // Initialize all the byte in the new file to 0
        memset(s_NV, 0, NV_MEMORY_SIZE);

        // If NVChip file does not exist, try to create it for read/write
        fopen_s(&s_NVFile, "NVChip", "w+b");
        // Start initialize at the end of new file
        fseek(s_NVFile, 0, SEEK_END);
        // Write 0s to NVChip file
        fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);
```

```
39          }
40          else
41          {
42              // If NVChip file exist, assume the size is correct
43              fseek(s_NVFile, 0, SEEK_END);
44              pAssert(ftell(s_NVFile) == NV_MEMORY_SIZE);
45              // read NV file data to memory
46              fseek(s_NVFile, 0, SEEK_SET);
47              fread(s_NV, NV_MEMORY_SIZE, 1, s_NVFile);
48          }
49      #endif
50
51          return 0;
52      }
```

C.6.3.2. _plat__NVDisable()

Disable NV memory

```
53      void
54      _plat__NVDisable(void)
55      {
56      #ifdef  FILE_BACKED_NV
57
58          pAssert(s_NVFile != NULL);
59          // Close NV file
60          fclose(s_NVFile);
61          // Set file handle to NULL
62          s_NVFile = NULL;
63
64      #endif
65
66          return;
67      }
```

C.6.3.3. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
68      int
69      _plat__IsNvAvailable(void)
70      {
71
72          if(s_NvIsAvailable == FALSE)
73              return 1;
74
75      #ifdef FILE_BACKED_NV
76          if(s_NVFile == NULL)
77              return 1;
78      #endif
79
80          return 0;
81
82      }
```

 Copyright © TCG 2006-2013

DRAFT

C.6.3.4. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```
83  void
84  _plat__NvMemoryRead(
85      unsigned int        startOffset,        // IN: read start
86      unsigned int        size,               // IN: size of bytes to read
87      void                *data               // OUT: data buffer
88  )
89  {
90      pAssert(startOffset + size <= NV_MEMORY_SIZE);
91
92      // Copy data from RAM
93      memcpy(data, &s_NV[startOffset], size);
94      return;
95  }
```

C.6.3.5. _plat__NvIsDifferent()

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE	the NV location is different from the test value
FALSE	the NV location is the same as the test value

```
96  BOOL
97  _plat__NvIsDifferent(
98      unsigned int         startOffset,         // IN: read start
99      unsigned int         size,               // IN: size of bytes to read
100     void                *data                // IN: data buffer
101     )
102 {
103     return (memcmp(&s_NV[startOffset], data, size) != 0);
104 }
```

C.6.3.6. _plat__NvMemoryWrite()

This function is used to update NV memory. The **write** is to a memory copy of NV. At the end of the current command, any changes are written to the actual NV memory.

```
105 void
106 _plat__NvMemoryWrite(
107     unsigned int        startOffset,        // IN: write start
108     unsigned int        size,               // IN: size of bytes to write
109     void                *data               // OUT: data buffer
110 )
111 {
112     pAssert(startOffset + size <= NV_MEMORY_SIZE);
113
114     // Copy the data to the NV image
115     memcpy(&s_NV[startOffset], data, size);
116 }
```

 Copyright © TCG 2006-2013

C.6.3.7. _plat__NvMemoryMove()

Function: Move a chunk of NV memory from source to destination This function should ensure that if there overlap, the original data is copied before it is written

```
117  void
118  _plat__NvMemoryMove(
119      unsigned int        sourceOffset,         // IN: source offset
120      unsigned int        destOffset,           // IN: destination offset
121      unsigned int        size                  // IN: size of data being moved
122  )
123  {
124      pAssert(sourceOffset + size <= NV_MEMORY_SIZE);
125      pAssert(destOffset + size <= NV_MEMORY_SIZE);
126
127
128      // Move data in RAM
129      memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);
130
131      return;
132  }
```

C.6.3.8. _plat__NvCommit()

Update NV chip

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
133  int
134  _plat__NvCommit(void)
135  {
136  #ifdef FILE_BACKED_NV
137      // If NV file is not available, return failure
138      if(s_NVFile == NULL || s_NvIsAvailable == FALSE)
139          return 1;
140
141      // Write RAM data to NV
142      fseek(s_NVFile, 0, SEEK_SET);
143      fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);
144      return 0;
145  #else
146      return 0;
147  #endif
148
149  }
```

C.6.3.9. _plat__SetNvAvail()

Set the current NV state to available. This function is for testing purpose only. It is not part of the platform NV logic

```
150  void
151  _plat__SetNvAvail(void)
152  {
153      s_NvIsAvailable = TRUE;
154      return;
155  }
```

C.6.3.10. _plat__ClearNvAvail()

Set the current NV state to unavailable. This function is for testing purpose only. It is not part of the platform NV logic

```
156  void
157  _plat__ClearNvAvail(void)
158  {
159      s_NvIsAvailable = FALSE;
160      return;
161  }
```

C.7 PowerPlat.c

C.7.1. Includes and Function Prototypes

```
#include    "PlatformData.h"
#include    "Platform.h"
```

Platform power on and off functions

C.7.2. Functions

C.7.2.1. _plat__Signal_PowerOn()

Signal platform power on

```
void
_plat__Signal_PowerOn(void)
{
    // Start clock
    _plat__ClockReset();
    // Prepare NV memory for power on
    _plat__NVEnable(0);

    return;
}
```

C.7.2.2. _plat__Signal_PowerOff()

Signal platform power off

```
void
_plat__Signal_PowerOff(void)
{
    // Prepare NV memory for power off
    _plat__NVDisable();

    return;
}
```

DRAFT

C.8 Platform.h

```
#ifndef     PLATFORM_H
#define     PLATFORM_H
```

C.8.1. Includes

```
#include "bool.h"
#include "stdint.h"
```

C.8.2. Power Functions

C.8.2.1. _plat__Signal_PowerOn

Signal power on This signal is simulate by a RPC call

```
void
_plat__Signal_PowerOn(void);
```

C.8.2.2. _plat__Signal_PowerOff()

Signal power off This signal is simulate by a RPC call

```
void
_plat__Signal_PowerOff(void);
```

C.8.3. Physical Presence Functions

C.8.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

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

```
BOOL
_plat__PhysicalPresenceAsserted(void);
```

C.8.3.2. _plat__Signal_PhysicalPresenceOn

Signal physical presence on This signal is simulate by a RPC call

```
void
_plat__Signal_PhysicalPresenceOn(void);
```

C.8.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off This signal is simulate by a RPC call

```
void
_plat__Signal_PhysicalPresenceOff(void);
```

C.8.4. Command Canceling Functions

C.8.4.1. _plat__IsCanceled()

Check if the cancel flag is set

Return Value	Meaning
TRUE	if cancel flag is set
FALSE	if cancel flag is not set

```
15  BOOL
16  _plat__IsCanceled(void);
```

C.8.4.2. _plat__SetCancel()

Set cancel flag.

```
17  void
18  _plat__SetCancel(void);
```

C.8.4.3. _plat__ClearCancel()

Clear cancel flag

```
19  void
20  _plat__ClearCancel( void);
```

C.8.5. NV memory functions

C.8.5.1. _plat__NVEnable()

Enable platform NV memory NV memory is automatically enabled at power on event. This function is mostly for TPM_Manufacture() to access NV memory without a power on event

Return Value	Meaning
0	if success
non-0	if fail

```
21  int
22  _plat__NVEnable(
23      void    *platParameter              // IN: platform specific parameters
24  );
```

C.8.5.2. _plat__NVDisable()

Disable platform NV memory NV memory is automatically disabled at power off event. This function is mostly for TPM_Manufacture() to disable NV memory without a power off event

```
25  void
26  _plat__NVDisable(void);
```

 Copyright © TCG 2006-2013

C.8.5.3. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
27  int
28  _plat__IsNvAvailable(void);
```

C.8.5.4. _plat__NvCommit()

Update NV chip

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
29  int
30  _plat__NvCommit(void);
```

C.8.5.5. _plat__NvMemoryRead()

Read a chunk of NV memory

```
31  void
32  _plat__NvMemoryRead(
33      unsigned int        startOffset,        // IN: read start
34      unsigned int        size,               // IN: size of bytes to read
35      void                *data               // OUT: data buffer
36  );
```

C.8.5.6. _plat__NvIsDifferent()

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE	the NV location is different from the test value
FALSE	the NV location is the same as the test value

```
37  BOOL
38  _plat__NvIsDifferent(
39      unsigned int         startOffset,        // IN: read start
40      unsigned int         size,               // IN: size of bytes to compare
41      void                *data                // IN: data buffer
42      );
```

C.8.5.7. _plat__NvMemoryWrite()

Write a chunk of NV memory

```
43  void
44  _plat__NvMemoryWrite(
45      unsigned int        startOffset,        // IN: read start
46      unsigned int        size,               // IN: size of bytes to read
47      void                *data               // OUT: data buffer
48  );
```

C.8.5.8. _plat__NvMemoryMove()

Move a chunk of NV memory from source to destination This function should ensure that if there overlap, the original data is copied before it is written

```
49  void
50  _plat__NvMemoryMove(
51      unsigned int        sourceOffset,        // IN: source offset
52      unsigned int        destOffset,          // IN: destination offset
53      unsigned int        size                 // IN: size of data being moved
54  );
```

C.8.5.9. _plat__SetNvAvail()

Set the current NV state to available. This function is for testing purposes only. It is not part of the platform NV logic

```
55  void
56  _plat__SetNvAvail(void);
```

C.8.5.10. _plat__ClearNvAvail()

Set the current NV state to unavailable. This function is for testing purposes only. It is not part of the platform NV logic

```
57  void
58  _plat__ClearNvAvail(void);
```

C.8.6. Locality Functions

C.8.6.1. _plat__LocalityGet()

Get the most recent command locality in locality value form

```
59  unsigned char
60  _plat__LocalityGet(void);
```

C.8.6.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
61  void
62  _plat__LocalitySet(
63      unsigned char    locality
64  );
```

DRAFT

C.8.6.3. _plat__IsRsaKeyCacheEnabled()

This funciton is used to check if the RSA key cache is enabled or not.

```
65  int
66  _plat__IsRsaKeyCacheEnabled(
67      void
68      );
```

C.8.7. Clock Constants and Functions

Assume that the nominal divisor is 30000

```
69  #define     CLOCK_NOMINAL           30000
```

A 1% change in rate is 300 counts

```
70  #define     CLOCK_ADJUST_COARSE     300
```

A . 1 change in rate is 30 counts

```
71  #define     CLOCK_ADJUST_MEDIUM     30
```

A minimum change in rate is 1 count

```
72  #define     CLOCK_ADJUST_FINE       1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16. 7%)

```
73  #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

```
74  void
75  _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.

```
76  unsigned long long
77  _plat__ClockTimeFromStart(
78      void
79      );
```

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

```
80  unsigned long long
```

```
81  _plat__ClockTimeElapsed(void);
```

C.8.7.4. _plat__ClockAdjustRate()

Adjust the clock rate

```
82  void
83  _plat__ClockAdjustRate(
84      int         adjust              // IN: the adjust number.  It could be
85                                      // positive or negative
86      );
```

C.8.8. Single Function Files

C.8.8.1. _plat__GetEntropy()

This funtion is used to get available hardare 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)

```
87  int32_t
88  _plat__GetEntropy(
89      unsigned char       *entropy,           // output buffer
90      uint32_t             amount             // amount requested
91  );
```

C.8.8.2. _plat__TpmFail()

Put TPM in failure mode.

```
92  int
93  _plat__TpmFail(
94      const char      *function,
95      int              line,
96      int              code);
97  #endif
```

 Copyright © TCG 2006-2013

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    "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;
```

From PPPlat.c Physical presence. It is initialized to FALSE

```
extern BOOL     s_physicalPresence;
#endif // _PLATFORM_DATA_H_
```

 Copyright © TCG 2006-2013

C.10 PlatformData.c

C.10.1. Description

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

C.10.2. Includes

This include is required to set the NV memory size consistently across all parts of the implementation.

```
1   #include    "Implementation.h"
2   #include    "Platform.h"
3   #include    "PlatformData.h"
```

From Cancel.c

```
4   BOOL     s_isCanceled = FALSE;
```

From Clock.c

```
5   unsigned long long   s_initClock = 0;
6   unsigned int         s_adjustRate = CLOCK_NOMINAL;
```

From LocalityPlat.c

```
7   unsigned char s_locality = 0;
```

From NVMem.c

```
8   #ifdef FILE_BACKED_NV
9   FILE             *s_NVFile = NULL;
10  #endif
11  unsigned char     s_NV[NV_MEMORY_SIZE];
12  BOOL              s_NvIsAvailable = TRUE;
```

From PPPlat.c

```
13  BOOL  s_physicalPresence;
```

DRAFT

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

```
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

```
void
_plat__Signal_PhysicalPresenceOn(void)
{
    s_physicalPresence = TRUE;
    return;
}
```

C.11.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
void
_plat__Signal_PhysicalPresenceOff( void)
{
    s_physicalPresence = FALSE;
    return;
}
```

C.12 TpmFail.c

C.12.1. Description

This file contains the function that is called when the TPM experiences a fatal error. This function is stubbed out. It should be replaced with a function that will save the calling parameters so that they may be returned on a subsequent TPM2_GetTestResult(). The function should then clean the stack (as much as possible), set the flag to indicate that the TPM is in failure mode, and return TPM_RC_FAIL.

```
1   #include "assert.h"
2   #define UNREFERNCED_PARAMETER(param)     (void)param
```

C.12.2. _plat__TpmFail()

```
3   int
4   _plat__TpmFail(
5       const char* function,
6       int line,
7       int code
8   )
9   {
10      UNREFERNCED_PARAMETER(function);
11      UNREFERNCED_PARAMETER(line);
12      UNREFERNCED_PARAMETER(code);
13
14      assert(0);
15      return 0;
16  }
```

DRAFT

Annex D
(informative)
Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.

 Copyright © TCG 2006-2013

D.2 TpmTcpProtocol.h

```
#ifndef     TCP_TPM_PROTOCOL_H
#define     TCP_TPM_PROTOCOL_H
```

D.2.1. Introduction

TPM commands are communicated as BYTE streams on a TCP connection. The TPM command protocol is enveloped with the interface protocol described in this file. The command is indicated by a UINT32 with one of the values below. Most commands take no parameters return no TPM errors. In these cases the TPM interface protocol acknowledges that command processing is complete by returning a UINT32=0. The command TPM_SIGNAL_HASH_DATA takes a UINT32-prepended variable length BYTE array and the interface protocol acknowledges command completion with a UINT32=0. Most TPM commands are enveloped using the TPM_SEND_COMMAND interface command. The parameters are as indicated below. The interface layer also appends a UIN32=0 to the TPM response for regularity.

D.2.2. Typedefs

TPM Commands. All commands acknowledge processing by returning a UINT32 == 0 except where noted

```
#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_SESSION_END             20
#define TPM_STOP                    21
enum TpmEndPointInfo
{
    tpmPlatformAvailable = 0x01,
    tpmUsesTbs = 0x02,
    tpmInRawMode = 0x04,
    tpmSupportsPP = 0x08
};

// Existing RPC interface type definitions retained so that the implementation
// can be re-used
typedef struct
{
    unsigned long BufferSize;
    unsigned char *Buffer;
} _IN_BUFFER;

typedef unsigned char *_OUTPUT_BUFFER;

```

DRAFT

```
42  typedef struct
43  {
44      unsigned long BufferSize;
45      _OUTPUT_BUFFER Buffer;
46  } _OUT_BUFFER;
47
48  //** TPM Command Function Prototypes
49  void _rpc__Signal_PowerOn();
50  void _rpc__Signal_PowerOff();
51  void _rpc__Signal_PhysicalPresenceOn();
52  void _rpc__Signal_PhysicalPresenceOff();
53  void _rpc__Signal_Hash_Start();
54  void _rpc__Signal_Hash_Data(
55      _IN_BUFFER input
56  );
57  void _rpc__Signal_HashEnd();
58  void _rpc__Send_Command(
59      unsigned char   locality,
60      _IN_BUFFER      request,
61      _OUT_BUFFER     *response
62  );
63  void _rpc__Signal_CanCelOn();
64  void _rpc__Signal_CanCelOff();
65  void _rpc__Signal_NvOn();
66  void _rpc__Signal_NvOff();
67  void _rpc__RsaKeyCacheControl(int);
68  BOOL _rpc__InjectEPS(
69      const char* seed,
70      int seedSize
71  );
```

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.

```
72  BOOL TpmServer(SOCKET ServerSocket);
73  #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 <stdio.h>
#include "TpmTcpProtocol.h"
BOOL ReadBytes(SOCKET s, char* buffer, int NumBytes);
BOOL ReadVarBytes(SOCKET s, char* buffer, UINT32* BytesReceived, int MaxLen);
BOOL WriteBytes(SOCKET s, char* buffer, int NumBytes);
BOOL WriteUINT32(SOCKET s, UINT32 val);
static UINT32 ServerVersion = 1;
```

The input and output data buffers for the simulator.

```
#define MAX_BUFFER 1048576
char InputBuffer[MAX_BUFFER];
char OutputBuffer[MAX_BUFFER];
```

D.3.3. Functions

D.3.3.1. CreateSocket()

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);
    if(INVALID_SOCKET == *listenSocket)
    {
        printf("Cannot create server listen socket.  Error is 0x%x\n",
                WSAGetLastError());
        return -1;
    }
```

DRAFT

```
43
44      // bind the listening socket to the specified port
45      ZeroMemory(&MyAddress, sizeof(MyAddress));
46      MyAddress.sin_port=htons((short) PortNumber);
47      MyAddress.sin_family=AF_INET;
48
49      res= bind(*listenSocket,(struct sockaddr*) &MyAddress,sizeof(MyAddress));
50      if(res==SOCKET_ERROR)
51      {
52          printf("Bind error.  Error is 0x%x\n", WSAGetLastError());
53          return -1;
54      };
55
56      // listen/wait for server connections
57      res= listen(*listenSocket,3);
58      if(res==SOCKET_ERROR)
59      {
60          printf("Listen error.  Error is 0x%x\n", WSAGetLastError());
61          return -1;
62      };
63
64      return 0;
65  }
```

D.3.3.2. PlatformServer()

This function processes incoming platform requests.

```
66  BOOL
67  PlatformServer(
68      SOCKET              s
69      )
70  {
71      BOOL                ok = TRUE;
72      UINT32              length = 0;
73      UINT32              Command;
74
75      for(;;)
76      {
77          ok = ReadBytes(s, (char*) &Command, 4);
78          // client disconnected (or other error).  We stop processing this client
79          // and return to our caller who can stop the server or listen for another
80          // connection.
81          if(!ok) return TRUE;
82          Command = ntohl(Command);
83          switch(Command)
84          {
85              case TPM_SIGNAL_POWER_ON:
86                  _rpc__Signal_PowerOn();
87                  break;
88
89              case TPM_SIGNAL_POWER_OFF:
90                  _rpc__Signal_PowerOff();
91                  break;
92
93              case TPM_SIGNAL_PHYS_PRES_ON:
94                  _rpc__Signal_PhysicalPresenceOn();
95                  break;
96
97              case TPM_SIGNAL_PHYS_PRES_OFF:
98                  _rpc__Signal_PhysicalPresenceOff();
99                  break;
100
101             case TPM_SIGNAL_CANCEL_ON:
```

```
                _rpc__Signal_CanCelOn();
                break;

            case TPM_SIGNAL_CANCEL_OFF:
                _rpc__Signal_CanCelOff();
                break;

            case TPM_SIGNAL_NV_ON:
                _rpc__Signal_NvOn();
                break;

            case TPM_SIGNAL_NV_OFF:
                _rpc__Signal_NvOff();
                break;

            case TPM_SIGNAL_KEY_CACHE_ON:
                _rpc__RsaKeyCacheControl(TRUE);
                break;

            case TPM_SIGNAL_KEY_CACHE_OFF:
                _rpc__RsaKeyCacheControl(FALSE);
                break;

            case TPM_SESSION_END:
                // Client signaled end-of-session
                return TRUE;

            case TPM_STOP:
                // Client requested the simulator to exit
                return FALSE;

            default:
                printf("Unrecognized platform interface command %d\n", Command);
                WriteUINT32(s, 1);
                return TRUE;
        }
        WriteUINT32(s,0);
    }
    return FALSE;
}
```

D.3.3.3. PlatformSvcRoutine()

This function is called to set up the socket interfaces listen for commands.

```
DWORD WINAPI
PlatformSvcRoutine(
        LPVOID          port
        )
{
    int                 PortNumber = (int)(INT_PTR) port;
    SOCKET              listenSocket, serverSocket;
    struct              sockaddr_in HerAddress;
    int                 res;
    int                 length;
    BOOL                continueServing;

    res = CreateSocket(PortNumber, &listenSocket);
    if(res != 0)
    {
        printf("Create platform service socket fail\n");
        return res;
    }

```

```
161         // Loop accepting connections one-by-one until we are killed or asked to stop
162         // Note the platform service is single-threaded so we don't listen for a new
163         // connection until the prior connection drops.
164         do
165         {
166             printf("Platform server listening on port %d\n", PortNumber);
167 
168             // blocking accept
169             length = sizeof(HerAddress);
170             serverSocket = accept(listenSocket,
171                                   (struct sockaddr*) &HerAddress,
172                                   &length);
173             if(serverSocket == SOCKET_ERROR)
174             {
175                 printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
176                 return -1;
177             };
178             printf("Client accepted\n");
179 
180             // normal behavior on client disconnection is to wait for a new client
181             // to connect
182             continueServing = PlatformServer(serverSocket);
183             closesocket(serverSocket);
184         }
185         while(continueServing);
186 
187         return 0;
188     }
```

D.3.3.4. PlatformSignalService()

Start service for processing platform signals. This function starts a new thread waiting for platform signals. Platform signals are processed by a single thread in sequence.

```
189     int
190     PlatformSignalService(
191         int                 PortNumber
192         )
193     {
194         HANDLE              hPlatformSvc;
195         int                 ThreadId;
196         int                 port = PortNumber;
197 
198         // Create service thread for platform signals
199         hPlatformSvc = CreateThread(NULL, 0,
200                                     (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
201                                     (LPVOID) (INT_PTR) port, 0, (LPDWORD)&ThreadId);
202         if(hPlatformSvc == NULL)
203         {
204             printf("Thread Creation failed\n");
205             return -1;
206         }
207 
208         return 0;
209     }
```

D.3.3.5. RegularCommandService()

```
210     int
211     RegularCommandService(
212         int                 PortNumber
213         )
214     {
```

```
215         SOCKET              listenSocket;
216         SOCKET              serverSocket;
217         struct              sockaddr_in HerAddress;
218
219         int res, length;
220         BOOL continueServing;
221
222         res = CreateSocket(PortNumber, &listenSocket);
223         if(res != 0)
224         {
225             printf("Create platform service socket fail\n");
226             return res;
227         }
228
229         // Loop accepting connections one-by-one until we are killed or asked to stop
230         // Note the TPM command service is single-threaded so we don't listen for
231         // a new connection until the prior connection drops.
232         do
233         {
234             printf("TPM command server listening on port %d\n", PortNumber);
235
236             // blocking accept
237             length = sizeof(HerAddress);
238             serverSocket = accept(listenSocket,
239                                   (struct sockaddr*) &HerAddress,
240                                   &length);
241             if(serverSocket ==SOCKET_ERROR)
242             {
243                 printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
244                 return -1;
245             };
246             printf("Client accepted\n");
247
248             // normal behavior on client disconnection is to wait for a new client
249             // to connect
250             continueServing = TpmServer(serverSocket);
251             closesocket(serverSocket);
252         }
253         while(continueServing);
254
255         return 0;
256     }
```

D.3.3.6. StartTcpServer()

Main entry-point. The server listens on port specified. Note that there is no way to specify the network interface in this implementation.

```
257     int
258     StartTcpServer(
259         int                 PortNumber)
260     {
261         int                 res;
262
263         // Start Platform Signal Processing Service
264         res = PlatformSignalService(PortNumber+1);
265         if (res != 0)
266         {
267             printf("PlatformSignalService failed\n");
268             return res;
269         }
270
271         // Start Regular/DRTM TPM command service
272         res = RegularCommandService(PortNumber);
```

```
273         if (res != 0)
274         {
275             printf("RegularCommandService failed\n");
276             return res;
277         }
278 
279         return 0;
280     }
```

D.3.3.7. ReadBytes()

Read NumBytes() into buffer on indicated socket.

```
281     BOOL
282     ReadBytes(
283         SOCKET              s,
284         char               *buffer,
285         int                 NumBytes
286         )
287     {
288         int                 res;
289         int                 numGot = 0;
290 
291         while(numGot<NumBytes)
292         {
293             res = recv(s, buffer+numGot, NumBytes-numGot, 0);
294             if(res == -1)
295             {
296                 printf("Receive error.  Error is 0x%x\n", WSAGetLastError());
297                 return FALSE;
298             }
299             if(res==0)
300             {
301                 return FALSE;
302             }
303             numGot+=res;
304         }
305         return TRUE;
306     }
```

D.3.3.8. WriteBytes()

Send NumBytes() on indicated socket

```
307     BOOL
308     WriteBytes(
309         SOCKET              s,
310         char               *buffer,
311         int                 NumBytes
312         )
313     {
314         int                 res;
315         int                 numSent = 0;
316         while(numSent<NumBytes)
317         {
318             res = send(s, buffer+numSent, NumBytes-numSent, 0);
319             if(res == -1)
320             {
321                 if(WSAGetLastError() == 0x2745)
322                 {
323                     printf("Client disconnected\n");
324                 }
325                 else
```

```
326             {
327                 printf("Send error.  Error is 0x%x\n", WSAGetLastError());
328             }
329             return FALSE;
330         }
331         numSent+=res;
332     }
333     return TRUE;
334 }
```

D.3.3.9. WriteUINT32()

Send one byte containing hton(1)

```
335 BOOL
336 WriteUINT32(
337     SOCKET              s,
338     UINT32              val
339     )
340 {
341     UINT32 netVal = htonl(val);
342     return WriteBytes(s, (char*) &netVal, 4);
343 }
```

D.3.3.10. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order

```
344 BOOL
345 ReadVarBytes(
346     SOCKET              s,
347     char                *buffer,
348     UINT32              *BytesReceived,
349     int                 MaxLen
350     )
351 {
352     int                 length;
353     BOOL                res;
354 
355     res = ReadBytes(s, (char*) &length, 4);
356     if(!res) return res;
357     length = ntohl(length);
358     *BytesReceived = length;
359     if(length>MaxLen)
360     {
361         printf("Buffer too big.  Client says %d\n", length);
362         return FALSE;
363     }
364     if(length==0) return TRUE;
365     res = ReadBytes(s, buffer, length);
366     if(!res) return res;
367     return TRUE;
368 }
```

D.3.3.11. WriteVarBytes()

Send a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order

```
369 BOOL
370 WriteVarBytes(
371     SOCKET              s,
```

```
372         char                *buffer,
373         int                 BytesToSend
374         )
375     {
376         UINT32              netLength = htonl(BytesToSend);
377         BOOL res;
378
379         res = WriteBytes(s, (char*) &netLength, 4);
380         if(!res) return res;
381         res = WriteBytes(s, buffer, BytesToSend);
382         if(!res) return res;
383         return TRUE;
384     }
```

D.3.3.12. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```
385     BOOL
386     TpmServer(
387         SOCKET              s
388         )
389     {
390         UINT32              length;
391         UINT32              Command;
392         BYTE                locality;
393         BOOL                ok;
394         int                 result;
395         int                 clientVersion;
396         _IN_BUFFER          InBuffer;
397         _OUT_BUFFER         OutBuffer;
398
399         for(;;)
400         {
401             ok = ReadBytes(s, (char*) &Command, 4);
402             // client disconnected (or other error).  We stop processing this client
403             // and return to our caller who can stop the server or listen for another
404             // connection.
405             if(!ok)
406                 return TRUE;
407             Command = ntohl(Command);
408             switch(Command)
409             {
410                 case TPM_SIGNAL_HASH_START:
411                     _rpc__Signal_Hash_Start();
412                     break;
413
414                 case TPM_SIGNAL_HASH_END:
415                     _rpc__Signal_HashEnd();
416                     break;
417
418                 case TPM_SIGNAL_HASH_DATA:
419                     ok = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
420                     if(!ok) return TRUE;
421                     InBuffer.Buffer = (BYTE*) InputBuffer;
422                     InBuffer.BufferSize = length;
423                     _rpc__Signal_Hash_Data(InBuffer);
424                     break;
425
426                 case TPM_SEND_COMMAND:
427                     ok = ReadBytes(s, (char*) &locality, 1);
428                     if(!ok)
429                         return TRUE;
430
```

```
431                 ok = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
432                 if(!ok)
433                     return TRUE;
434                 InBuffer.Buffer = (BYTE*) InputBuffer;
435                 InBuffer.BufferSize = length;
436                 OutBuffer.BufferSize = MAX_BUFFER;
437                 OutBuffer.Buffer = (_OUTPUT_BUFFER) OutputBuffer;
438                 _rpc__Send_Command(locality, InBuffer, &OutBuffer);
439                 ok = WriteVarBytes(s,
440                                    (char*) OutBuffer.Buffer,
441                                    OutBuffer.BufferSize);
442                 if(!ok)
443                     return TRUE;
444                 break;
445 
446             case TPM_REMOTE_HANDSHAKE:
447                 ok = ReadBytes(s, (char*)&clientVersion, 4);
448                 if(!ok)
449                     return TRUE;
450                 if( clientVersion == 0 )
451                 {
452                     printf("Unsupported client version (0).\n");
453                     return TRUE;
454                 }
455                 ok &= WriteUINT32(s, ServerVersion);
456                 ok &= WriteUINT32(s,
457                                   tpmInRawMode | tpmPlatformAvailable | tpmSupportsPP);
458                 break;
459 
460             case TPM_SET_ALTERNATIVE_RESULT:
461                 ok = ReadBytes(s, (char*)&result, 4);
462                 if(!ok)
463                     return TRUE;
464                 // Alternative result is not applicable to the simulator.
465                 break;
466 
467             case TPM_SESSION_END:
468                 // Client signaled end-of-session
469                 return TRUE;
470 
471             case TPM_STOP:
472                 // Client requested the simulator to exit
473                 return FALSE;
474             default:
475                 printf("Unrecognized TPM interface command %d\n", Command);
476                 return TRUE;
477         }
478         ok = WriteUINT32(s,0);
479         if(!ok)
480             return TRUE;
481     }
482     return FALSE;
483 }
```

DRAFT

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

```
#include <stdlib.h>
#include <stdio.h>
#include "bool.h"
#include "TPMLib.h"
#include "platform.h"
```

Preclude inclusion of unnecessary simulator headers

```
#define _SWAP_H
#include <windows.h>
#include "TpmTcpProtocol.h"
static BOOL     s_isPowerOn = FALSE;
```

D.4.3. Functions

D.4.3.1. Signal_PowerOn()

Signal a power on event.

```
void
_rpc__Signal_PowerOn()
{
    if(s_isPowerOn) return;

    // Pass power on signal to platform
    _plat__Signal_PowerOn();

    // Pass power on signal to TPM
    _TPM_Init();

    // Set state as power on
    s_isPowerOn = TRUE;
}
```

D.4.3.2. Signal_PowerOff()

Signal a power off event.

```
void
_rpc__Signal_PowerOff()
{
    if(!s_isPowerOn) return;

    // Pass power off signal to platform
    _plat__Signal_PowerOff();

    s_isPowerOn = FALSE;
```

```
33
34      return;
35  }
```

D.4.3.3. _rpc__Signal_PhysicalPresenceOn()

Function to simulate activation of the physical presence **pin**.

```
36  void
37  _rpc__Signal_PhysicalPresenceOn()
38  {
39      // If TPM is power off, reject this signal
40      if(!s_isPowerOn) return;
41
42      // Pass physical presence on to platform
43      _plat__Signal_PhysicalPresenceOn();
44
45      return;
46  }
```

D.4.3.4. _rpc__Signal_PhysicalPresenceOff()

Function to simulate deactivation of the physical presence **pin**.

```
47  void
48  _rpc__Signal_PhysicalPresenceOff()
49  {
50      // If TPM is power off, reject this signal
51      if(!s_isPowerOn) return;
52
53      // Pass physical presence off to platform
54      _plat__Signal_PhysicalPresenceOff();
55
56      return;
57  }
```

D.4.3.5. _rpc__Signal_Hash_Start()

Function to simulate a _TPM_Hash_Start() event.

```
58  void
59  _rpc__Signal_Hash_Start()
60  {
61      // If TPM is power off, reject this signal
62      if(!s_isPowerOn) return;
63
64      // Pass _TPM_Hash_Start signal to TPM
65      Signal_Hash_Start();
66      return;
67  }
```

D.4.3.6. _rpc__Signal_Hash_Data()

Function to simulate a _TPM_Hash_Data() event.

```
68  void
69  _rpc__Signal_Hash_Data(
70      _IN_BUFFER input
71  )
72  {
```

```
73      // If TPM is power off, reject this signal
74      if(!s_isPowerOn) return;
75
76      // Pass _TPM_Hash_Data signal to TPM
77      Signal_Hash_Data(input.BufferSize, input.Buffer);
78      return;
79  }
```

D.4.3.7. _rpc__Signal_HashEnd()

Function to simulate a _TPM_Hash_End() event.

```
80  void
81  _rpc__Signal_HashEnd()
82  {
83      // If TPM is power off, reject this signal
84      if(!s_isPowerOn) return;
85
86      // Pass _TPM_HashEnd signal to TPM
87      Signal_Hash_End();
88      return;
89  }
```

D.4.3.8. _rpc__Send_Command()

This is the TPM command interface.

```
90  void
91  _rpc__Send_Command(
92      unsigned char   locality,
93      _IN_BUFFER      request,
94      _OUT_BUFFER     *response
95  )
96  {
97      // If TPM is power off, reject any commands.
98      if(!s_isPowerOn)
99      {
100         response->BufferSize = 0;
101         return;
102     }
103
104     // Set command locality.  Command locality is a signal rather than a part
105     // of TPM internal state.  So we always set the locality information even
106     // the command may fail
107     _plat__LocalitySet(locality);
108
109     // Call command execution
110     // response buffer space is provided by the called function.
111     ExecuteCommand(request.BufferSize, request.Buffer,
112                    &response->BufferSize, &response->Buffer);
113     if(response->BufferSize == 10 && response->Buffer[9] != 0)
114         response->Buffer[6] = 0;
115
116     return;
117
118 }
```

D.4.3.9. _rpc__Signal_CanCelOn()

Function to turn on the indication to cancel a command in process.

```
119  void
120  _rpc__Signal_CanCelOn()
121  {
122      // If TPM is power off, reject this signal
123      if(!s_isPowerOn) return;
124
125      // Set the platform canceling flag.
126      _plat__SetCancel();
127
128      return;
129  }
```

D.4.3.10. _rpc__Signal_CanCelOff()

Function to turn off the indication to cancel a command in process.

```
130  void
131  _rpc__Signal_CanCelOff()
132  {
133      // If TPM is power off, reject this signal
134      if(!s_isPowerOn) return;
135
136      // Set the platform canceling flag.
137      _plat__ClearCancel();
138
139      return;
140  }
```

D.4.3.11. _rpc__Signal_NvOn()

In a system where the NV memory used by the TPM is not within the TPM, the NV may not always be available. This function turns on the indicator that indicates that NV is available.

```
141  void
142  _rpc__Signal_NvOn()
143  {
144      // If TPM is power off, reject this signal
145      if(!s_isPowerOn) return;
146
147      _plat__SetNvAvail();
148      return;
149  }
```

D.4.3.12. _rpc__Signal_NvOff()

This function set the indication that NV memory is no longer available.

```
150  void
151  _rpc__Signal_NvOff()
152  {
153      // If TPM is power off, reject this signal
154      if(!s_isPowerOn) return;
155
156      _plat__ClearNvAvail();
157      return;
158  }
```

D.4.3.13. _rpc__RsaKeyCacheControl()

This function is used to enable/disable the use of the RSA key cache during simulaton.

 Copyright © TCG 2006-2013

DRAFT

```
159 void
160 _rpc__RsaKeyCacheControl(
161     int    state
162     )
163 {
164     _plat__RsaKeyCacheControl(state);
165 }
166 #if 1
```

D.4.3.14. _rpc__Shutdown()

This function is used to stop the TPM simulator.

```
167 void
168 _rpc__Shutdown()
169 {
170     RPC_STATUS status;
171
172     // Stop TPM
173     TPM_TearDown();
174
175     status = RpcMgmtStopServerListening(NULL);
176     if (status != RPC_S_OK)
177     {
178         printf_s("RpcMgmtStopServerListening returned: 0x%x\n", status);
179         exit(status);
180     }
181
182     status = RpcServerUnregisterIf(NULL, NULL, FALSE);
183     if (status != RPC_S_OK)
184     {
185         printf_s("RpcServerUnregisterIf returned 0x%x\n", status);
186         exit(status);
187     }
188 }
189 #endif //
```

 Copyright © TCG 2006-2013

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 <ctype.h>
#include <windows.h>
#include <strsafe.h>
#include "string.h"
#include "TpmTcpProtocol.h"
#define PURPOSE \
"TPM Reference Simulator.\nCopyright Microsoft 2010, 2011.\n"
#define DEFAULT_TPM_PORT 2321
void* MainPointer;
int TPM_Manufacture();
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()

Entry point for the simulator.

main: register the interface, start listening for clients

```
void __cdecl
main(int argc, char * argv[])
{
    int portNum = DEFAULT_TPM_PORT;
    if(argc>2)
    {
        Usage(argv[0]);
    }

```

```
38      if(argc==2)
39      {
40          if(strcmp(argv[1], "?") ==0)
41          {
42              Usage(argv[0]);
43          }
44          portNum = atoi(argv[1]);
45          if(portNum <=0 || portNum>65535)
46          {
47              Usage(argv[0]);
48          }
49      }
50      _plat__NVEnable(NULL);
51      if(TPM_Manufacture() != 0)
52      {
53          exit(RPC_S_INTERNAL_ERROR);
54      }
55      // Disable NV memory
56      _plat__NVDisable();
57
58      StartTcpServer(portNum);
59      return;
60  }
```