Trusted Platform Module Library

Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 01.38

September 29, 2016

Contact: admin@trustedcomputinggroup.org

TCG Published

Licenses and Notices

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.

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.

Disclaimers:

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

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

CONTENTS

Trusted Platform Module Library
Part 4: Supporting Routines

1 Scope

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

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

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

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

2 Terms and definitions

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

3 Symbols and abbreviated terms

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

4 Automation

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

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

The automation produces TPM_Types.h, a header representing TPM 2.0 Part 2. It also produces, for each major clause of Part 4, a header of the form _fp.h with the function prototypes.

EXAMPLE The header file for SessionProcess.c is SessionProcess_fp.h.

4.1 Configuration Parser

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

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

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

4.2 Structure Parser

4.2.1 Introduction

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

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

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

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

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

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

4.2.2 Unmarshaling Code Prototype

4.2.2.1 Simple Types and Structures

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

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

Where:

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

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

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

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

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

4.2.2.2 Union Types

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

The function prototype for a union has the form:

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

where:

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

4.2.2.3 Null Types

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

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

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

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

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 marshaling 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 **buffer** is a NULL pointer, then no data is marshaled, but the routine will compute and return the size of the memory required to marshal the indicated type. ***size** is not changed.

If **buffer** is not a NULL pointer, data is marshaled, ***buffer** is advanced to point to the first octet of the next location in the output buffer, and the called routine will return the number of octets marshaled into ****buffer**. This occurs even if **size** is a NULL pointer. If **size** is a not NULL pointer ***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. The presumption is also that the **size** is sufficient for the source being marshaled.

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 *source, 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 Part 3 Parsing

The Command / Response tables in Part 3 of this specification are processed by scripts to produce the command-specific data structures used by functions in this TPM 2.0 Part 4. They are:

- **CommandAttributeData.h** -- This file contains the command attributes reported by TPM2_GetCapability.
- **CommandAttributes.h** – This file contains the definition of command attributes that are extracted by the parsing code. The file mainly exists to ensure that the parsing code and the function code are using the same attributes.
- **CommandDispatchData.h** – This file contains the data definitions for the table driven version of the command dispatcher.

Part 3 parsing also produces special function prototype files as described in **Error! Reference source not found.**.

4.4 Function Prototypes

For functions that have entry definitions not defined by Part 3 tables. a script is used to extracts function prototypes from the code. For each .c file that is not in Part 3, a file with the same name is created with a suffix of _fp.h. For example, the function prototypes for Create.c will be placed in a file called Create_fp.h. The _fp.h is added because some files have two types of associated headers: the one containing the function prototypes for the file and another containing definitions that are specific to that file.

In some cases, a function will be replaced by a macro. The macro is defined in the .c file and extracted by the function prototype processor. A special comment tag ("//%") is used to indicate that the line is to be included in the function prototype file. If the "//%" tag occurs at the start of the line, it is deleted. If it occurs later in the line, it is preserved. Removing the "//%/ at the start of the line allows the macro to be placed in the .c file with the tag as a prefix, and then show up in the _fp.h file as the actual macro. This allows the code that includes that function prototype code to use the appropriate macro.

For files that that contain the command actions, a special _fp.h file is created from the tables in Part 3. These files contain:

- the definition of the input and output structure of the function;
- definition of command-specific return code modifiers (parameter identifiers); and
- the function prototype for the command action function.

Create_fp.h (shown below) is prototypical of the command _fp.h files.

```
1   #ifdef TPM_CC_Create // Command must be defined
2   #ifndef _Create_H
3   #define _Create_H
```

Input structure definition

```
4   typedef struct {
5       TPMI_DH_OBJECT            parentHandle;
6       TPM2B_SENSITIVE_CREATE    inSensitive;
7       TPM2B_PUBLIC              inPublic;
8       TPM2B_DATA                outsideInfo;
9       TPML_PCR_SELECTION        creationPCR;
10  } Create_In;
```

Output structure definition

```
11  typedef struct {
12      TPM2B_PRIVATE             outPrivate;
13      TPM2B_PUBLIC              outPublic;
```

```
14      TPM2B_CREATION_DATA       creationData;
15      TPM2B_DIGEST              creationHash;
16      TPMT_TK_CREATION          creationTicket;
17  } Create_Out;
```

Response code modifiers

```
18  #define    RC_Create_parentHandle     (TPM_RC_H + TPM_RC_1)
19  #define    RC_Create_inSensitive      (TPM_RC_P + TPM_RC_1)
20  #define    RC_Create_inPublic         (TPM_RC_P + TPM_RC_2)
21  #define    RC_Create_outsideInfo      (TPM_RC_P + TPM_RC_3)
22  #define    RC_Create_creationPCR      (TPM_RC_P + TPM_RC_4)
```

Function prototype

```
23  TPM_RC
24  TPM2_Create(
25      Create_In *in,
26      Create_Out *out
27  );
28  #endif  // _Create_H
29  #endif  // TPM_CC_Create
```

4.5 Portability

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

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

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

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

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

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

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

5 Header Files

5.1 Introduction

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

5.2 BaseTypes.h

```
1   #ifndef _BASETYPES_H
2   #define _BASETYPES_H
```

NULL definition

```
3   #ifndef         NULL
4   #define         NULL        (0)
5   #endif
6   typedef  uint8_t            UINT8;
7   typedef  uint8_t            BYTE;
8   typedef  int8_t             INT8;
9   typedef  int                BOOL;
10  typedef  uint16_t           UINT16;
11  typedef  int16_t            INT16;
12  typedef  uint32_t           UINT32;
13  typedef  int32_t            INT32;
14  typedef  uint64_t           UINT64;
15  typedef  int64_t            INT64;
16  #endif
```


5.3 Bits_fp.h

```
#ifndef    _BITS_FP_H_
#define    _BITS_FP_H_
```

5.3.1 TestBit()

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

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

5.3.2 SetBit()

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

```
#ifndef INLINE_FUNCTIONS
void
SetBit(
    unsigned int     bitNum,        // IN: number of the bit in 'bArray'
    BYTE            *bArray,        // IN: array containing the bits
    unsigned int     bytesInArray   // IN: size in bytes of 'bArray'
    );
#else
INLINE void
SetBit(
    unsigned int     bitNum,        // IN: number of the bit in 'bArray'
    BYTE            *bArray,        // IN: array containing the bits
    unsigned int     bytesInArray   // IN: size in bytes of 'bArray'
    )
{
    pAssert(bytesInArray > (bitNum >> 3));
    bArray[bitNum >> 3] |= (1 << (bitNum & 7));
}
#endif // INLINE_FUNCTIONS
```


5.3.3 ClearBit()

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

```
41  #ifndef INLINE_FUNCTIONS
42  void
43  ClearBit(
44      unsigned int     bitNum,        // IN: number of the bit in 'bArray'.
45      BYTE            *bArray,        // IN: array containing the bits
46      unsigned int     bytesInArray   // IN: size in bytes of 'bArray'
47      );
48  #else
49  INLINE void
50  ClearBit(
51      unsigned int     bitNum,        // IN: number of the bit in 'bArray'.
52      BYTE            *bArray,        // IN: array containing the bits
53      unsigned int     bytesInArray   // IN: size in bytes of 'bArray'
54      )
55  {
56      pAssert(bytesInArray > (bitNum >> 3));
57      bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
58  }
59  #endif // INLINE_FUNCTIONS
60  #endif  // _BITS_FP_H_
```


5.4 Bool.h

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


5.5 Capabilities.h

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

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

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

```
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  #define    MAX_TAGGED_POLICIES   (MAX_CAP_DATA/sizeof(TPMS_TAGGED_POLICY))
11  #endif
```


5.6 CommandAttributeData.h

This file should only be included by CommandCodeAttibutes.c

```
#ifdef _COMMAND_CODE_ATTRIBUTES_
#include "CommandAttributes.h"
#if defined COMPRESSED_LISTS
#   define      PAD_LIST    0
#else
#   define      PAD_LIST    1
#endif
```

This is the command code attribute array for GetCapability(). Both this array and *s_commandAttributes* provides command code attributes, but tuned for different purpose

```
const TPMA_CC    s_ccAttr [] = {
#if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
        {0x011f, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_UndefineSpaceSpecial
#endif
#if (PAD_LIST || CC_EvictControl)
        {0x0120, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_EvictControl
#endif
#if (PAD_LIST || CC_HierarchyControl)
        {0x0121, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_HierarchyControl
#endif
#if (PAD_LIST || CC_NV_UndefineSpace)
        {0x0122, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_UndefineSpace
#endif
#if (PAD_LIST)
        {0x0123, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
#endif
#if (PAD_LIST || CC_ChangeEPS)
        {0x0124, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_ChangeEPS
#endif
#if (PAD_LIST || CC_ChangePPS)
        {0x0125, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_ChangePPS
#endif
#if (PAD_LIST || CC_Clear)
        {0x0126, 0, 1, 1, 0, 1, 0, 0, 0},     // TPM_CC_Clear
#endif
#if (PAD_LIST || CC_ClearControl)
        {0x0127, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClearControl
#endif
#if (PAD_LIST || CC_ClockSet)
        {0x0128, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClockSet
#endif
#if (PAD_LIST || CC_HierarchyChangeAuth)
        {0x0129, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_HierarchyChangeAuth
#endif
#if (PAD_LIST || CC_NV_DefineSpace)
        {0x012a, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_DefineSpace
#endif
#if (PAD_LIST || CC_PCR_Allocate)
        {0x012b, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Allocate
#endif
#if (PAD_LIST || CC_PCR_SetAuthPolicy)
        {0x012c, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_SetAuthPolicy
#endif
#if (PAD_LIST || CC_PP_Commands)
        {0x012d, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PP_Commands
#endif
#if (PAD_LIST || CC_SetPrimaryPolicy)
        {0x012e, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetPrimaryPolicy
#endif
```

```
#if (PAD_LIST || CC_FieldUpgradeStart)
        {0x012f, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_FieldUpgradeStart
#endif
#if (PAD_LIST || CC_ClockRateAdjust)
        {0x0130, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ClockRateAdjust
#endif
#if (PAD_LIST || CC_CreatePrimary)
        {0x0131, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_CreatePrimary
#endif
#if (PAD_LIST || CC_NV_GlobalWriteLock)
        {0x0132, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_GlobalWriteLock
#endif
#if (PAD_LIST || CC_GetCommandAuditDigest)
        {0x0133, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_GetCommandAuditDigest
#endif
#if (PAD_LIST || CC_NV_Increment)
        {0x0134, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Increment
#endif
#if (PAD_LIST || CC_NV_SetBits)
        {0x0135, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_SetBits
#endif
#if (PAD_LIST || CC_NV_Extend)
        {0x0136, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Extend
#endif
#if (PAD_LIST || CC_NV_Write)
        {0x0137, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Write
#endif
#if (PAD_LIST || CC_NV_WriteLock)
        {0x0138, 0, 1, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_WriteLock
#endif
#if (PAD_LIST || CC_DictionaryAttackLockReset)
        {0x0139, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_DictionaryAttackLockReset
#endif
#if (PAD_LIST || CC_DictionaryAttackParameters)
        {0x013a, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_DictionaryAttackParameters
#endif
#if (PAD_LIST || CC_NV_ChangeAuth)
        {0x013b, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_ChangeAuth
#endif
#if (PAD_LIST || CC_PCR_Event)
        {0x013c, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Event
#endif
#if (PAD_LIST || CC_PCR_Reset)
        {0x013d, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Reset
#endif
#if (PAD_LIST || CC_SequenceComplete)
        {0x013e, 0, 0, 0, 1, 1, 0, 0, 0},     // TPM_CC_SequenceComplete
#endif
#if (PAD_LIST || CC_SetAlgorithmSet)
        {0x013f, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetAlgorithmSet
#endif
#if (PAD_LIST || CC_SetCommandCodeAuditStatus)
        {0x0140, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_SetCommandCodeAuditStatus
#endif
#if (PAD_LIST || CC_FieldUpgradeData)
        {0x0141, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_FieldUpgradeData
#endif
#if (PAD_LIST || CC_IncrementalSelfTest)
        {0x0142, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_IncrementalSelfTest
#endif
#if (PAD_LIST || CC_SelfTest)
        {0x0143, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_SelfTest
#endif
#if (PAD_LIST || CC_Startup)
        {0x0144, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_Startup
#endif
```



```
123   #if (PAD_LIST || CC_Shutdown)
124           {0x0145, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_Shutdown
125   #endif
126   #if (PAD_LIST || CC_StirRandom)
127           {0x0146, 0, 1, 0, 0, 0, 0, 0, 0},     // TPM_CC_StirRandom
128   #endif
129   #if (PAD_LIST || CC_ActivateCredential)
130           {0x0147, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_ActivateCredential
131   #endif
132   #if (PAD_LIST || CC_Certify)
133           {0x0148, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Certify
134   #endif
135   #if (PAD_LIST || CC_PolicyNV)
136           {0x0149, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_PolicyNV
137   #endif
138   #if (PAD_LIST || CC_CertifyCreation)
139           {0x014a, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_CertifyCreation
140   #endif
141   #if (PAD_LIST || CC_Duplicate)
142           {0x014b, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Duplicate
143   #endif
144   #if (PAD_LIST || CC_GetTime)
145           {0x014c, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_GetTime
146   #endif
147   #if (PAD_LIST || CC_GetSessionAuditDigest)
148           {0x014d, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_GetSessionAuditDigest
149   #endif
150   #if (PAD_LIST || CC_NV_Read)
151           {0x014e, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_Read
152   #endif
153   #if (PAD_LIST || CC_NV_ReadLock)
154           {0x014f, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_NV_ReadLock
155   #endif
156   #if (PAD_LIST || CC_ObjectChangeAuth)
157           {0x0150, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_ObjectChangeAuth
158   #endif
159   #if (PAD_LIST || CC_PolicySecret)
160           {0x0151, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_PolicySecret
161   #endif
162   #if (PAD_LIST || CC_Rewrap)
163           {0x0152, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_Rewrap
164   #endif
165   #if (PAD_LIST || CC_Create)
166           {0x0153, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Create
167   #endif
168   #if (PAD_LIST || CC_ECDH_ZGen)
169           {0x0154, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ECDH_ZGen
170   #endif
171   #if (PAD_LIST || CC_HMAC)
172           {0x0155, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_HMAC
173   #endif
174   #if (PAD_LIST || CC_Import)
175           {0x0156, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Import
176   #endif
177   #if (PAD_LIST || CC_Load)
178           {0x0157, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_Load
179   #endif
180   #if (PAD_LIST || CC_Quote)
181           {0x0158, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Quote
182   #endif
183   #if (PAD_LIST || CC_RSA_Decrypt)
184           {0x0159, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_RSA_Decrypt
185   #endif
186   #if (PAD_LIST)
187           {0x015a, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
188   #endif
```

```
189   #if (PAD_LIST || CC_HMAC_Start)
190           {0x015b, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_HMAC_Start
191   #endif
192   #if (PAD_LIST || CC_SequenceUpdate)
193           {0x015c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_SequenceUpdate
194   #endif
195   #if (PAD_LIST || CC_Sign)
196           {0x015d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Sign
197   #endif
198   #if (PAD_LIST || CC_Unseal)
199           {0x015e, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Unseal
200   #endif
201   #if (PAD_LIST)
202           {0x015f, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
203   #endif
204   #if (PAD_LIST || CC_PolicySigned)
205           {0x0160, 0, 0, 0, 0, 2, 0, 0, 0},     // TPM_CC_PolicySigned
206   #endif
207   #if (PAD_LIST || CC_ContextLoad)
208           {0x0161, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_ContextLoad
209   #endif
210   #if (PAD_LIST || CC_ContextSave)
211           {0x0162, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ContextSave
212   #endif
213   #if (PAD_LIST || CC_ECDH_KeyGen)
214           {0x0163, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ECDH_KeyGen
215   #endif
216   #if (PAD_LIST || CC_EncryptDecrypt)
217           {0x0164, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_EncryptDecrypt
218   #endif
219   #if (PAD_LIST || CC_FlushContext)
220           {0x0165, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_FlushContext
221   #endif
222   #if (PAD_LIST)
223           {0x0166, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
224   #endif
225   #if (PAD_LIST || CC_LoadExternal)
226           {0x0167, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_LoadExternal
227   #endif
228   #if (PAD_LIST || CC_MakeCredential)
229           {0x0168, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_MakeCredential
230   #endif
231   #if (PAD_LIST || CC_NV_ReadPublic)
232           {0x0169, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_NV_ReadPublic
233   #endif
234   #if (PAD_LIST || CC_PolicyAuthorize)
235           {0x016a, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyAuthorize
236   #endif
237   #if (PAD_LIST || CC_PolicyAuthValue)
238           {0x016b, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyAuthValue
239   #endif
240   #if (PAD_LIST || CC_PolicyCommandCode)
241           {0x016c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCommandCode
242   #endif
243   #if (PAD_LIST || CC_PolicyCounterTimer)
244           {0x016d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCounterTimer
245   #endif
246   #if (PAD_LIST || CC_PolicyCpHash)
247           {0x016e, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyCpHash
248   #endif
249   #if (PAD_LIST || CC_PolicyLocality)
250           {0x016f, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyLocality
251   #endif
252   #if (PAD_LIST || CC_PolicyNameHash)
253           {0x0170, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyNameHash
254   #endif
```



```
255   #if (PAD_LIST || CC_PolicyOR)
256           {0x0171, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyOR
257   #endif
258   #if (PAD_LIST || CC_PolicyTicket)
259           {0x0172, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyTicket
260   #endif
261   #if (PAD_LIST || CC_ReadPublic)
262           {0x0173, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ReadPublic
263   #endif
264   #if (PAD_LIST || CC_RSA_Encrypt)
265           {0x0174, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_RSA_Encrypt
266   #endif
267   #if (PAD_LIST)
268           {0x0175, 0, 0, 0, 0, 0, 0, 0, 0},     // No command
269   #endif
270   #if (PAD_LIST || CC_StartAuthSession)
271           {0x0176, 0, 0, 0, 0, 2, 1, 0, 0},     // TPM_CC_StartAuthSession
272   #endif
273   #if (PAD_LIST || CC_VerifySignature)
274           {0x0177, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_VerifySignature
275   #endif
276   #if (PAD_LIST || CC_ECC_Parameters)
277           {0x0178, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_ECC_Parameters
278   #endif
279   #if (PAD_LIST || CC_FirmwareRead)
280           {0x0179, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_FirmwareRead
281   #endif
282   #if (PAD_LIST || CC_GetCapability)
283           {0x017a, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetCapability
284   #endif
285   #if (PAD_LIST || CC_GetRandom)
286           {0x017b, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetRandom
287   #endif
288   #if (PAD_LIST || CC_GetTestResult)
289           {0x017c, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_GetTestResult
290   #endif
291   #if (PAD_LIST || CC_Hash)
292           {0x017d, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_Hash
293   #endif
294   #if (PAD_LIST || CC_PCR_Read)
295           {0x017e, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_PCR_Read
296   #endif
297   #if (PAD_LIST || CC_PolicyPCR)
298           {0x017f, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPCR
299   #endif
300   #if (PAD_LIST || CC_PolicyRestart)
301           {0x0180, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyRestart
302   #endif
303   #if (PAD_LIST || CC_ReadClock)
304           {0x0181, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_ReadClock
305   #endif
306   #if (PAD_LIST || CC_PCR_Extend)
307           {0x0182, 0, 1, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_Extend
308   #endif
309   #if (PAD_LIST || CC_PCR_SetAuthValue)
310           {0x0183, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PCR_SetAuthValue
311   #endif
312   #if (PAD_LIST || CC_NV_Certify)
313           {0x0184, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_NV_Certify
314   #endif
315   #if (PAD_LIST || CC_EventSequenceComplete)
316           {0x0185, 0, 1, 0, 1, 2, 0, 0, 0},     // TPM_CC_EventSequenceComplete
317   #endif
318   #if (PAD_LIST || CC_HashSequenceStart)
319           {0x0186, 0, 0, 0, 0, 0, 1, 0, 0},     // TPM_CC_HashSequenceStart
320   #endif
```



```
321   #if (PAD_LIST || CC_PolicyPhysicalPresence)
322           {0x0187, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPhysicalPresence
323   #endif
324   #if (PAD_LIST || CC_PolicyDuplicationSelect)
325           {0x0188, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyDuplicationSelect
326   #endif
327   #if (PAD_LIST || CC_PolicyGetDigest)
328           {0x0189, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyGetDigest
329   #endif
330   #if (PAD_LIST || CC_TestParms)
331           {0x018a, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_TestParms
332   #endif
333   #if (PAD_LIST || CC_Commit)
334           {0x018b, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_Commit
335   #endif
336   #if (PAD_LIST || CC_PolicyPassword)
337           {0x018c, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyPassword
338   #endif
339   #if (PAD_LIST || CC_ZGen_2Phase)
340           {0x018d, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_ZGen_2Phase
341   #endif
342   #if (PAD_LIST || CC_EC_Ephemeral)
343           {0x018e, 0, 0, 0, 0, 0, 0, 0, 0},     // TPM_CC_EC_Ephemeral
344   #endif
345   #if (PAD_LIST || CC_PolicyNvWritten)
346           {0x018f, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyNvWritten
347   #endif
348   #if (PAD_LIST || CC_PolicyTemplate)
349           {0x0190, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_PolicyTemplate
350   #endif
351   #if (PAD_LIST || CC_CreateLoaded)
352           {0x0191, 0, 0, 0, 0, 1, 1, 0, 0},     // TPM_CC_CreateLoaded
353   #endif
354   #if (PAD_LIST || CC_PolicyAuthorizeNV)
355           {0x0192, 0, 0, 0, 0, 3, 0, 0, 0},     // TPM_CC_PolicyAuthorizeNV
356   #endif
357   #if (PAD_LIST || CC_EncryptDecrypt2)
358           {0x0193, 0, 0, 0, 0, 1, 0, 0, 0},     // TPM_CC_EncryptDecrypt2
359   #endif
360   #if (PAD_LIST || CC_Vendor_TCG_Test)
361           {0x0000, 0, 0, 0, 0, 0, 0, 1, 0},     // TPM_CC_Vendor_TCG_Test
362   #endif
363           {0}
364   };
```

This is the command code attribute structure.

```
365   const COMMAND_ATTRIBUTES    s_commandAttributes [] = {
366   #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
367           (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpaceSpecial    *  // 0x011f
368               (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),
369   #endif
370   #if (PAD_LIST || CC_EvictControl)
371           (COMMAND_ATTRIBUTES)(CC_EvictControl               *  // 0x0120
372               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
373   #endif
374   #if (PAD_LIST || CC_HierarchyControl)
375           (COMMAND_ATTRIBUTES)(CC_HierarchyControl           *  // 0x0121
376               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
377   #endif
378   #if (PAD_LIST || CC_NV_UndefineSpace)
379           (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpace           *  // 0x0122
380               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
381   #endif
382   #if (PAD_LIST)
```

```
383             (COMMAND_ATTRIBUTES)(0),                                // 0x0123
384     #endif
385     #if (PAD_LIST || CC_ChangeEPS)
386             (COMMAND_ATTRIBUTES)(CC_ChangeEPS                       *  // 0x0124
387                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
388     #endif
389     #if (PAD_LIST || CC_ChangePPS)
390             (COMMAND_ATTRIBUTES)(CC_ChangePPS                       *  // 0x0125
391                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
392     #endif
393     #if (PAD_LIST || CC_Clear)
394             (COMMAND_ATTRIBUTES)(CC_Clear                           *  // 0x0126
395                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
396     #endif
397     #if (PAD_LIST || CC_ClearControl)
398             (COMMAND_ATTRIBUTES)(CC_ClearControl                    *  // 0x0127
399                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
400     #endif
401     #if (PAD_LIST || CC_ClockSet)
402             (COMMAND_ATTRIBUTES)(CC_ClockSet                        *  // 0x0128
403                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
404     #endif
405     #if (PAD_LIST || CC_HierarchyChangeAuth)
406             (COMMAND_ATTRIBUTES)(CC_HierarchyChangeAuth             *  // 0x0129
407                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
408     #endif
409     #if (PAD_LIST || CC_NV_DefineSpace)
410             (COMMAND_ATTRIBUTES)(CC_NV_DefineSpace                  *  // 0x012a
411                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
412     #endif
413     #if (PAD_LIST || CC_PCR_Allocate)
414             (COMMAND_ATTRIBUTES)(CC_PCR_Allocate                    *  // 0x012b
415                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
416     #endif
417     #if (PAD_LIST || CC_PCR_SetAuthPolicy)
418             (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthPolicy               *  // 0x012c
419                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
420     #endif
421     #if (PAD_LIST || CC_PP_Commands)
422             (COMMAND_ATTRIBUTES)(CC_PP_Commands                     *  // 0x012d
423                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),
424     #endif
425     #if (PAD_LIST || CC_SetPrimaryPolicy)
426             (COMMAND_ATTRIBUTES)(CC_SetPrimaryPolicy                *  // 0x012e
427                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
428     #endif
429     #if (PAD_LIST || CC_FieldUpgradeStart)
430             (COMMAND_ATTRIBUTES)(CC_FieldUpgradeStart               *  // 0x012f
431                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),
432     #endif
433     #if (PAD_LIST || CC_ClockRateAdjust)
434             (COMMAND_ATTRIBUTES)(CC_ClockRateAdjust                 *  // 0x0130
435                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
436     #endif
437     #if (PAD_LIST || CC_CreatePrimary)
438             (COMMAND_ATTRIBUTES)(CC_CreatePrimary                   *  // 0x0131
439                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
440     #endif
441     #if (PAD_LIST || CC_NV_GlobalWriteLock)
442             (COMMAND_ATTRIBUTES)(CC_NV_GlobalWriteLock              *  // 0x0132
443                 (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
444     #endif
445     #if (PAD_LIST || CC_GetCommandAuditDigest)
446             (COMMAND_ATTRIBUTES)(CC_GetCommandAuditDigest           *  // 0x0133
447                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
448     #endif
```

```
449   #if (PAD_LIST || CC_NV_Increment)
450           (COMMAND_ATTRIBUTES)(CC_NV_Increment               *  // 0x0134
451               (IS_IMPLEMENTED+HANDLE_1_USER)),
452   #endif
453   #if (PAD_LIST || CC_NV_SetBits)
454           (COMMAND_ATTRIBUTES)(CC_NV_SetBits                 *  // 0x0135
455               (IS_IMPLEMENTED+HANDLE_1_USER)),
456   #endif
457   #if (PAD_LIST || CC_NV_Extend)
458           (COMMAND_ATTRIBUTES)(CC_NV_Extend                  *  // 0x0136
459               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
460   #endif
461   #if (PAD_LIST || CC_NV_Write)
462           (COMMAND_ATTRIBUTES)(CC_NV_Write                   *  // 0x0137
463               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
464   #endif
465   #if (PAD_LIST || CC_NV_WriteLock)
466           (COMMAND_ATTRIBUTES)(CC_NV_WriteLock               *  // 0x0138
467               (IS_IMPLEMENTED+HANDLE_1_USER)),
468   #endif
469   #if (PAD_LIST || CC_DictionaryAttackLockReset)
470           (COMMAND_ATTRIBUTES)(CC_DictionaryAttackLockReset  *  // 0x0139
471               (IS_IMPLEMENTED+HANDLE_1_USER)),
472   #endif
473   #if (PAD_LIST || CC_DictionaryAttackParameters)
474           (COMMAND_ATTRIBUTES)(CC_DictionaryAttackParameters *  // 0x013a
475               (IS_IMPLEMENTED+HANDLE_1_USER)),
476   #endif
477   #if (PAD_LIST || CC_NV_ChangeAuth)
478           (COMMAND_ATTRIBUTES)(CC_NV_ChangeAuth              *  // 0x013b
479               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),
480   #endif
481   #if (PAD_LIST || CC_PCR_Event)
482           (COMMAND_ATTRIBUTES)(CC_PCR_Event                  *  // 0x013c
483               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
484   #endif
485   #if (PAD_LIST || CC_PCR_Reset)
486           (COMMAND_ATTRIBUTES)(CC_PCR_Reset                  *  // 0x013d
487               (IS_IMPLEMENTED+HANDLE_1_USER)),
488   #endif
489   #if (PAD_LIST || CC_SequenceComplete)
490           (COMMAND_ATTRIBUTES)(CC_SequenceComplete           *  // 0x013e
491               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
492   #endif
493   #if (PAD_LIST || CC_SetAlgorithmSet)
494           (COMMAND_ATTRIBUTES)(CC_SetAlgorithmSet            *  // 0x013f
495               (IS_IMPLEMENTED+HANDLE_1_USER)),
496   #endif
497   #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
498           (COMMAND_ATTRIBUTES)(CC_SetCommandCodeAuditStatus  *  // 0x0140
499               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
500   #endif
501   #if (PAD_LIST || CC_FieldUpgradeData)
502           (COMMAND_ATTRIBUTES)(CC_FieldUpgradeData           *  // 0x0141
503               (IS_IMPLEMENTED+DECRYPT_2)),
504   #endif
505   #if (PAD_LIST || CC_IncrementalSelfTest)
506           (COMMAND_ATTRIBUTES)(CC_IncrementalSelfTest        *  // 0x0142
507               (IS_IMPLEMENTED)),
508   #endif
509   #if (PAD_LIST || CC_SelfTest)
510           (COMMAND_ATTRIBUTES)(CC_SelfTest                   *  // 0x0143
511               (IS_IMPLEMENTED)),
512   #endif
513   #if (PAD_LIST || CC_Startup)
514           (COMMAND_ATTRIBUTES)(CC_Startup                    *  // 0x0144
```

```
515             (IS_IMPLEMENTED+NO_SESSIONS)),
516     #endif
517     #if (PAD_LIST || CC_Shutdown)
518             (COMMAND_ATTRIBUTES)(CC_Shutdown                    *  // 0x0145
519                 (IS_IMPLEMENTED)),
520     #endif
521     #if (PAD_LIST || CC_StirRandom)
522             (COMMAND_ATTRIBUTES)(CC_StirRandom                  *  // 0x0146
523                 (IS_IMPLEMENTED+DECRYPT_2)),
524     #endif
525     #if (PAD_LIST || CC_ActivateCredential)
526             (COMMAND_ATTRIBUTES)(CC_ActivateCredential          *  // 0x0147
527                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
528     #endif
529     #if (PAD_LIST || CC_Certify)
530             (COMMAND_ATTRIBUTES)(CC_Certify                     *  // 0x0148
531                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
532     #endif
533     #if (PAD_LIST || CC_PolicyNV)
534             (COMMAND_ATTRIBUTES)(CC_PolicyNV                    *  // 0x0149
535                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL)),
536     #endif
537     #if (PAD_LIST || CC_CertifyCreation)
538             (COMMAND_ATTRIBUTES)(CC_CertifyCreation             *  // 0x014a
539                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
540     #endif
541     #if (PAD_LIST || CC_Duplicate)
542             (COMMAND_ATTRIBUTES)(CC_Duplicate                   *  // 0x014b
543                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),
544     #endif
545     #if (PAD_LIST || CC_GetTime)
546             (COMMAND_ATTRIBUTES)(CC_GetTime                     *  // 0x014c
547                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
548     #endif
549     #if (PAD_LIST || CC_GetSessionAuditDigest)
550             (COMMAND_ATTRIBUTES)(CC_GetSessionAuditDigest       *  // 0x014d
551                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
552     #endif
553     #if (PAD_LIST || CC_NV_Read)
554             (COMMAND_ATTRIBUTES)(CC_NV_Read                     *  // 0x014e
555                 (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
556     #endif
557     #if (PAD_LIST || CC_NV_ReadLock)
558             (COMMAND_ATTRIBUTES)(CC_NV_ReadLock                 *  // 0x014f
559                 (IS_IMPLEMENTED+HANDLE_1_USER)),
560     #endif
561     #if (PAD_LIST || CC_ObjectChangeAuth)
562             (COMMAND_ATTRIBUTES)(CC_ObjectChangeAuth            *  // 0x0150
563                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),
564     #endif
565     #if (PAD_LIST || CC_PolicySecret)
566             (COMMAND_ATTRIBUTES)(CC_PolicySecret                *  // 0x0151
567                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL+ENCRYPT_2)),
568     #endif
569     #if (PAD_LIST || CC_Rewrap)
570             (COMMAND_ATTRIBUTES)(CC_Rewrap                      *  // 0x0152
571                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
572     #endif
573     #if (PAD_LIST || CC_Create)
574             (COMMAND_ATTRIBUTES)(CC_Create                      *  // 0x0153
575                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
576     #endif
577     #if (PAD_LIST || CC_ECDH_ZGen)
578             (COMMAND_ATTRIBUTES)(CC_ECDH_ZGen                   *  // 0x0154
579                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
580     #endif
```

```
581   #if (PAD_LIST || CC_HMAC)
582           (COMMAND_ATTRIBUTES)(CC_HMAC                        *  // 0x0155
583               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
584   #endif
585   #if (PAD_LIST || CC_Import)
586           (COMMAND_ATTRIBUTES)(CC_Import                      *  // 0x0156
587               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
588   #endif
589   #if (PAD_LIST || CC_Load)
590           (COMMAND_ATTRIBUTES)(CC_Load                        *  // 0x0157
591               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),
592   #endif
593   #if (PAD_LIST || CC_Quote)
594           (COMMAND_ATTRIBUTES)(CC_Quote                       *  // 0x0158
595               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
596   #endif
597   #if (PAD_LIST || CC_RSA_Decrypt)
598           (COMMAND_ATTRIBUTES)(CC_RSA_Decrypt                 *  // 0x0159
599               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
600   #endif
601   #if (PAD_LIST)
602           (COMMAND_ATTRIBUTES)(0),                               // 0x015a
603   #endif
604   #if (PAD_LIST || CC_HMAC_Start)
605           (COMMAND_ATTRIBUTES)(CC_HMAC_Start                  *  // 0x015b
606               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),
607   #endif
608   #if (PAD_LIST || CC_SequenceUpdate)
609           (COMMAND_ATTRIBUTES)(CC_SequenceUpdate              *  // 0x015c
610               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
611   #endif
612   #if (PAD_LIST || CC_Sign)
613           (COMMAND_ATTRIBUTES)(CC_Sign                        *  // 0x015d
614               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
615   #endif
616   #if (PAD_LIST || CC_Unseal)
617           (COMMAND_ATTRIBUTES)(CC_Unseal                      *  // 0x015e
618               (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
619   #endif
620   #if (PAD_LIST)
621           (COMMAND_ATTRIBUTES)(0),                               // 0x015f
622   #endif
623   #if (PAD_LIST || CC_PolicySigned)
624           (COMMAND_ATTRIBUTES)(CC_PolicySigned                *  // 0x0160
625               (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL+ENCRYPT_2)),
626   #endif
627   #if (PAD_LIST || CC_ContextLoad)
628           (COMMAND_ATTRIBUTES)(CC_ContextLoad                 *  // 0x0161
629               (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
630   #endif
631   #if (PAD_LIST || CC_ContextSave)
632           (COMMAND_ATTRIBUTES)(CC_ContextSave                 *  // 0x0162
633               (IS_IMPLEMENTED+NO_SESSIONS)),
634   #endif
635   #if (PAD_LIST || CC_ECDH_KeyGen)
636           (COMMAND_ATTRIBUTES)(CC_ECDH_KeyGen                 *  // 0x0163
637               (IS_IMPLEMENTED+ENCRYPT_2)),
638   #endif
639   #if (PAD_LIST || CC_EncryptDecrypt)
640           (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt              *  // 0x0164
641               (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
642   #endif
643   #if (PAD_LIST || CC_FlushContext)
644           (COMMAND_ATTRIBUTES)(CC_FlushContext                *  // 0x0165
645               (IS_IMPLEMENTED+NO_SESSIONS)),
646   #endif
```

```
647   #if (PAD_LIST)
648           (COMMAND_ATTRIBUTES)(0),                                  // 0x0166
649   #endif
650   #if (PAD_LIST || CC_LoadExternal)
651           (COMMAND_ATTRIBUTES)(CC_LoadExternal                *  // 0x0167
652               (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
653   #endif
654   #if (PAD_LIST || CC_MakeCredential)
655           (COMMAND_ATTRIBUTES)(CC_MakeCredential              *  // 0x0168
656               (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
657   #endif
658   #if (PAD_LIST || CC_NV_ReadPublic)
659           (COMMAND_ATTRIBUTES)(CC_NV_ReadPublic               *  // 0x0169
660               (IS_IMPLEMENTED+ENCRYPT_2)),
661   #endif
662   #if (PAD_LIST || CC_PolicyAuthorize)
663           (COMMAND_ATTRIBUTES)(CC_PolicyAuthorize             *  // 0x016a
664               (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
665   #endif
666   #if (PAD_LIST || CC_PolicyAuthValue)
667           (COMMAND_ATTRIBUTES)(CC_PolicyAuthValue             *  // 0x016b
668               (IS_IMPLEMENTED+ALLOW_TRIAL)),
669   #endif
670   #if (PAD_LIST || CC_PolicyCommandCode)
671           (COMMAND_ATTRIBUTES)(CC_PolicyCommandCode           *  // 0x016c
672               (IS_IMPLEMENTED+ALLOW_TRIAL)),
673   #endif
674   #if (PAD_LIST || CC_PolicyCounterTimer)
675           (COMMAND_ATTRIBUTES)(CC_PolicyCounterTimer          *  // 0x016d
676               (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
677   #endif
678   #if (PAD_LIST || CC_PolicyCpHash)
679           (COMMAND_ATTRIBUTES)(CC_PolicyCpHash                *  // 0x016e
680               (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
681   #endif
682   #if (PAD_LIST || CC_PolicyLocality)
683           (COMMAND_ATTRIBUTES)(CC_PolicyLocality              *  // 0x016f
684               (IS_IMPLEMENTED+ALLOW_TRIAL)),
685   #endif
686   #if (PAD_LIST || CC_PolicyNameHash)
687           (COMMAND_ATTRIBUTES)(CC_PolicyNameHash              *  // 0x0170
688               (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
689   #endif
690   #if (PAD_LIST || CC_PolicyOR)
691           (COMMAND_ATTRIBUTES)(CC_PolicyOR                    *  // 0x0171
692               (IS_IMPLEMENTED+ALLOW_TRIAL)),
693   #endif
694   #if (PAD_LIST || CC_PolicyTicket)
695           (COMMAND_ATTRIBUTES)(CC_PolicyTicket                *  // 0x0172
696               (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
697   #endif
698   #if (PAD_LIST || CC_ReadPublic)
699           (COMMAND_ATTRIBUTES)(CC_ReadPublic                  *  // 0x0173
700               (IS_IMPLEMENTED+ENCRYPT_2)),
701   #endif
702   #if (PAD_LIST || CC_RSA_Encrypt)
703           (COMMAND_ATTRIBUTES)(CC_RSA_Encrypt                 *  // 0x0174
704               (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
705   #endif
706   #if (PAD_LIST)
707           (COMMAND_ATTRIBUTES)(0),                                  // 0x0175
708   #endif
709   #if (PAD_LIST || CC_StartAuthSession)
710           (COMMAND_ATTRIBUTES)(CC_StartAuthSession            *  // 0x0176
711               (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
712   #endif
```

```
713   #if (PAD_LIST || CC_VerifySignature)
714           (COMMAND_ATTRIBUTES)(CC_VerifySignature           *  // 0x0177
715               (IS_IMPLEMENTED+DECRYPT_2)),
716   #endif
717   #if (PAD_LIST || CC_ECC_Parameters)
718           (COMMAND_ATTRIBUTES)(CC_ECC_Parameters            *  // 0x0178
719               (IS_IMPLEMENTED)),
720   #endif
721   #if (PAD_LIST || CC_FirmwareRead)
722           (COMMAND_ATTRIBUTES)(CC_FirmwareRead              *  // 0x0179
723               (IS_IMPLEMENTED+ENCRYPT_2)),
724   #endif
725   #if (PAD_LIST || CC_GetCapability)
726           (COMMAND_ATTRIBUTES)(CC_GetCapability             *  // 0x017a
727               (IS_IMPLEMENTED)),
728   #endif
729   #if (PAD_LIST || CC_GetRandom)
730           (COMMAND_ATTRIBUTES)(CC_GetRandom                 *  // 0x017b
731               (IS_IMPLEMENTED+ENCRYPT_2)),
732   #endif
733   #if (PAD_LIST || CC_GetTestResult)
734           (COMMAND_ATTRIBUTES)(CC_GetTestResult             *  // 0x017c
735               (IS_IMPLEMENTED+ENCRYPT_2)),
736   #endif
737   #if (PAD_LIST || CC_Hash)
738           (COMMAND_ATTRIBUTES)(CC_Hash                      *  // 0x017d
739               (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
740   #endif
741   #if (PAD_LIST || CC_PCR_Read)
742           (COMMAND_ATTRIBUTES)(CC_PCR_Read                  *  // 0x017e
743               (IS_IMPLEMENTED)),
744   #endif
745   #if (PAD_LIST || CC_PolicyPCR)
746           (COMMAND_ATTRIBUTES)(CC_PolicyPCR                 *  // 0x017f
747               (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
748   #endif
749   #if (PAD_LIST || CC_PolicyRestart)
750           (COMMAND_ATTRIBUTES)(CC_PolicyRestart             *  // 0x0180
751               (IS_IMPLEMENTED+ALLOW_TRIAL)),
752   #endif
753   #if (PAD_LIST || CC_ReadClock)
754           (COMMAND_ATTRIBUTES)(CC_ReadClock                 *  // 0x0181
755               (IS_IMPLEMENTED)),
756   #endif
757   #if (PAD_LIST || CC_PCR_Extend)
758           (COMMAND_ATTRIBUTES)(CC_PCR_Extend                *  // 0x0182
759               (IS_IMPLEMENTED+HANDLE_1_USER)),
760   #endif
761   #if (PAD_LIST || CC_PCR_SetAuthValue)
762           (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthValue          *  // 0x0183
763               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
764   #endif
765   #if (PAD_LIST || CC_NV_Certify)
766           (COMMAND_ATTRIBUTES)(CC_NV_Certify                *  // 0x0184
767               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
768   #endif
769   #if (PAD_LIST || CC_EventSequenceComplete)
770           (COMMAND_ATTRIBUTES)(CC_EventSequenceComplete     *  // 0x0185
771               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),
772   #endif
773   #if (PAD_LIST || CC_HashSequenceStart)
774           (COMMAND_ATTRIBUTES)(CC_HashSequenceStart         *  // 0x0186
775               (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
776   #endif
777   #if (PAD_LIST || CC_PolicyPhysicalPresence)
778           (COMMAND_ATTRIBUTES)(CC_PolicyPhysicalPresence    *  // 0x0187
```

```
779             (IS_IMPLEMENTED+ALLOW_TRIAL)),
780     #endif
781     #if (PAD_LIST || CC_PolicyDuplicationSelect)
782             (COMMAND_ATTRIBUTES)(CC_PolicyDuplicationSelect    *  // 0x0188
783                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
784     #endif
785     #if (PAD_LIST || CC_PolicyGetDigest)
786             (COMMAND_ATTRIBUTES)(CC_PolicyGetDigest            *  // 0x0189
787                 (IS_IMPLEMENTED+ALLOW_TRIAL+ENCRYPT_2)),
788     #endif
789     #if (PAD_LIST || CC_TestParms)
790             (COMMAND_ATTRIBUTES)(CC_TestParms                  *  // 0x018a
791                 (IS_IMPLEMENTED)),
792     #endif
793     #if (PAD_LIST || CC_Commit)
794             (COMMAND_ATTRIBUTES)(CC_Commit                     *  // 0x018b
795                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
796     #endif
797     #if (PAD_LIST || CC_PolicyPassword)
798             (COMMAND_ATTRIBUTES)(CC_PolicyPassword             *  // 0x018c
799                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
800     #endif
801     #if (PAD_LIST || CC_ZGen_2Phase)
802             (COMMAND_ATTRIBUTES)(CC_ZGen_2Phase                *  // 0x018d
803                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
804     #endif
805     #if (PAD_LIST || CC_EC_Ephemeral)
806             (COMMAND_ATTRIBUTES)(CC_EC_Ephemeral               *  // 0x018e
807                 (IS_IMPLEMENTED+ENCRYPT_2)),
808     #endif
809     #if (PAD_LIST || CC_PolicyNvWritten)
810             (COMMAND_ATTRIBUTES)(CC_PolicyNvWritten            *  // 0x018f
811                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
812     #endif
813     #if (PAD_LIST || CC_PolicyTemplate)
814             (COMMAND_ATTRIBUTES)(CC_PolicyTemplate             *  // 0x0190
815                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
816     #endif
817     #if (PAD_LIST || CC_CreateLoaded)
818             (COMMAND_ATTRIBUTES)(CC_CreateLoaded               *  // 0x0191
819                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
820     #endif
821     #if (PAD_LIST || CC_PolicyAuthorizeNV)
822             (COMMAND_ATTRIBUTES)(CC_PolicyAuthorizeNV          *  // 0x0192
823                 (IS_IMPLEMENTED+HANDLE_1_USER+ALLOW_TRIAL)),
824     #endif
825     #if (PAD_LIST || CC_EncryptDecrypt2)
826             (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt2            *  // 0x0193
827                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
828     #endif
829     #if (PAD_LIST || CC_Vendor_TCG_Test)
830             (COMMAND_ATTRIBUTES)(CC_Vendor_TCG_Test            *  // 0x0000
831                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
832     #endif
833             0
834     };
835     #endif  // _COMMAND_CODE_ATTRIBUTES_
```

5.7 CommandAttributes.h

The attributes defined in this file are produced by the parser that creates the structure definitions from Part 3. The attributes are defined in that parser and should track the attributes being tested in CommandCodeAttributes.c. Generally, when an attribute is added to this list, new code will be needed in CommandCodeAttributes.c to test it.

```
1   #ifndef        COMMAND_ATTRIBUTES_H
2   #define         COMMAND_ATTRIBUTES_H
3   typedef  UINT16             COMMAND_ATTRIBUTES;
4   #define  NOT_IMPLEMENTED   (COMMAND_ATTRIBUTES)(0)
5   #define  ENCRYPT_2         (COMMAND_ATTRIBUTES)(1 <<  0)
6   #define  ENCRYPT_4         (COMMAND_ATTRIBUTES)(1 <<  1)
7   #define  DECRYPT_2         (COMMAND_ATTRIBUTES)(1 <<  2)
8   #define  DECRYPT_4         (COMMAND_ATTRIBUTES)(1 <<  3)
9   #define  HANDLE_1_USER     (COMMAND_ATTRIBUTES)(1 <<  4)
10  #define  HANDLE_1_ADMIN    (COMMAND_ATTRIBUTES)(1 <<  5)
11  #define  HANDLE_1_DUP      (COMMAND_ATTRIBUTES)(1 <<  6)
12  #define  HANDLE_2_USER     (COMMAND_ATTRIBUTES)(1 <<  7)
13  #define  PP_COMMAND        (COMMAND_ATTRIBUTES)(1 <<  8)
14  #define  IS_IMPLEMENTED    (COMMAND_ATTRIBUTES)(1 <<  9)
15  #define  NO_SESSIONS       (COMMAND_ATTRIBUTES)(1 << 10)
16  #define  NV_COMMAND        (COMMAND_ATTRIBUTES)(1 << 11)
17  #define  PP_REQUIRED       (COMMAND_ATTRIBUTES)(1 << 12)
18  #define  R_HANDLE          (COMMAND_ATTRIBUTES)(1 << 13)
19  #define  ALLOW_TRIAL       (COMMAND_ATTRIBUTES)(1 << 14)
20  #endif // COMMAND_ATTRIBUTES_H
```


5.8 CommandDispatchData.h

This file should only be included by CommandCodeAttibutes.c

```
#ifdef _COMMAND_TABLE_DISPATCH_
```

Define the stop value

```
#define END_OF_LIST     0xff
#define ADD_FLAG        0x80
```

The UnmarshalArray() contains the dispatch functions for the unmarshaling code. The defines in this array are used to make it easier to cross reference the unmarshaling values in the types array of each command

```
const UNMARSHAL_t UnmarshalArray[] = {
#define  TPMI_DH_CONTEXT_H_UNMARSHAL        0
             (UNMARSHAL_t)TPMI_DH_CONTEXT_Unmarshal,
#define TPMI_RH_CLEAR_H_UNMARSHAL       (TPMI_DH_CONTEXT_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_CLEAR_Unmarshal,
#define TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL  (TPMI_RH_CLEAR_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_HIERARCHY_AUTH_Unmarshal,
#define TPMI_RH_LOCKOUT_H_UNMARSHAL     (TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_LOCKOUT_Unmarshal,
#define TPMI_RH_NV_AUTH_H_UNMARSHAL     (TPMI_RH_LOCKOUT_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_NV_AUTH_Unmarshal,
#define TPMI_RH_NV_INDEX_H_UNMARSHAL    (TPMI_RH_NV_AUTH_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_NV_INDEX_Unmarshal,
#define TPMI_RH_PLATFORM_H_UNMARSHAL    (TPMI_RH_NV_INDEX_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_PLATFORM_Unmarshal,
#define TPMI_RH_PROVISION_H_UNMARSHAL   (TPMI_RH_PLATFORM_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_PROVISION_Unmarshal,
#define TPMI_SH_HMAC_H_UNMARSHAL        (TPMI_RH_PROVISION_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_SH_HMAC_Unmarshal,
#define TPMI_SH_POLICY_H_UNMARSHAL      (TPMI_SH_HMAC_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_SH_POLICY_Unmarshal,
// HANDLE_FIRST_FLAG_TYPE is the first handle that needs a flag when called.
#define HANDLE_FIRST_FLAG_TYPE          (TPMI_SH_POLICY_H_UNMARSHAL + 1)
#define TPMI_DH_ENTITY_H_UNMARSHAL      (TPMI_SH_POLICY_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_DH_ENTITY_Unmarshal,
#define TPMI_DH_OBJECT_H_UNMARSHAL      (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_DH_OBJECT_Unmarshal,
#define TPMI_DH_PARENT_H_UNMARSHAL      (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_DH_PARENT_Unmarshal,
#define TPMI_DH_PCR_H_UNMARSHAL         (TPMI_DH_PARENT_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_DH_PCR_Unmarshal,
#define TPMI_RH_ENDORSEMENT_H_UNMARSHAL  (TPMI_DH_PCR_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_ENDORSEMENT_Unmarshal,
#define TPMI_RH_HIERARCHY_H_UNMARSHAL   (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
             (UNMARSHAL_t)TPMI_RH_HIERARCHY_Unmarshal,
// PARAMETER_FIRST_TYPE marks the end of the handle list.
#define PARAMETER_FIRST_TYPE            (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
#define UINT32_P_UNMARSHAL              (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
             (UNMARSHAL_t)UINT32_Unmarshal,
#define TPM2B_DIGEST_P_UNMARSHAL        (UINT32_P_UNMARSHAL + 1)
             (UNMARSHAL_t)TPM2B_DIGEST_Unmarshal,
#define TPM2B_DATA_P_UNMARSHAL          (TPM2B_DIGEST_P_UNMARSHAL + 1)
             (UNMARSHAL_t)TPM2B_DATA_Unmarshal,
#define TPM2B_ECC_PARAMETER_P_UNMARSHAL  (TPM2B_DATA_P_UNMARSHAL + 1)
             (UNMARSHAL_t)TPM2B_ECC_PARAMETER_Unmarshal,
#define TPM2B_ECC_POINT_P_UNMARSHAL     (TPM2B_ECC_PARAMETER_P_UNMARSHAL + 1)
             (UNMARSHAL_t)TPM2B_ECC_POINT_Unmarshal,
#define TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL  (TPM2B_ECC_POINT_P_UNMARSHAL + 1)
```



```
52              (UNMARSHAL_t)TPM2B_ENCRYPTED_SECRET_Unmarshal,
53      #define TPM2B_EVENT_P_UNMARSHAL         (TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL + 1)
54              (UNMARSHAL_t)TPM2B_EVENT_Unmarshal,
55      #define TPM2B_ID_OBJECT_P_UNMARSHAL     (TPM2B_EVENT_P_UNMARSHAL + 1)
56              (UNMARSHAL_t)TPM2B_ID_OBJECT_Unmarshal,
57      #define TPM2B_IV_P_UNMARSHAL            (TPM2B_ID_OBJECT_P_UNMARSHAL + 1)
58              (UNMARSHAL_t)TPM2B_IV_Unmarshal,
59      #define TPM2B_MAX_BUFFER_P_UNMARSHAL    (TPM2B_IV_P_UNMARSHAL + 1)
60              (UNMARSHAL_t)TPM2B_MAX_BUFFER_Unmarshal,
61      #define TPM2B_MAX_NV_BUFFER_P_UNMARSHAL  (TPM2B_MAX_BUFFER_P_UNMARSHAL + 1)
62              (UNMARSHAL_t)TPM2B_MAX_NV_BUFFER_Unmarshal,
63      #define TPM2B_NAME_P_UNMARSHAL          (TPM2B_MAX_NV_BUFFER_P_UNMARSHAL + 1)
64              (UNMARSHAL_t)TPM2B_NAME_Unmarshal,
65      #define TPM2B_NV_PUBLIC_P_UNMARSHAL     (TPM2B_NAME_P_UNMARSHAL + 1)
66              (UNMARSHAL_t)TPM2B_NV_PUBLIC_Unmarshal,
67      #define TPM2B_PRIVATE_P_UNMARSHAL       (TPM2B_NV_PUBLIC_P_UNMARSHAL + 1)
68              (UNMARSHAL_t)TPM2B_PRIVATE_Unmarshal,
69      #define TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL  (TPM2B_PRIVATE_P_UNMARSHAL + 1)
70              (UNMARSHAL_t)TPM2B_PUBLIC_KEY_RSA_Unmarshal,
71      #define TPM2B_SENSITIVE_P_UNMARSHAL     (TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL + 1)
72              (UNMARSHAL_t)TPM2B_SENSITIVE_Unmarshal,
73      #define TPM2B_SENSITIVE_CREATE_P_UNMARSHAL  (TPM2B_SENSITIVE_P_UNMARSHAL + 1)
74              (UNMARSHAL_t)TPM2B_SENSITIVE_CREATE_Unmarshal,
75      #define TPM2B_SENSITIVE_DATA_P_UNMARSHAL  (TPM2B_SENSITIVE_CREATE_P_UNMARSHAL + 1)
76              (UNMARSHAL_t)TPM2B_SENSITIVE_DATA_Unmarshal,
77      #define TPM2B_TEMPLATE_P_UNMARSHAL      (TPM2B_SENSITIVE_DATA_P_UNMARSHAL + 1)
78              (UNMARSHAL_t)TPM2B_TEMPLATE_Unmarshal,
79      #define UINT8_P_UNMARSHAL               (TPM2B_TEMPLATE_P_UNMARSHAL + 1)
80              (UNMARSHAL_t)UINT8_Unmarshal,
81      #define TPMI_DH_CONTEXT_P_UNMARSHAL     (UINT8_P_UNMARSHAL + 1)
82              (UNMARSHAL_t)TPMI_DH_CONTEXT_Unmarshal,
83      #define TPMI_DH_PERSISTENT_P_UNMARSHAL  (TPMI_DH_CONTEXT_P_UNMARSHAL + 1)
84              (UNMARSHAL_t)TPMI_DH_PERSISTENT_Unmarshal,
85      #define TPMI_ECC_CURVE_P_UNMARSHAL      (TPMI_DH_PERSISTENT_P_UNMARSHAL + 1)
86              (UNMARSHAL_t)TPMI_ECC_CURVE_Unmarshal,
87      #define TPMI_YES_NO_P_UNMARSHAL         (TPMI_ECC_CURVE_P_UNMARSHAL + 1)
88              (UNMARSHAL_t)TPMI_YES_NO_Unmarshal,
89      #define TPML_ALG_P_UNMARSHAL            (TPMI_YES_NO_P_UNMARSHAL + 1)
90              (UNMARSHAL_t)TPML_ALG_Unmarshal,
91      #define TPML_CC_P_UNMARSHAL             (TPML_ALG_P_UNMARSHAL + 1)
92              (UNMARSHAL_t)TPML_CC_Unmarshal,
93      #define TPML_DIGEST_P_UNMARSHAL         (TPML_CC_P_UNMARSHAL + 1)
94              (UNMARSHAL_t)TPML_DIGEST_Unmarshal,
95      #define TPML_DIGEST_VALUES_P_UNMARSHAL  (TPML_DIGEST_P_UNMARSHAL + 1)
96              (UNMARSHAL_t)TPML_DIGEST_VALUES_Unmarshal,
97      #define TPML_PCR_SELECTION_P_UNMARSHAL  (TPML_DIGEST_VALUES_P_UNMARSHAL + 1)
98              (UNMARSHAL_t)TPML_PCR_SELECTION_Unmarshal,
99      #define TPMS_CONTEXT_P_UNMARSHAL        (TPML_PCR_SELECTION_P_UNMARSHAL + 1)
100             (UNMARSHAL_t)TPMS_CONTEXT_Unmarshal,
101     #define TPMT_PUBLIC_PARMS_P_UNMARSHAL   (TPMS_CONTEXT_P_UNMARSHAL + 1)
102             (UNMARSHAL_t)TPMT_PUBLIC_PARMS_Unmarshal,
103     #define TPMT_TK_AUTH_P_UNMARSHAL        (TPMT_PUBLIC_PARMS_P_UNMARSHAL + 1)
104             (UNMARSHAL_t)TPMT_TK_AUTH_Unmarshal,
105     #define TPMT_TK_CREATION_P_UNMARSHAL    (TPMT_TK_AUTH_P_UNMARSHAL + 1)
106             (UNMARSHAL_t)TPMT_TK_CREATION_Unmarshal,
107     #define TPMT_TK_HASHCHECK_P_UNMARSHAL   (TPMT_TK_CREATION_P_UNMARSHAL + 1)
108             (UNMARSHAL_t)TPMT_TK_HASHCHECK_Unmarshal,
109     #define TPMT_TK_VERIFIED_P_UNMARSHAL    (TPMT_TK_HASHCHECK_P_UNMARSHAL + 1)
110             (UNMARSHAL_t)TPMT_TK_VERIFIED_Unmarshal,
111     #define TPM_CAP_P_UNMARSHAL             (TPMT_TK_VERIFIED_P_UNMARSHAL + 1)
112             (UNMARSHAL_t)TPM_CAP_Unmarshal,
113     #define TPM_CLOCK_ADJUST_P_UNMARSHAL    (TPM_CAP_P_UNMARSHAL + 1)
114             (UNMARSHAL_t)TPM_CLOCK_ADJUST_Unmarshal,
115     #define TPM_EO_P_UNMARSHAL              (TPM_CLOCK_ADJUST_P_UNMARSHAL + 1)
116             (UNMARSHAL_t)TPM_EO_Unmarshal,
117     #define TPM_SE_P_UNMARSHAL              (TPM_EO_P_UNMARSHAL + 1)
```

```
118             (UNMARSHAL_t)TPM_SE_Unmarshal,
119     #define TPM_SU_P_UNMARSHAL              (TPM_SE_P_UNMARSHAL + 1)
120             (UNMARSHAL_t)TPM_SU_Unmarshal,
121     #define UINT16_P_UNMARSHAL              (TPM_SU_P_UNMARSHAL + 1)
122             (UNMARSHAL_t)UINT16_Unmarshal,
123     #define UINT64_P_UNMARSHAL              (UINT16_P_UNMARSHAL + 1)
124             (UNMARSHAL_t)UINT64_Unmarshal,
125     //PARAMETER_FIRST_FLAG_TYPE is the first parameter to need a flag.
126     #define PARAMETER_FIRST_FLAG_TYPE       (UINT64_P_UNMARSHAL + 1)
127     #define TPM2B_PUBLIC_P_UNMARSHAL        (UINT64_P_UNMARSHAL + 1)
128             (UNMARSHAL_t)TPM2B_PUBLIC_Unmarshal,
129     #define TPMI_ALG_HASH_P_UNMARSHAL       (TPM2B_PUBLIC_P_UNMARSHAL + 1)
130             (UNMARSHAL_t)TPMI_ALG_HASH_Unmarshal,
131     #define TPMI_ALG_SYM_MODE_P_UNMARSHAL   (TPMI_ALG_HASH_P_UNMARSHAL + 1)
132             (UNMARSHAL_t)TPMI_ALG_SYM_MODE_Unmarshal,
133     #define TPMI_DH_PCR_P_UNMARSHAL         (TPMI_ALG_SYM_MODE_P_UNMARSHAL + 1)
134             (UNMARSHAL_t)TPMI_DH_PCR_Unmarshal,
135     #define TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL  (TPMI_DH_PCR_P_UNMARSHAL + 1)
136             (UNMARSHAL_t)TPMI_ECC_KEY_EXCHANGE_Unmarshal,
137     #define TPMI_RH_ENABLES_P_UNMARSHAL     (TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL + 1)
138             (UNMARSHAL_t)TPMI_RH_ENABLES_Unmarshal,
139     #define TPMI_RH_HIERARCHY_P_UNMARSHAL   (TPMI_RH_ENABLES_P_UNMARSHAL + 1)
140             (UNMARSHAL_t)TPMI_RH_HIERARCHY_Unmarshal,
141     #define TPMT_RSA_DECRYPT_P_UNMARSHAL    (TPMI_RH_HIERARCHY_P_UNMARSHAL + 1)
142             (UNMARSHAL_t)TPMT_RSA_DECRYPT_Unmarshal,
143     #define TPMT_SIGNATURE_P_UNMARSHAL      (TPMT_RSA_DECRYPT_P_UNMARSHAL + 1)
144             (UNMARSHAL_t)TPMT_SIGNATURE_Unmarshal,
145     #define TPMT_SIG_SCHEME_P_UNMARSHAL     (TPMT_SIGNATURE_P_UNMARSHAL + 1)
146             (UNMARSHAL_t)TPMT_SIG_SCHEME_Unmarshal,
147     #define TPMT_SYM_DEF_P_UNMARSHAL        (TPMT_SIG_SCHEME_P_UNMARSHAL + 1)
148             (UNMARSHAL_t)TPMT_SYM_DEF_Unmarshal,
149     #define TPMT_SYM_DEF_OBJECT_P_UNMARSHAL  (TPMT_SYM_DEF_P_UNMARSHAL + 1)
150             (UNMARSHAL_t)TPMT_SYM_DEF_OBJECT_Unmarshal
151     #define PARAMETER_LAST_TYPE             (TPMT_SYM_DEF_OBJECT_P_UNMARSHAL)
152     };
```

The MarshalArray() contains the dispatch functions for the marshaling code. The defines in this array are used to make it easier to cross reference the marshaling values in the types array of each command

```
153     const MARSHAL_t MarshalArray[] = {
154     #define  UINT32_H_MARSHAL                   0
155             (MARSHAL_t)UINT32_Marshal,
156     #define RESPONSE_PARAMETER_FIRST_TYPE   (UINT32_H_MARSHAL + 1)
157     #define TPM2B_ATTEST_P_MARSHAL          (UINT32_H_MARSHAL + 1)
158             (MARSHAL_t)TPM2B_ATTEST_Marshal,
159     #define TPM2B_CREATION_DATA_P_MARSHAL   (TPM2B_ATTEST_P_MARSHAL + 1)
160             (MARSHAL_t)TPM2B_CREATION_DATA_Marshal,
161     #define TPM2B_DATA_P_MARSHAL            (TPM2B_CREATION_DATA_P_MARSHAL + 1)
162             (MARSHAL_t)TPM2B_DATA_Marshal,
163     #define TPM2B_DIGEST_P_MARSHAL          (TPM2B_DATA_P_MARSHAL + 1)
164             (MARSHAL_t)TPM2B_DIGEST_Marshal,
165     #define TPM2B_ECC_POINT_P_MARSHAL       (TPM2B_DIGEST_P_MARSHAL + 1)
166             (MARSHAL_t)TPM2B_ECC_POINT_Marshal,
167     #define TPM2B_ENCRYPTED_SECRET_P_MARSHAL  (TPM2B_ECC_POINT_P_MARSHAL + 1)
168             (MARSHAL_t)TPM2B_ENCRYPTED_SECRET_Marshal,
169     #define TPM2B_ID_OBJECT_P_MARSHAL       (TPM2B_ENCRYPTED_SECRET_P_MARSHAL + 1)
170             (MARSHAL_t)TPM2B_ID_OBJECT_Marshal,
171     #define TPM2B_IV_P_MARSHAL              (TPM2B_ID_OBJECT_P_MARSHAL + 1)
172             (MARSHAL_t)TPM2B_IV_Marshal,
173     #define TPM2B_MAX_BUFFER_P_MARSHAL      (TPM2B_IV_P_MARSHAL + 1)
174             (MARSHAL_t)TPM2B_MAX_BUFFER_Marshal,
175     #define TPM2B_MAX_NV_BUFFER_P_MARSHAL   (TPM2B_MAX_BUFFER_P_MARSHAL + 1)
176             (MARSHAL_t)TPM2B_MAX_NV_BUFFER_Marshal,
177     #define TPM2B_NAME_P_MARSHAL            (TPM2B_MAX_NV_BUFFER_P_MARSHAL + 1)
178             (MARSHAL_t)TPM2B_NAME_Marshal,
```

```
179  #define TPM2B_NV_PUBLIC_P_MARSHAL       (TPM2B_NAME_P_MARSHAL + 1)
180              (MARSHAL_t)TPM2B_NV_PUBLIC_Marshal,
181  #define TPM2B_PRIVATE_P_MARSHAL         (TPM2B_NV_PUBLIC_P_MARSHAL + 1)
182              (MARSHAL_t)TPM2B_PRIVATE_Marshal,
183  #define TPM2B_PUBLIC_P_MARSHAL          (TPM2B_PRIVATE_P_MARSHAL + 1)
184              (MARSHAL_t)TPM2B_PUBLIC_Marshal,
185  #define TPM2B_PUBLIC_KEY_RSA_P_MARSHAL  (TPM2B_PUBLIC_P_MARSHAL + 1)
186              (MARSHAL_t)TPM2B_PUBLIC_KEY_RSA_Marshal,
187  #define TPM2B_SENSITIVE_DATA_P_MARSHAL  (TPM2B_PUBLIC_KEY_RSA_P_MARSHAL + 1)
188              (MARSHAL_t)TPM2B_SENSITIVE_DATA_Marshal,
189  #define UINT8_P_MARSHAL                 (TPM2B_SENSITIVE_DATA_P_MARSHAL + 1)
190              (MARSHAL_t)UINT8_Marshal,
191  #define TPML_ALG_P_MARSHAL              (UINT8_P_MARSHAL + 1)
192              (MARSHAL_t)TPML_ALG_Marshal,
193  #define TPML_DIGEST_P_MARSHAL           (TPML_ALG_P_MARSHAL + 1)
194              (MARSHAL_t)TPML_DIGEST_Marshal,
195  #define TPML_DIGEST_VALUES_P_MARSHAL    (TPML_DIGEST_P_MARSHAL + 1)
196              (MARSHAL_t)TPML_DIGEST_VALUES_Marshal,
197  #define TPML_PCR_SELECTION_P_MARSHAL    (TPML_DIGEST_VALUES_P_MARSHAL + 1)
198              (MARSHAL_t)TPML_PCR_SELECTION_Marshal,
199  #define TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL  (TPML_PCR_SELECTION_P_MARSHAL + 1)
200              (MARSHAL_t)TPMS_ALGORITHM_DETAIL_ECC_Marshal,
201  #define TPMS_CAPABILITY_DATA_P_MARSHAL  (TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL + 1)
202              (MARSHAL_t)TPMS_CAPABILITY_DATA_Marshal,
203  #define TPMS_CONTEXT_P_MARSHAL          (TPMS_CAPABILITY_DATA_P_MARSHAL + 1)
204              (MARSHAL_t)TPMS_CONTEXT_Marshal,
205  #define TPMS_TIME_INFO_P_MARSHAL        (TPMS_CONTEXT_P_MARSHAL + 1)
206              (MARSHAL_t)TPMS_TIME_INFO_Marshal,
207  #define TPMT_HA_P_MARSHAL               (TPMS_TIME_INFO_P_MARSHAL + 1)
208              (MARSHAL_t)TPMT_HA_Marshal,
209  #define TPMT_SIGNATURE_P_MARSHAL        (TPMT_HA_P_MARSHAL + 1)
210              (MARSHAL_t)TPMT_SIGNATURE_Marshal,
211  #define TPMT_TK_AUTH_P_MARSHAL          (TPMT_SIGNATURE_P_MARSHAL + 1)
212              (MARSHAL_t)TPMT_TK_AUTH_Marshal,
213  #define TPMT_TK_CREATION_P_MARSHAL      (TPMT_TK_AUTH_P_MARSHAL + 1)
214              (MARSHAL_t)TPMT_TK_CREATION_Marshal,
215  #define TPMT_TK_HASHCHECK_P_MARSHAL     (TPMT_TK_CREATION_P_MARSHAL + 1)
216              (MARSHAL_t)TPMT_TK_HASHCHECK_Marshal,
217  #define TPMT_TK_VERIFIED_P_MARSHAL      (TPMT_TK_HASHCHECK_P_MARSHAL + 1)
218              (MARSHAL_t)TPMT_TK_VERIFIED_Marshal,
219  #define UINT32_P_MARSHAL                (TPMT_TK_VERIFIED_P_MARSHAL + 1)
220              (MARSHAL_t)UINT32_Marshal,
221  #define UINT16_P_MARSHAL                (UINT32_P_MARSHAL + 1)
222              (MARSHAL_t)UINT16_Marshal
223  #define RESPONSE_PARAMETER_LAST_TYPE    (UINT16_P_MARSHAL)
224  };
225  #if CC_Startup == YES
226  #include "Startup_fp.h"
227  typedef TPM_RC  (Startup_Entry)(
228      Startup_In *in
229  );
230  typedef const struct {
231      Startup_Entry    *entry;
232      UINT16           inSize;
233      UINT16           outSize;
234      UINT16           offsetOfTypes;
235      BYTE             types[3];
236  } Startup_COMMAND_DESCRIPTOR_t;
237  Startup_COMMAND_DESCRIPTOR_t _StartupData = {
238  /* entry  */          &TPM2_Startup,
239  /* inSize */          (UINT16)(sizeof(Startup_In)),
240  /* outSize */         0,
241  /* offsetOfTypes */   offsetof(Startup_COMMAND_DESCRIPTOR_t, types),
242  /* offsets */         // No parameter offsets
243  /* types */           {TPM_SU_P_UNMARSHAL,
244                         END_OF_LIST,
```



```
                        END_OF_LIST}
};
#define _StartupDataAddress (&_StartupData)
#else
#define _StartupDataAddress 0
#endif
#if CC_Shutdown == YES
#include "Shutdown_fp.h"
typedef TPM_RC  (Shutdown_Entry)(
    Shutdown_In *in
);
typedef const struct {
    Shutdown_Entry    *entry;
    UINT16            inSize;
    UINT16            outSize;
    UINT16            offsetOfTypes;
    BYTE              types[3];
} Shutdown_COMMAND_DESCRIPTOR_t;
Shutdown_COMMAND_DESCRIPTOR_t _ShutdownData = {
/* entry  */          &TPM2_Shutdown,
/* inSize */          (UINT16)(sizeof(Shutdown_In)),
/* outSize */         0,
/* offsetOfTypes */   offsetof(Shutdown_COMMAND_DESCRIPTOR_t, types),
/* offsets */         // No parameter offsets
/* types */           {TPM_SU_P_UNMARSHAL,
                       END_OF_LIST,
                       END_OF_LIST}
};
#define _ShutdownDataAddress (&_ShutdownData)
#else
#define _ShutdownDataAddress 0
#endif
#if CC_SelfTest == YES
#include "SelfTest_fp.h"
typedef TPM_RC  (SelfTest_Entry)(
    SelfTest_In *in
);
typedef const struct {
    SelfTest_Entry    *entry;
    UINT16            inSize;
    UINT16            outSize;
    UINT16            offsetOfTypes;
    BYTE              types[3];
} SelfTest_COMMAND_DESCRIPTOR_t;
SelfTest_COMMAND_DESCRIPTOR_t _SelfTestData = {
/* entry  */          &TPM2_SelfTest,
/* inSize */          (UINT16)(sizeof(SelfTest_In)),
/* outSize */         0,
/* offsetOfTypes */   offsetof(SelfTest_COMMAND_DESCRIPTOR_t, types),
/* offsets */         // No parameter offsets
/* types */           {TPMI_YES_NO_P_UNMARSHAL,
                       END_OF_LIST,
                       END_OF_LIST}
};
#define _SelfTestDataAddress (&_SelfTestData)
#else
#define _SelfTestDataAddress 0
#endif
#if CC_IncrementalSelfTest == YES
#include "IncrementalSelfTest_fp.h"
typedef TPM_RC  (IncrementalSelfTest_Entry)(
    IncrementalSelfTest_In *in,
    IncrementalSelfTest_Out *out
);
typedef const struct {
    IncrementalSelfTest_Entry    *entry;
```



```
311         UINT16                      inSize;
312         UINT16                      outSize;
313         UINT16                      offsetOfTypes;
314         BYTE                        types[4];
315     } IncrementalSelfTest_COMMAND_DESCRIPTOR_t;
316     IncrementalSelfTest_COMMAND_DESCRIPTOR_t _IncrementalSelfTestData = {
317     /* entry  */          &TPM2_IncrementalSelfTest,
318     /* inSize */          (UINT16)(sizeof(IncrementalSelfTest_In)),
319     /* outSize */         (UINT16)(sizeof(IncrementalSelfTest_Out)),
320     /* offsetOfTypes */   offsetof(IncrementalSelfTest_COMMAND_DESCRIPTOR_t, types),
321     /* offsets */         // No parameter offsets
322     /* types */           {TPML_ALG_P_UNMARSHAL,
323                            END_OF_LIST,
324                            TPML_ALG_P_MARSHAL,
325                            END_OF_LIST}
326     };
327     #define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
328     #else
329     #define _IncrementalSelfTestDataAddress 0
330     #endif
331     #if CC_GetTestResult == YES
332     #include "GetTestResult_fp.h"
333     typedef TPM_RC  (GetTestResult_Entry)(
334         GetTestResult_Out *out
335     );
336     typedef const struct {
337         GetTestResult_Entry    *entry;
338         UINT16                 inSize;
339         UINT16                 outSize;
340         UINT16                 offsetOfTypes;
341         UINT16                 paramOffsets[1];
342         BYTE                   types[4];
343     } GetTestResult_COMMAND_DESCRIPTOR_t;
344     GetTestResult_COMMAND_DESCRIPTOR_t _GetTestResultData = {
345     /* entry  */          &TPM2_GetTestResult,
346     /* inSize */          0,
347     /* outSize */         (UINT16)(sizeof(GetTestResult_Out)),
348     /* offsetOfTypes */   offsetof(GetTestResult_COMMAND_DESCRIPTOR_t, types),
349     /* offsets */         {(UINT16)(offsetof(GetTestResult_Out, testResult))},
350     /* types */           {END_OF_LIST,
351                            TPM2B_MAX_BUFFER_P_MARSHAL,
352                            UINT32_P_MARSHAL,
353                            END_OF_LIST}
354     };
355     #define _GetTestResultDataAddress (&_GetTestResultData)
356     #else
357     #define _GetTestResultDataAddress 0
358     #endif
359     #if CC_StartAuthSession == YES
360     #include "StartAuthSession_fp.h"
361     typedef TPM_RC  (StartAuthSession_Entry)(
362         StartAuthSession_In *in,
363         StartAuthSession_Out *out
364     );
365     typedef const struct {
366         StartAuthSession_Entry    *entry;
367         UINT16                    inSize;
368         UINT16                    outSize;
369         UINT16                    offsetOfTypes;
370         UINT16                    paramOffsets[7];
371         BYTE                      types[11];
372     } StartAuthSession_COMMAND_DESCRIPTOR_t;
373     StartAuthSession_COMMAND_DESCRIPTOR_t _StartAuthSessionData = {
374     /* entry  */          &TPM2_StartAuthSession,
375     /* inSize */          (UINT16)(sizeof(StartAuthSession_In)),
376     /* outSize */         (UINT16)(sizeof(StartAuthSession_Out)),
```

```
377   /* offsetOfTypes */   offsetof(StartAuthSession_COMMAND_DESCRIPTOR_t, types),
378   /* offsets */         {(UINT16)(offsetof(StartAuthSession_In, bind)),
379                          (UINT16)(offsetof(StartAuthSession_In, nonceCaller)),
380                          (UINT16)(offsetof(StartAuthSession_In, encryptedSalt)),
381                          (UINT16)(offsetof(StartAuthSession_In, sessionType)),
382                          (UINT16)(offsetof(StartAuthSession_In, symmetric)),
383                          (UINT16)(offsetof(StartAuthSession_In, authHash)),
384                          (UINT16)(offsetof(StartAuthSession_Out, nonceTPM))},
385   /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
386                          TPMI_DH_ENTITY_H_UNMARSHAL + ADD_FLAG,
387                          TPM2B_DIGEST_P_UNMARSHAL,
388                          TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
389                          TPM_SE_P_UNMARSHAL,
390                          TPMT_SYM_DEF_P_UNMARSHAL + ADD_FLAG,
391                          TPMI_ALG_HASH_P_UNMARSHAL,
392                          END_OF_LIST,
393                          UINT32_H_MARSHAL,
394                          TPM2B_DIGEST_P_MARSHAL,
395                          END_OF_LIST}
396   };
397   #define _StartAuthSessionDataAddress (&_StartAuthSessionData)
398   #else
399   #define _StartAuthSessionDataAddress 0
400   #endif
401   #if CC_PolicyRestart == YES
402   #include "PolicyRestart_fp.h"
403   typedef TPM_RC  (PolicyRestart_Entry)(
404       PolicyRestart_In *in
405   );
406   typedef const struct {
407       PolicyRestart_Entry    *entry;
408       UINT16                 inSize;
409       UINT16                 outSize;
410       UINT16                 offsetOfTypes;
411       BYTE                   types[3];
412   } PolicyRestart_COMMAND_DESCRIPTOR_t;
413   PolicyRestart_COMMAND_DESCRIPTOR_t _PolicyRestartData = {
414   /* entry  */          &TPM2_PolicyRestart,
415   /* inSize */          (UINT16)(sizeof(PolicyRestart_In)),
416   /* outSize */         0,
417   /* offsetOfTypes */   offsetof(PolicyRestart_COMMAND_DESCRIPTOR_t, types),
418   /* offsets */         // No parameter offsets
419   /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
420                          END_OF_LIST,
421                          END_OF_LIST}
422   };
423   #define _PolicyRestartDataAddress (&_PolicyRestartData)
424   #else
425   #define _PolicyRestartDataAddress 0
426   #endif
427   #if CC_Create == YES
428   #include "Create_fp.h"
429   typedef TPM_RC  (Create_Entry)(
430       Create_In *in,
431       Create_Out *out
432   );
433   typedef const struct {
434       Create_Entry    *entry;
435       UINT16          inSize;
436       UINT16          outSize;
437       UINT16          offsetOfTypes;
438       UINT16          paramOffsets[8];
439       BYTE            types[12];
440   } Create_COMMAND_DESCRIPTOR_t;
441   Create_COMMAND_DESCRIPTOR_t _CreateData = {
442   /* entry  */          &TPM2_Create,
```



```
443   /* inSize */          (UINT16)(sizeof(Create_In)),
444   /* outSize */         (UINT16)(sizeof(Create_Out)),
445   /* offsetOfTypes */   offsetof(Create_COMMAND_DESCRIPTOR_t, types),
446   /* offsets */         {(UINT16)(offsetof(Create_In, inSensitive)),
447                          (UINT16)(offsetof(Create_In, inPublic)),
448                          (UINT16)(offsetof(Create_In, outsideInfo)),
449                          (UINT16)(offsetof(Create_In, creationPCR)),
450                          (UINT16)(offsetof(Create_Out, outPublic)),
451                          (UINT16)(offsetof(Create_Out, creationData)),
452                          (UINT16)(offsetof(Create_Out, creationHash)),
453                          (UINT16)(offsetof(Create_Out, creationTicket))},
454   /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
455                          TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
456                          TPM2B_PUBLIC_P_UNMARSHAL,
457                          TPM2B_DATA_P_UNMARSHAL,
458                          TPML_PCR_SELECTION_P_UNMARSHAL,
459                          END_OF_LIST,
460                          TPM2B_PRIVATE_P_MARSHAL,
461                          TPM2B_PUBLIC_P_MARSHAL,
462                          TPM2B_CREATION_DATA_P_MARSHAL,
463                          TPM2B_DIGEST_P_MARSHAL,
464                          TPMT_TK_CREATION_P_MARSHAL,
465                          END_OF_LIST}
466   };
467   #define _CreateDataAddress (&_CreateData)
468   #else
469   #define _CreateDataAddress 0
470   #endif
471   #if CC_Load == YES
472   #include "Load_fp.h"
473   typedef TPM_RC  (Load_Entry)(
474       Load_In *in,
475       Load_Out *out
476   );
477   typedef const struct {
478       Load_Entry    *entry;
479       UINT16        inSize;
480       UINT16        outSize;
481       UINT16        offsetOfTypes;
482       UINT16        paramOffsets[3];
483       BYTE          types[7];
484   } Load_COMMAND_DESCRIPTOR_t;
485   Load_COMMAND_DESCRIPTOR_t _LoadData = {
486   /* entry  */          &TPM2_Load,
487   /* inSize */          (UINT16)(sizeof(Load_In)),
488   /* outSize */         (UINT16)(sizeof(Load_Out)),
489   /* offsetOfTypes */   offsetof(Load_COMMAND_DESCRIPTOR_t, types),
490   /* offsets */         {(UINT16)(offsetof(Load_In, inPrivate)),
491                          (UINT16)(offsetof(Load_In, inPublic)),
492                          (UINT16)(offsetof(Load_Out, name))},
493   /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
494                          TPM2B_PRIVATE_P_UNMARSHAL,
495                          TPM2B_PUBLIC_P_UNMARSHAL,
496                          END_OF_LIST,
497                          UINT32_H_MARSHAL,
498                          TPM2B_NAME_P_MARSHAL,
499                          END_OF_LIST}
500   };
501   #define _LoadDataAddress (&_LoadData)
502   #else
503   #define _LoadDataAddress 0
504   #endif
505   #if CC_LoadExternal == YES
506   #include "LoadExternal_fp.h"
507   typedef TPM_RC  (LoadExternal_Entry)(
508       LoadExternal_In *in,
```

```
509         LoadExternal_Out *out
510     );
511     typedef const struct {
512         LoadExternal_Entry    *entry;
513         UINT16                inSize;
514         UINT16                outSize;
515         UINT16                offsetOfTypes;
516         UINT16                paramOffsets[3];
517         BYTE                  types[7];
518     } LoadExternal_COMMAND_DESCRIPTOR_t;
519     LoadExternal_COMMAND_DESCRIPTOR_t _LoadExternalData = {
520     /* entry  */          &TPM2_LoadExternal,
521     /* inSize */          (UINT16)(sizeof(LoadExternal_In)),
522     /* outSize */         (UINT16)(sizeof(LoadExternal_Out)),
523     /* offsetOfTypes */   offsetof(LoadExternal_COMMAND_DESCRIPTOR_t, types),
524     /* offsets */         {(UINT16)(offsetof(LoadExternal_In, inPublic)),
525                           (UINT16)(offsetof(LoadExternal_In, hierarchy)),
526                           (UINT16)(offsetof(LoadExternal_Out, name))},
527     /* types */           {TPM2B_SENSITIVE_P_UNMARSHAL,
528                           TPM2B_PUBLIC_P_UNMARSHAL + ADD_FLAG,
529                           TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
530                           END_OF_LIST,
531                           UINT32_H_MARSHAL,
532                           TPM2B_NAME_P_MARSHAL,
533                           END_OF_LIST}
534     };
535     #define _LoadExternalDataAddress (&_LoadExternalData)
536     #else
537     #define _LoadExternalDataAddress 0
538     #endif
539     #if CC_ReadPublic == YES
540     #include "ReadPublic_fp.h"
541     typedef TPM_RC  (ReadPublic_Entry)(
542         ReadPublic_In *in,
543         ReadPublic_Out *out
544     );
545     typedef const struct {
546         ReadPublic_Entry    *entry;
547         UINT16              inSize;
548         UINT16              outSize;
549         UINT16              offsetOfTypes;
550         UINT16              paramOffsets[2];
551         BYTE                types[6];
552     } ReadPublic_COMMAND_DESCRIPTOR_t;
553     ReadPublic_COMMAND_DESCRIPTOR_t _ReadPublicData = {
554     /* entry  */          &TPM2_ReadPublic,
555     /* inSize */          (UINT16)(sizeof(ReadPublic_In)),
556     /* outSize */         (UINT16)(sizeof(ReadPublic_Out)),
557     /* offsetOfTypes */   offsetof(ReadPublic_COMMAND_DESCRIPTOR_t, types),
558     /* offsets */         {(UINT16)(offsetof(ReadPublic_Out, name)),
559                           (UINT16)(offsetof(ReadPublic_Out, qualifiedName))},
560     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
561                           END_OF_LIST,
562                           TPM2B_PUBLIC_P_MARSHAL,
563                           TPM2B_NAME_P_MARSHAL,
564                           TPM2B_NAME_P_MARSHAL,
565                           END_OF_LIST}
566     };
567     #define _ReadPublicDataAddress (&_ReadPublicData)
568     #else
569     #define _ReadPublicDataAddress 0
570     #endif
571     #if CC_ActivateCredential == YES
572     #include "ActivateCredential_fp.h"
573     typedef TPM_RC  (ActivateCredential_Entry)(
574         ActivateCredential_In *in,
```

```
575         ActivateCredential_Out *out
576     );
577     typedef const struct {
578         ActivateCredential_Entry    *entry;
579         UINT16                      inSize;
580         UINT16                      outSize;
581         UINT16                      offsetOfTypes;
582         UINT16                      paramOffsets[3];
583         BYTE                        types[7];
584     } ActivateCredential_COMMAND_DESCRIPTOR_t;
585     ActivateCredential_COMMAND_DESCRIPTOR_t _ActivateCredentialData = {
586     /* entry  */          &TPM2_ActivateCredential,
587     /* inSize */          (UINT16)(sizeof(ActivateCredential_In)),
588     /* outSize */         (UINT16)(sizeof(ActivateCredential_Out)),
589     /* offsetOfTypes */   offsetof(ActivateCredential_COMMAND_DESCRIPTOR_t, types),
590     /* offsets */         {(UINT16)(offsetof(ActivateCredential_In, keyHandle)),
591                           (UINT16)(offsetof(ActivateCredential_In, credentialBlob)),
592                           (UINT16)(offsetof(ActivateCredential_In, secret))},
593     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
594                           TPMI_DH_OBJECT_H_UNMARSHAL,
595                           TPM2B_ID_OBJECT_P_UNMARSHAL,
596                           TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
597                           END_OF_LIST,
598                           TPM2B_DIGEST_P_MARSHAL,
599                           END_OF_LIST}
600     };
601     #define _ActivateCredentialDataAddress (&_ActivateCredentialData)
602     #else
603     #define _ActivateCredentialDataAddress 0
604     #endif
605     #if CC_MakeCredential == YES
606     #include "MakeCredential_fp.h"
607     typedef TPM_RC  (MakeCredential_Entry)(
608         MakeCredential_In *in,
609         MakeCredential_Out *out
610     );
611     typedef const struct {
612         MakeCredential_Entry    *entry;
613         UINT16                  inSize;
614         UINT16                  outSize;
615         UINT16                  offsetOfTypes;
616         UINT16                  paramOffsets[3];
617         BYTE                    types[7];
618     } MakeCredential_COMMAND_DESCRIPTOR_t;
619     MakeCredential_COMMAND_DESCRIPTOR_t _MakeCredentialData = {
620     /* entry  */          &TPM2_MakeCredential,
621     /* inSize */          (UINT16)(sizeof(MakeCredential_In)),
622     /* outSize */         (UINT16)(sizeof(MakeCredential_Out)),
623     /* offsetOfTypes */   offsetof(MakeCredential_COMMAND_DESCRIPTOR_t, types),
624     /* offsets */         {(UINT16)(offsetof(MakeCredential_In, credential)),
625                           (UINT16)(offsetof(MakeCredential_In, objectName)),
626                           (UINT16)(offsetof(MakeCredential_Out, secret))},
627     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
628                           TPM2B_DIGEST_P_UNMARSHAL,
629                           TPM2B_NAME_P_UNMARSHAL,
630                           END_OF_LIST,
631                           TPM2B_ID_OBJECT_P_MARSHAL,
632                           TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
633                           END_OF_LIST}
634     };
635     #define _MakeCredentialDataAddress (&_MakeCredentialData)
636     #else
637     #define _MakeCredentialDataAddress 0
638     #endif
639     #if CC_Unseal == YES
640     #include "Unseal_fp.h"
```

```
641   typedef TPM_RC  (Unseal_Entry)(
642       Unseal_In *in,
643       Unseal_Out *out
644   );
645   typedef const struct {
646       Unseal_Entry    *entry;
647       UINT16          inSize;
648       UINT16          outSize;
649       UINT16          offsetOfTypes;
650       BYTE            types[4];
651   } Unseal_COMMAND_DESCRIPTOR_t;
652   Unseal_COMMAND_DESCRIPTOR_t _UnsealData = {
653   /* entry  */          &TPM2_Unseal,
654   /* inSize */          (UINT16)(sizeof(Unseal_In)),
655   /* outSize */         (UINT16)(sizeof(Unseal_Out)),
656   /* offsetOfTypes */   offsetof(Unseal_COMMAND_DESCRIPTOR_t, types),
657   /* offsets */         // No parameter offsets
658   /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
659                          END_OF_LIST,
660                          TPM2B_SENSITIVE_DATA_P_MARSHAL,
661                          END_OF_LIST}
662   };
663   #define _UnsealDataAddress (&_UnsealData)
664   #else
665   #define _UnsealDataAddress 0
666   #endif
667   #if CC_ObjectChangeAuth == YES
668   #include "ObjectChangeAuth_fp.h"
669   typedef TPM_RC  (ObjectChangeAuth_Entry)(
670       ObjectChangeAuth_In *in,
671       ObjectChangeAuth_Out *out
672   );
673   typedef const struct {
674       ObjectChangeAuth_Entry    *entry;
675       UINT16                    inSize;
676       UINT16                    outSize;
677       UINT16                    offsetOfTypes;
678       UINT16                    paramOffsets[2];
679       BYTE                      types[6];
680   } ObjectChangeAuth_COMMAND_DESCRIPTOR_t;
681   ObjectChangeAuth_COMMAND_DESCRIPTOR_t _ObjectChangeAuthData = {
682   /* entry  */          &TPM2_ObjectChangeAuth,
683   /* inSize */          (UINT16)(sizeof(ObjectChangeAuth_In)),
684   /* outSize */         (UINT16)(sizeof(ObjectChangeAuth_Out)),
685   /* offsetOfTypes */   offsetof(ObjectChangeAuth_COMMAND_DESCRIPTOR_t, types),
686   /* offsets */         {(UINT16)(offsetof(ObjectChangeAuth_In, parentHandle)),
687                          (UINT16)(offsetof(ObjectChangeAuth_In, newAuth))},
688   /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
689                          TPMI_DH_OBJECT_H_UNMARSHAL,
690                          TPM2B_DIGEST_P_UNMARSHAL,
691                          END_OF_LIST,
692                          TPM2B_PRIVATE_P_MARSHAL,
693                          END_OF_LIST}
694   };
695   #define _ObjectChangeAuthDataAddress (&_ObjectChangeAuthData)
696   #else
697   #define _ObjectChangeAuthDataAddress 0
698   #endif
699   #if CC_CreateLoaded == YES
700   #include "CreateLoaded_fp.h"
701   typedef TPM_RC  (CreateLoaded_Entry)(
702       CreateLoaded_In *in,
703       CreateLoaded_Out *out
704   );
705   typedef const struct {
706       CreateLoaded_Entry    *entry;
```

```
707         UINT16                  inSize;
708         UINT16                  outSize;
709         UINT16                  offsetOfTypes;
710         UINT16                  paramOffsets[5];
711         BYTE                    types[9];
712     } CreateLoaded_COMMAND_DESCRIPTOR_t;
713     CreateLoaded_COMMAND_DESCRIPTOR_t _CreateLoadedData = {
714     /* entry  */            &TPM2_CreateLoaded,
715     /* inSize */            (UINT16)(sizeof(CreateLoaded_In)),
716     /* outSize */           (UINT16)(sizeof(CreateLoaded_Out)),
717     /* offsetOfTypes */     offsetof(CreateLoaded_COMMAND_DESCRIPTOR_t, types),
718     /* offsets */           {(UINT16)(offsetof(CreateLoaded_In, inSensitive)),
719                              (UINT16)(offsetof(CreateLoaded_In, inPublic)),
720                              (UINT16)(offsetof(CreateLoaded_Out, outPrivate)),
721                              (UINT16)(offsetof(CreateLoaded_Out, outPublic)),
722                              (UINT16)(offsetof(CreateLoaded_Out, name))},
723     /* types */             {TPMI_DH_PARENT_H_UNMARSHAL + ADD_FLAG,
724                              TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
725                              TPM2B_TEMPLATE_P_UNMARSHAL,
726                              END_OF_LIST,
727                              UINT32_H_MARSHAL,
728                              TPM2B_PRIVATE_P_MARSHAL,
729                              TPM2B_PUBLIC_P_MARSHAL,
730                              TPM2B_NAME_P_MARSHAL,
731                              END_OF_LIST}
732     };
733     #define _CreateLoadedDataAddress (&_CreateLoadedData)
734     #else
735     #define _CreateLoadedDataAddress 0
736     #endif
737     #if CC_Duplicate == YES
738     #include "Duplicate_fp.h"
739     typedef TPM_RC  (Duplicate_Entry)(
740         Duplicate_In *in,
741         Duplicate_Out *out
742     );
743     typedef const struct {
744         Duplicate_Entry    *entry;
745         UINT16             inSize;
746         UINT16             outSize;
747         UINT16             offsetOfTypes;
748         UINT16             paramOffsets[5];
749         BYTE               types[9];
750     } Duplicate_COMMAND_DESCRIPTOR_t;
751     Duplicate_COMMAND_DESCRIPTOR_t _DuplicateData = {
752     /* entry  */            &TPM2_Duplicate,
753     /* inSize */            (UINT16)(sizeof(Duplicate_In)),
754     /* outSize */           (UINT16)(sizeof(Duplicate_Out)),
755     /* offsetOfTypes */     offsetof(Duplicate_COMMAND_DESCRIPTOR_t, types),
756     /* offsets */           {(UINT16)(offsetof(Duplicate_In, newParentHandle)),
757                              (UINT16)(offsetof(Duplicate_In, encryptionKeyIn)),
758                              (UINT16)(offsetof(Duplicate_In, symmetricAlg)),
759                              (UINT16)(offsetof(Duplicate_Out, duplicate)),
760                              (UINT16)(offsetof(Duplicate_Out, outSymSeed))},
761     /* types */             {TPMI_DH_OBJECT_H_UNMARSHAL,
762                              TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
763                              TPM2B_DATA_P_UNMARSHAL,
764                              TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
765                              END_OF_LIST,
766                              TPM2B_DATA_P_MARSHAL,
767                              TPM2B_PRIVATE_P_MARSHAL,
768                              TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
769                              END_OF_LIST}
770     };
771     #define _DuplicateDataAddress (&_DuplicateData)
772     #else
```

```
773   #define _DuplicateDataAddress 0
774   #endif
775   #if CC_Rewrap == YES
776   #include "Rewrap_fp.h"
777   typedef TPM_RC  (Rewrap_Entry)(
778       Rewrap_In *in,
779       Rewrap_Out *out
780   );
781   typedef const struct {
782       Rewrap_Entry    *entry;
783       UINT16          inSize;
784       UINT16          outSize;
785       UINT16          offsetOfTypes;
786       UINT16          paramOffsets[5];
787       BYTE            types[9];
788   } Rewrap_COMMAND_DESCRIPTOR_t;
789   Rewrap_COMMAND_DESCRIPTOR_t _RewrapData = {
790   /* entry  */          &TPM2_Rewrap,
791   /* inSize */          (UINT16)(sizeof(Rewrap_In)),
792   /* outSize */         (UINT16)(sizeof(Rewrap_Out)),
793   /* offsetOfTypes */   offsetof(Rewrap_COMMAND_DESCRIPTOR_t, types),
794   /* offsets */         {(UINT16)(offsetof(Rewrap_In, newParent)),
795                          (UINT16)(offsetof(Rewrap_In, inDuplicate)),
796                          (UINT16)(offsetof(Rewrap_In, name)),
797                          (UINT16)(offsetof(Rewrap_In, inSymSeed)),
798                          (UINT16)(offsetof(Rewrap_Out, outSymSeed))},
799   /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
800                          TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
801                          TPM2B_PRIVATE_P_UNMARSHAL,
802                          TPM2B_NAME_P_UNMARSHAL,
803                          TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
804                          END_OF_LIST,
805                          TPM2B_PRIVATE_P_MARSHAL,
806                          TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
807                          END_OF_LIST}
808   };
809   #define _RewrapDataAddress (&_RewrapData)
810   #else
811   #define _RewrapDataAddress 0
812   #endif
813   #if CC_Import == YES
814   #include "Import_fp.h"
815   typedef TPM_RC  (Import_Entry)(
816       Import_In *in,
817       Import_Out *out
818   );
819   typedef const struct {
820       Import_Entry    *entry;
821       UINT16          inSize;
822       UINT16          outSize;
823       UINT16          offsetOfTypes;
824       UINT16          paramOffsets[5];
825       BYTE            types[9];
826   } Import_COMMAND_DESCRIPTOR_t;
827   Import_COMMAND_DESCRIPTOR_t _ImportData = {
828   /* entry  */          &TPM2_Import,
829   /* inSize */          (UINT16)(sizeof(Import_In)),
830   /* outSize */         (UINT16)(sizeof(Import_Out)),
831   /* offsetOfTypes */   offsetof(Import_COMMAND_DESCRIPTOR_t, types),
832   /* offsets */         {(UINT16)(offsetof(Import_In, encryptionKey)),
833                          (UINT16)(offsetof(Import_In, objectPublic)),
834                          (UINT16)(offsetof(Import_In, duplicate)),
835                          (UINT16)(offsetof(Import_In, inSymSeed)),
836                          (UINT16)(offsetof(Import_In, symmetricAlg))},
837   /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
838                          TPM2B_DATA_P_UNMARSHAL,
```

```
                        TPM2B_PUBLIC_P_UNMARSHAL,
                        TPM2B_PRIVATE_P_UNMARSHAL,
                        TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
                        TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
                        END_OF_LIST,
                        TPM2B_PRIVATE_P_MARSHAL,
                        END_OF_LIST}
};
#define _ImportDataAddress (&_ImportData)
#else
#define _ImportDataAddress 0
#endif
#if CC_RSA_Encrypt == YES
#include "RSA_Encrypt_fp.h"
typedef TPM_RC  (RSA_Encrypt_Entry)(
    RSA_Encrypt_In *in,
    RSA_Encrypt_Out *out
);
typedef const struct {
    RSA_Encrypt_Entry    *entry;
    UINT16               inSize;
    UINT16               outSize;
    UINT16               offsetOfTypes;
    UINT16               paramOffsets[3];
    BYTE                 types[7];
} RSA_Encrypt_COMMAND_DESCRIPTOR_t;
RSA_Encrypt_COMMAND_DESCRIPTOR_t _RSA_EncryptData = {
/* entry  */          &TPM2_RSA_Encrypt,
/* inSize */          (UINT16)(sizeof(RSA_Encrypt_In)),
/* outSize */         (UINT16)(sizeof(RSA_Encrypt_Out)),
/* offsetOfTypes */   offsetof(RSA_Encrypt_COMMAND_DESCRIPTOR_t, types),
/* offsets */         {(UINT16)(offsetof(RSA_Encrypt_In, message)),
                       (UINT16)(offsetof(RSA_Encrypt_In, inScheme)),
                       (UINT16)(offsetof(RSA_Encrypt_In, label))},
/* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
                       TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
                       TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
                       TPM2B_DATA_P_UNMARSHAL,
                       END_OF_LIST,
                       TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
                       END_OF_LIST}
};
#define _RSA_EncryptDataAddress (&_RSA_EncryptData)
#else
#define _RSA_EncryptDataAddress 0
#endif
#if CC_RSA_Decrypt == YES
#include "RSA_Decrypt_fp.h"
typedef TPM_RC  (RSA_Decrypt_Entry)(
    RSA_Decrypt_In *in,
    RSA_Decrypt_Out *out
);
typedef const struct {
    RSA_Decrypt_Entry    *entry;
    UINT16               inSize;
    UINT16               outSize;
    UINT16               offsetOfTypes;
    UINT16               paramOffsets[3];
    BYTE                 types[7];
} RSA_Decrypt_COMMAND_DESCRIPTOR_t;
RSA_Decrypt_COMMAND_DESCRIPTOR_t _RSA_DecryptData = {
/* entry  */          &TPM2_RSA_Decrypt,
/* inSize */          (UINT16)(sizeof(RSA_Decrypt_In)),
/* outSize */         (UINT16)(sizeof(RSA_Decrypt_Out)),
/* offsetOfTypes */   offsetof(RSA_Decrypt_COMMAND_DESCRIPTOR_t, types),
/* offsets */         {(UINT16)(offsetof(RSA_Decrypt_In, cipherText)),
```

```
905                         (UINT16)(offsetof(RSA_Decrypt_In, inScheme)),
906                         (UINT16)(offsetof(RSA_Decrypt_In, label))},
907     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
908                          TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
909                          TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
910                          TPM2B_DATA_P_UNMARSHAL,
911                          END_OF_LIST,
912                          TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
913                          END_OF_LIST}
914     };
915     #define _RSA_DecryptDataAddress (&_RSA_DecryptData)
916     #else
917     #define _RSA_DecryptDataAddress 0
918     #endif
919     #if CC_ECDH_KeyGen == YES
920     #include "ECDH_KeyGen_fp.h"
921     typedef TPM_RC  (ECDH_KeyGen_Entry)(
922         ECDH_KeyGen_In *in,
923         ECDH_KeyGen_Out *out
924     );
925     typedef const struct {
926         ECDH_KeyGen_Entry    *entry;
927         UINT16               inSize;
928         UINT16               outSize;
929         UINT16               offsetOfTypes;
930         UINT16               paramOffsets[1];
931         BYTE                 types[5];
932     } ECDH_KeyGen_COMMAND_DESCRIPTOR_t;
933     ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
934     /* entry  */          &TPM2_ECDH_KeyGen,
935     /* inSize */          (UINT16)(sizeof(ECDH_KeyGen_In)),
936     /* outSize */         (UINT16)(sizeof(ECDH_KeyGen_Out)),
937     /* offsetOfTypes */   offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types),
938     /* offsets */         {(UINT16)(offsetof(ECDH_KeyGen_Out, pubPoint))},
939     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
940                          END_OF_LIST,
941                          TPM2B_ECC_POINT_P_MARSHAL,
942                          TPM2B_ECC_POINT_P_MARSHAL,
943                          END_OF_LIST}
944     };
945     #define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)
946     #else
947     #define _ECDH_KeyGenDataAddress 0
948     #endif
949     #if CC_ECDH_ZGen == YES
950     #include "ECDH_ZGen_fp.h"
951     typedef TPM_RC  (ECDH_ZGen_Entry)(
952         ECDH_ZGen_In *in,
953         ECDH_ZGen_Out *out
954     );
955     typedef const struct {
956         ECDH_ZGen_Entry    *entry;
957         UINT16             inSize;
958         UINT16             outSize;
959         UINT16             offsetOfTypes;
960         UINT16             paramOffsets[1];
961         BYTE               types[5];
962     } ECDH_ZGen_COMMAND_DESCRIPTOR_t;
963     ECDH_ZGen_COMMAND_DESCRIPTOR_t _ECDH_ZGenData = {
964     /* entry  */          &TPM2_ECDH_ZGen,
965     /* inSize */          (UINT16)(sizeof(ECDH_ZGen_In)),
966     /* outSize */         (UINT16)(sizeof(ECDH_ZGen_Out)),
967     /* offsetOfTypes */   offsetof(ECDH_ZGen_COMMAND_DESCRIPTOR_t, types),
968     /* offsets */         {(UINT16)(offsetof(ECDH_ZGen_In, inPoint))},
969     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
970                          TPM2B_ECC_POINT_P_UNMARSHAL,
```



```
971                         END_OF_LIST,
972                         TPM2B_ECC_POINT_P_MARSHAL,
973                         END_OF_LIST}
974     };
975     #define _ECDH_ZGenDataAddress (&_ECDH_ZGenData)
976     #else
977     #define _ECDH_ZGenDataAddress 0
978     #endif
979     #if CC_ECC_Parameters == YES
980     #include "ECC_Parameters_fp.h"
981     typedef TPM_RC  (ECC_Parameters_Entry)(
982         ECC_Parameters_In *in,
983         ECC_Parameters_Out *out
984     );
985     typedef const struct {
986         ECC_Parameters_Entry    *entry;
987         UINT16                  inSize;
988         UINT16                  outSize;
989         UINT16                  offsetOfTypes;
990         BYTE                    types[4];
991     } ECC_Parameters_COMMAND_DESCRIPTOR_t;
992     ECC_Parameters_COMMAND_DESCRIPTOR_t _ECC_ParametersData = {
993     /* entry  */          &TPM2_ECC_Parameters,
994     /* inSize */          (UINT16)(sizeof(ECC_Parameters_In)),
995     /* outSize */         (UINT16)(sizeof(ECC_Parameters_Out)),
996     /* offsetOfTypes */   offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types),
997     /* offsets */         // No parameter offsets
998     /* types */           {TPMI_ECC_CURVE_P_UNMARSHAL,
999                           END_OF_LIST,
1000                          TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL,
1001                          END_OF_LIST}
1002    };
1003    #define _ECC_ParametersDataAddress (&_ECC_ParametersData)
1004    #else
1005    #define _ECC_ParametersDataAddress 0
1006    #endif
1007    #if CC_ZGen_2Phase == YES
1008    #include "ZGen_2Phase_fp.h"
1009    typedef TPM_RC  (ZGen_2Phase_Entry)(
1010        ZGen_2Phase_In *in,
1011        ZGen_2Phase_Out *out
1012    );
1013    typedef const struct {
1014        ZGen_2Phase_Entry    *entry;
1015        UINT16               inSize;
1016        UINT16               outSize;
1017        UINT16               offsetOfTypes;
1018        UINT16               paramOffsets[5];
1019        BYTE                 types[9];
1020    } ZGen_2Phase_COMMAND_DESCRIPTOR_t;
1021    ZGen_2Phase_COMMAND_DESCRIPTOR_t _ZGen_2PhaseData = {
1022    /* entry  */          &TPM2_ZGen_2Phase,
1023    /* inSize */          (UINT16)(sizeof(ZGen_2Phase_In)),
1024    /* outSize */         (UINT16)(sizeof(ZGen_2Phase_Out)),
1025    /* offsetOfTypes */   offsetof(ZGen_2Phase_COMMAND_DESCRIPTOR_t, types),
1026    /* offsets */         {(UINT16)(offsetof(ZGen_2Phase_In, inQsB)),
1027                           (UINT16)(offsetof(ZGen_2Phase_In, inQeB)),
1028                           (UINT16)(offsetof(ZGen_2Phase_In, inScheme)),
1029                           (UINT16)(offsetof(ZGen_2Phase_In, counter)),
1030                           (UINT16)(offsetof(ZGen_2Phase_Out, outZ2))},
1031    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1032                           TPM2B_ECC_POINT_P_UNMARSHAL,
1033                           TPM2B_ECC_POINT_P_UNMARSHAL,
1034                           TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL,
1035                           UINT16_P_UNMARSHAL,
1036                           END_OF_LIST,
```

```
1037                        TPM2B_ECC_POINT_P_MARSHAL,
1038                        TPM2B_ECC_POINT_P_MARSHAL,
1039                        END_OF_LIST}
1040    };
1041    #define _ZGen_2PhaseDataAddress (&_ZGen_2PhaseData)
1042    #else
1043    #define _ZGen_2PhaseDataAddress 0
1044    #endif
1045    #if CC_EncryptDecrypt == YES
1046    #include "EncryptDecrypt_fp.h"
1047    typedef TPM_RC  (EncryptDecrypt_Entry)(
1048        EncryptDecrypt_In *in,
1049        EncryptDecrypt_Out *out
1050    );
1051    typedef const struct {
1052        EncryptDecrypt_Entry    *entry;
1053        UINT16                  inSize;
1054        UINT16                  outSize;
1055        UINT16                  offsetOfTypes;
1056        UINT16                  paramOffsets[5];
1057        BYTE                    types[9];
1058    } EncryptDecrypt_COMMAND_DESCRIPTOR_t;
1059    EncryptDecrypt_COMMAND_DESCRIPTOR_t _EncryptDecryptData = {
1060    /* entry  */          &TPM2_EncryptDecrypt,
1061    /* inSize */          (UINT16)(sizeof(EncryptDecrypt_In)),
1062    /* outSize */         (UINT16)(sizeof(EncryptDecrypt_Out)),
1063    /* offsetOfTypes */   offsetof(EncryptDecrypt_COMMAND_DESCRIPTOR_t, types),
1064    /* offsets */         {(UINT16)(offsetof(EncryptDecrypt_In, decrypt)),
1065                           (UINT16)(offsetof(EncryptDecrypt_In, mode)),
1066                           (UINT16)(offsetof(EncryptDecrypt_In, ivIn)),
1067                           (UINT16)(offsetof(EncryptDecrypt_In, inData)),
1068                           (UINT16)(offsetof(EncryptDecrypt_Out, ivOut))},
1069    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1070                           TPMI_YES_NO_P_UNMARSHAL,
1071                           TPMI_ALG_SYM_MODE_P_UNMARSHAL + ADD_FLAG,
1072                           TPM2B_IV_P_UNMARSHAL,
1073                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1074                           END_OF_LIST,
1075                           TPM2B_MAX_BUFFER_P_MARSHAL,
1076                           TPM2B_IV_P_MARSHAL,
1077                           END_OF_LIST}
1078    };
1079    #define _EncryptDecryptDataAddress (&_EncryptDecryptData)
1080    #else
1081    #define _EncryptDecryptDataAddress 0
1082    #endif
1083    #if CC_EncryptDecrypt2 == YES
1084    #include "EncryptDecrypt2_fp.h"
1085    typedef TPM_RC  (EncryptDecrypt2_Entry)(
1086        EncryptDecrypt2_In *in,
1087        EncryptDecrypt2_Out *out
1088    );
1089    typedef const struct {
1090        EncryptDecrypt2_Entry    *entry;
1091        UINT16                   inSize;
1092        UINT16                   outSize;
1093        UINT16                   offsetOfTypes;
1094        UINT16                   paramOffsets[5];
1095        BYTE                     types[9];
1096    } EncryptDecrypt2_COMMAND_DESCRIPTOR_t;
1097    EncryptDecrypt2_COMMAND_DESCRIPTOR_t _EncryptDecrypt2Data = {
1098    /* entry  */          &TPM2_EncryptDecrypt2,
1099    /* inSize */          (UINT16)(sizeof(EncryptDecrypt2_In)),
1100    /* outSize */         (UINT16)(sizeof(EncryptDecrypt2_Out)),
1101    /* offsetOfTypes */   offsetof(EncryptDecrypt2_COMMAND_DESCRIPTOR_t, types),
1102    /* offsets */         {(UINT16)(offsetof(EncryptDecrypt2_In, inData)),
```

```
1103                        (UINT16)(offsetof(EncryptDecrypt2_In, decrypt)),
1104                        (UINT16)(offsetof(EncryptDecrypt2_In, mode)),
1105                        (UINT16)(offsetof(EncryptDecrypt2_In, ivIn)),
1106                        (UINT16)(offsetof(EncryptDecrypt2_Out, ivOut))},
1107    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1108                        TPM2B_MAX_BUFFER_P_UNMARSHAL,
1109                        TPMI_YES_NO_P_UNMARSHAL,
1110                        TPMI_ALG_SYM_MODE_P_UNMARSHAL + ADD_FLAG,
1111                        TPM2B_IV_P_UNMARSHAL,
1112                        END_OF_LIST,
1113                        TPM2B_MAX_BUFFER_P_MARSHAL,
1114                        TPM2B_IV_P_MARSHAL,
1115                        END_OF_LIST}
1116    };
1117    #define _EncryptDecrypt2DataAddress (&_EncryptDecrypt2Data)
1118    #else
1119    #define _EncryptDecrypt2DataAddress 0
1120    #endif
1121    #if CC_Hash == YES
1122    #include "Hash_fp.h"
1123    typedef TPM_RC  (Hash_Entry)(
1124        Hash_In *in,
1125        Hash_Out *out
1126    );
1127    typedef const struct {
1128        Hash_Entry    *entry;
1129        UINT16        inSize;
1130        UINT16        outSize;
1131        UINT16        offsetOfTypes;
1132        UINT16        paramOffsets[3];
1133        BYTE          types[7];
1134    } Hash_COMMAND_DESCRIPTOR_t;
1135    Hash_COMMAND_DESCRIPTOR_t _HashData = {
1136    /* entry  */          &TPM2_Hash,
1137    /* inSize */          (UINT16)(sizeof(Hash_In)),
1138    /* outSize */         (UINT16)(sizeof(Hash_Out)),
1139    /* offsetOfTypes */   offsetof(Hash_COMMAND_DESCRIPTOR_t, types),
1140    /* offsets */         {(UINT16)(offsetof(Hash_In, hashAlg)),
1141                         (UINT16)(offsetof(Hash_In, hierarchy)),
1142                         (UINT16)(offsetof(Hash_Out, validation))},
1143    /* types */           {TPM2B_MAX_BUFFER_P_UNMARSHAL,
1144                         TPMI_ALG_HASH_P_UNMARSHAL,
1145                         TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1146                         END_OF_LIST,
1147                         TPM2B_DIGEST_P_MARSHAL,
1148                         TPMT_TK_HASHCHECK_P_MARSHAL,
1149                         END_OF_LIST}
1150    };
1151    #define _HashDataAddress (&_HashData)
1152    #else
1153    #define _HashDataAddress 0
1154    #endif
1155    #if CC_HMAC == YES
1156    #include "HMAC_fp.h"
1157    typedef TPM_RC  (HMAC_Entry)(
1158        HMAC_In *in,
1159        HMAC_Out *out
1160    );
1161    typedef const struct {
1162        HMAC_Entry    *entry;
1163        UINT16        inSize;
1164        UINT16        outSize;
1165        UINT16        offsetOfTypes;
1166        UINT16        paramOffsets[2];
1167        BYTE          types[6];
1168    } HMAC_COMMAND_DESCRIPTOR_t;
```

```
1169    HMAC_COMMAND_DESCRIPTOR_t _HMACData = {
1170    /* entry  */          &TPM2_HMAC,
1171    /* inSize */          (UINT16)(sizeof(HMAC_In)),
1172    /* outSize */         (UINT16)(sizeof(HMAC_Out)),
1173    /* offsetOfTypes */   offsetof(HMAC_COMMAND_DESCRIPTOR_t, types),
1174    /* offsets */         {(UINT16)(offsetof(HMAC_In, buffer)),
1175                           (UINT16)(offsetof(HMAC_In, hashAlg))},
1176    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1177                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1178                           TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1179                           END_OF_LIST,
1180                           TPM2B_DIGEST_P_MARSHAL,
1181                           END_OF_LIST}
1182    };
1183    #define _HMACDataAddress (&_HMACData)
1184    #else
1185    #define _HMACDataAddress 0
1186    #endif
1187    #if CC_GetRandom == YES
1188    #include "GetRandom_fp.h"
1189    typedef TPM_RC  (GetRandom_Entry)(
1190        GetRandom_In *in,
1191        GetRandom_Out *out
1192    );
1193    typedef const struct {
1194        GetRandom_Entry    *entry;
1195        UINT16             inSize;
1196        UINT16             outSize;
1197        UINT16             offsetOfTypes;
1198        BYTE               types[4];
1199    } GetRandom_COMMAND_DESCRIPTOR_t;
1200    GetRandom_COMMAND_DESCRIPTOR_t _GetRandomData = {
1201    /* entry  */          &TPM2_GetRandom,
1202    /* inSize */          (UINT16)(sizeof(GetRandom_In)),
1203    /* outSize */         (UINT16)(sizeof(GetRandom_Out)),
1204    /* offsetOfTypes */   offsetof(GetRandom_COMMAND_DESCRIPTOR_t, types),
1205    /* offsets */         // No parameter offsets
1206    /* types */           {UINT16_P_UNMARSHAL,
1207                           END_OF_LIST,
1208                           TPM2B_DIGEST_P_MARSHAL,
1209                           END_OF_LIST}
1210    };
1211    #define _GetRandomDataAddress (&_GetRandomData)
1212    #else
1213    #define _GetRandomDataAddress 0
1214    #endif
1215    #if CC_StirRandom == YES
1216    #include "StirRandom_fp.h"
1217    typedef TPM_RC  (StirRandom_Entry)(
1218        StirRandom_In *in
1219    );
1220    typedef const struct {
1221        StirRandom_Entry    *entry;
1222        UINT16              inSize;
1223        UINT16              outSize;
1224        UINT16              offsetOfTypes;
1225        BYTE                types[3];
1226    } StirRandom_COMMAND_DESCRIPTOR_t;
1227    StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {
1228    /* entry  */          &TPM2_StirRandom,
1229    /* inSize */          (UINT16)(sizeof(StirRandom_In)),
1230    /* outSize */         0,
1231    /* offsetOfTypes */   offsetof(StirRandom_COMMAND_DESCRIPTOR_t, types),
1232    /* offsets */         // No parameter offsets
1233    /* types */           {TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1234                           END_OF_LIST,
```

```
                        END_OF_LIST}
};
#define _StirRandomDataAddress (&_StirRandomData)
#else
#define _StirRandomDataAddress 0
#endif
#if CC_HMAC_Start == YES
#include "HMAC_Start_fp.h"
typedef TPM_RC  (HMAC_Start_Entry)(
    HMAC_Start_In *in,
    HMAC_Start_Out *out
);
typedef const struct {
    HMAC_Start_Entry    *entry;
    UINT16              inSize;
    UINT16              outSize;
    UINT16              offsetOfTypes;
    UINT16              paramOffsets[2];
    BYTE                types[6];
} HMAC_Start_COMMAND_DESCRIPTOR_t;
HMAC_Start_COMMAND_DESCRIPTOR_t _HMAC_StartData = {
/* entry  */          &TPM2_HMAC_Start,
/* inSize */          (UINT16)(sizeof(HMAC_Start_In)),
/* outSize */         (UINT16)(sizeof(HMAC_Start_Out)),
/* offsetOfTypes */   offsetof(HMAC_Start_COMMAND_DESCRIPTOR_t, types),
/* offsets */         {(UINT16)(offsetof(HMAC_Start_In, auth)),
                      (UINT16)(offsetof(HMAC_Start_In, hashAlg))},
/* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
                       TPM2B_DIGEST_P_UNMARSHAL,
                       TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
                       END_OF_LIST,
                       UINT32_H_MARSHAL,
                       END_OF_LIST}
};
#define _HMAC_StartDataAddress (&_HMAC_StartData)
#else
#define _HMAC_StartDataAddress 0
#endif
#if CC_HashSequenceStart == YES
#include "HashSequenceStart_fp.h"
typedef TPM_RC  (HashSequenceStart_Entry)(
    HashSequenceStart_In *in,
    HashSequenceStart_Out *out
);
typedef const struct {
    HashSequenceStart_Entry    *entry;
    UINT16                     inSize;
    UINT16                     outSize;
    UINT16                     offsetOfTypes;
    UINT16                     paramOffsets[1];
    BYTE                       types[5];
} HashSequenceStart_COMMAND_DESCRIPTOR_t;
HashSequenceStart_COMMAND_DESCRIPTOR_t _HashSequenceStartData = {
/* entry  */          &TPM2_HashSequenceStart,
/* inSize */          (UINT16)(sizeof(HashSequenceStart_In)),
/* outSize */         (UINT16)(sizeof(HashSequenceStart_Out)),
/* offsetOfTypes */   offsetof(HashSequenceStart_COMMAND_DESCRIPTOR_t, types),
/* offsets */         {(UINT16)(offsetof(HashSequenceStart_In, hashAlg))},
/* types */           {TPM2B_DIGEST_P_UNMARSHAL,
                       TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
                       END_OF_LIST,
                       UINT32_H_MARSHAL,
                       END_OF_LIST}
};
#define _HashSequenceStartDataAddress (&_HashSequenceStartData)
#else
```

```
1301    #define _HashSequenceStartDataAddress 0
1302    #endif
1303    #if CC_SequenceUpdate == YES
1304    #include "SequenceUpdate_fp.h"
1305    typedef TPM_RC  (SequenceUpdate_Entry)(
1306        SequenceUpdate_In *in
1307    );
1308    typedef const struct {
1309        SequenceUpdate_Entry    *entry;
1310        UINT16                  inSize;
1311        UINT16                  outSize;
1312        UINT16                  offsetOfTypes;
1313        UINT16                  paramOffsets[1];
1314        BYTE                    types[4];
1315    } SequenceUpdate_COMMAND_DESCRIPTOR_t;
1316    SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
1317    /* entry  */          &TPM2_SequenceUpdate,
1318    /* inSize */          (UINT16)(sizeof(SequenceUpdate_In)),
1319    /* outSize */         0,
1320    /* offsetOfTypes */   offsetof(SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
1321    /* offsets */         {(UINT16)(offsetof(SequenceUpdate_In, buffer))},
1322    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1323                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1324                           END_OF_LIST,
1325                           END_OF_LIST}
1326    };
1327    #define _SequenceUpdateDataAddress (&_SequenceUpdateData)
1328    #else
1329    #define _SequenceUpdateDataAddress 0
1330    #endif
1331    #if CC_SequenceComplete == YES
1332    #include "SequenceComplete_fp.h"
1333    typedef TPM_RC  (SequenceComplete_Entry)(
1334        SequenceComplete_In *in,
1335        SequenceComplete_Out *out
1336    );
1337    typedef const struct {
1338        SequenceComplete_Entry    *entry;
1339        UINT16                    inSize;
1340        UINT16                    outSize;
1341        UINT16                    offsetOfTypes;
1342        UINT16                    paramOffsets[3];
1343        BYTE                      types[7];
1344    } SequenceComplete_COMMAND_DESCRIPTOR_t;
1345    SequenceComplete_COMMAND_DESCRIPTOR_t _SequenceCompleteData = {
1346    /* entry  */          &TPM2_SequenceComplete,
1347    /* inSize */          (UINT16)(sizeof(SequenceComplete_In)),
1348    /* outSize */         (UINT16)(sizeof(SequenceComplete_Out)),
1349    /* offsetOfTypes */   offsetof(SequenceComplete_COMMAND_DESCRIPTOR_t, types),
1350    /* offsets */         {(UINT16)(offsetof(SequenceComplete_In, buffer)),
1351                           (UINT16)(offsetof(SequenceComplete_In, hierarchy)),
1352                           (UINT16)(offsetof(SequenceComplete_Out, validation))},
1353    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1354                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1355                           TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1356                           END_OF_LIST,
1357                           TPM2B_DIGEST_P_MARSHAL,
1358                           TPMT_TK_HASHCHECK_P_MARSHAL,
1359                           END_OF_LIST}
1360    };
1361    #define _SequenceCompleteDataAddress (&_SequenceCompleteData)
1362    #else
1363    #define _SequenceCompleteDataAddress 0
1364    #endif
1365    #if CC_EventSequenceComplete == YES
1366    #include "EventSequenceComplete_fp.h"
```



```
1367    typedef TPM_RC  (EventSequenceComplete_Entry)(
1368        EventSequenceComplete_In *in,
1369        EventSequenceComplete_Out *out
1370    );
1371    typedef const struct {
1372        EventSequenceComplete_Entry    *entry;
1373        UINT16                         inSize;
1374        UINT16                         outSize;
1375        UINT16                         offsetOfTypes;
1376        UINT16                         paramOffsets[2];
1377        BYTE                           types[6];
1378    } EventSequenceComplete_COMMAND_DESCRIPTOR_t;
1379    EventSequenceComplete_COMMAND_DESCRIPTOR_t _EventSequenceCompleteData = {
1380    /* entry  */          &TPM2_EventSequenceComplete,
1381    /* inSize */          (UINT16)(sizeof(EventSequenceComplete_In)),
1382    /* outSize */         (UINT16)(sizeof(EventSequenceComplete_Out)),
1383    /* offsetOfTypes */   offsetof(EventSequenceComplete_COMMAND_DESCRIPTOR_t, types),
1384    /* offsets */         {(UINT16)(offsetof(EventSequenceComplete_In, sequenceHandle)),
1385                           (UINT16)(offsetof(EventSequenceComplete_In, buffer))},
1386    /* types */           {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1387                           TPMI_DH_OBJECT_H_UNMARSHAL,
1388                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1389                           END_OF_LIST,
1390                           TPML_DIGEST_VALUES_P_MARSHAL,
1391                           END_OF_LIST}
1392    };
1393    #define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)
1394    #else
1395    #define _EventSequenceCompleteDataAddress 0
1396    #endif
1397    #if CC_Certify == YES
1398    #include "Certify_fp.h"
1399    typedef TPM_RC  (Certify_Entry)(
1400        Certify_In *in,
1401        Certify_Out *out
1402    );
1403    typedef const struct {
1404        Certify_Entry    *entry;
1405        UINT16           inSize;
1406        UINT16           outSize;
1407        UINT16           offsetOfTypes;
1408        UINT16           paramOffsets[4];
1409        BYTE             types[8];
1410    } Certify_COMMAND_DESCRIPTOR_t;
1411    Certify_COMMAND_DESCRIPTOR_t _CertifyData = {
1412    /* entry  */          &TPM2_Certify,
1413    /* inSize */          (UINT16)(sizeof(Certify_In)),
1414    /* outSize */         (UINT16)(sizeof(Certify_Out)),
1415    /* offsetOfTypes */   offsetof(Certify_COMMAND_DESCRIPTOR_t, types),
1416    /* offsets */         {(UINT16)(offsetof(Certify_In, signHandle)),
1417                           (UINT16)(offsetof(Certify_In, qualifyingData)),
1418                           (UINT16)(offsetof(Certify_In, inScheme)),
1419                           (UINT16)(offsetof(Certify_Out, signature))},
1420    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1421                           TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1422                           TPM2B_DATA_P_UNMARSHAL,
1423                           TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1424                           END_OF_LIST,
1425                           TPM2B_ATTEST_P_MARSHAL,
1426                           TPMT_SIGNATURE_P_MARSHAL,
1427                           END_OF_LIST}
1428    };
1429    #define _CertifyDataAddress (&_CertifyData)
1430    #else
1431    #define _CertifyDataAddress 0
1432    #endif
```



```
1433  #if CC_CertifyCreation == YES
1434  #include "CertifyCreation_fp.h"
1435  typedef TPM_RC  (CertifyCreation_Entry)(
1436      CertifyCreation_In *in,
1437      CertifyCreation_Out *out
1438  );
1439  typedef const struct {
1440      CertifyCreation_Entry    *entry;
1441      UINT16                   inSize;
1442      UINT16                   outSize;
1443      UINT16                   offsetOfTypes;
1444      UINT16                   paramOffsets[6];
1445      BYTE                     types[10];
1446  } CertifyCreation_COMMAND_DESCRIPTOR_t;
1447  CertifyCreation_COMMAND_DESCRIPTOR_t _CertifyCreationData = {
1448  /* entry  */          &TPM2_CertifyCreation,
1449  /* inSize */          (UINT16)(sizeof(CertifyCreation_In)),
1450  /* outSize */         (UINT16)(sizeof(CertifyCreation_Out)),
1451  /* offsetOfTypes */   offsetof(CertifyCreation_COMMAND_DESCRIPTOR_t, types),
1452  /* offsets */         {(UINT16)(offsetof(CertifyCreation_In, objectHandle)),
1453                         (UINT16)(offsetof(CertifyCreation_In, qualifyingData)),
1454                         (UINT16)(offsetof(CertifyCreation_In, creationHash)),
1455                         (UINT16)(offsetof(CertifyCreation_In, inScheme)),
1456                         (UINT16)(offsetof(CertifyCreation_In, creationTicket)),
1457                         (UINT16)(offsetof(CertifyCreation_Out, signature))},
1458  /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1459                         TPMI_DH_OBJECT_H_UNMARSHAL,
1460                         TPM2B_DATA_P_UNMARSHAL,
1461                         TPM2B_DIGEST_P_UNMARSHAL,
1462                         TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1463                         TPMT_TK_CREATION_P_UNMARSHAL,
1464                         END_OF_LIST,
1465                         TPM2B_ATTEST_P_MARSHAL,
1466                         TPMT_SIGNATURE_P_MARSHAL,
1467                         END_OF_LIST}
1468  };
1469  #define _CertifyCreationDataAddress (&_CertifyCreationData)
1470  #else
1471  #define _CertifyCreationDataAddress 0
1472  #endif
1473  #if CC_Quote == YES
1474  #include "Quote_fp.h"
1475  typedef TPM_RC  (Quote_Entry)(
1476      Quote_In *in,
1477      Quote_Out *out
1478  );
1479  typedef const struct {
1480      Quote_Entry    *entry;
1481      UINT16         inSize;
1482      UINT16         outSize;
1483      UINT16         offsetOfTypes;
1484      UINT16         paramOffsets[4];
1485      BYTE           types[8];
1486  } Quote_COMMAND_DESCRIPTOR_t;
1487  Quote_COMMAND_DESCRIPTOR_t _QuoteData = {
1488  /* entry  */          &TPM2_Quote,
1489  /* inSize */          (UINT16)(sizeof(Quote_In)),
1490  /* outSize */         (UINT16)(sizeof(Quote_Out)),
1491  /* offsetOfTypes */   offsetof(Quote_COMMAND_DESCRIPTOR_t, types),
1492  /* offsets */         {(UINT16)(offsetof(Quote_In, qualifyingData)),
1493                         (UINT16)(offsetof(Quote_In, inScheme)),
1494                         (UINT16)(offsetof(Quote_In, PCRselect)),
1495                         (UINT16)(offsetof(Quote_Out, signature))},
1496  /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1497                         TPM2B_DATA_P_UNMARSHAL,
1498                         TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
```

```
                        TPML_PCR_SELECTION_P_UNMARSHAL,
                        END_OF_LIST,
                        TPM2B_ATTEST_P_MARSHAL,
                        TPMT_SIGNATURE_P_MARSHAL,
                        END_OF_LIST}
};
#define _QuoteDataAddress (&_QuoteData)
#else
#define _QuoteDataAddress 0
#endif
#if CC_GetSessionAuditDigest == YES
#include "GetSessionAuditDigest_fp.h"
typedef TPM_RC  (GetSessionAuditDigest_Entry)(
    GetSessionAuditDigest_In *in,
    GetSessionAuditDigest_Out *out
);
typedef const struct {
    GetSessionAuditDigest_Entry    *entry;
    UINT16                         inSize;
    UINT16                         outSize;
    UINT16                         offsetOfTypes;
    UINT16                         paramOffsets[5];
    BYTE                           types[9];
} GetSessionAuditDigest_COMMAND_DESCRIPTOR_t;
GetSessionAuditDigest_COMMAND_DESCRIPTOR_t _GetSessionAuditDigestData = {
/* entry  */          &TPM2_GetSessionAuditDigest,
/* inSize */          (UINT16)(sizeof(GetSessionAuditDigest_In)),
/* outSize */         (UINT16)(sizeof(GetSessionAuditDigest_Out)),
/* offsetOfTypes */   offsetof(GetSessionAuditDigest_COMMAND_DESCRIPTOR_t, types),
/* offsets */         {(UINT16)(offsetof(GetSessionAuditDigest_In, signHandle)),
                       (UINT16)(offsetof(GetSessionAuditDigest_In, sessionHandle)),
                       (UINT16)(offsetof(GetSessionAuditDigest_In, qualifyingData)),
                       (UINT16)(offsetof(GetSessionAuditDigest_In, inScheme)),
                       (UINT16)(offsetof(GetSessionAuditDigest_Out, signature))},
/* types */           {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
                       TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
                       TPMI_SH_HMAC_H_UNMARSHAL,
                       TPM2B_DATA_P_UNMARSHAL,
                       TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
                       END_OF_LIST,
                       TPM2B_ATTEST_P_MARSHAL,
                       TPMT_SIGNATURE_P_MARSHAL,
                       END_OF_LIST}
};
#define _GetSessionAuditDigestDataAddress (&_GetSessionAuditDigestData)
#else
#define _GetSessionAuditDigestDataAddress 0
#endif
#if CC_GetCommandAuditDigest == YES
#include "GetCommandAuditDigest_fp.h"
typedef TPM_RC  (GetCommandAuditDigest_Entry)(
    GetCommandAuditDigest_In *in,
    GetCommandAuditDigest_Out *out
);
typedef const struct {
    GetCommandAuditDigest_Entry    *entry;
    UINT16                         inSize;
    UINT16                         outSize;
    UINT16                         offsetOfTypes;
    UINT16                         paramOffsets[4];
    BYTE                           types[8];
} GetCommandAuditDigest_COMMAND_DESCRIPTOR_t;
GetCommandAuditDigest_COMMAND_DESCRIPTOR_t _GetCommandAuditDigestData = {
/* entry  */          &TPM2_GetCommandAuditDigest,
/* inSize */          (UINT16)(sizeof(GetCommandAuditDigest_In)),
/* outSize */         (UINT16)(sizeof(GetCommandAuditDigest_Out)),
```

```
1565    /* offsetOfTypes */   offsetof(GetCommandAuditDigest_COMMAND_DESCRIPTOR_t, types),
1566    /* offsets */         {(UINT16)(offsetof(GetCommandAuditDigest_In, signHandle)),
1567                           (UINT16)(offsetof(GetCommandAuditDigest_In, qualifyingData)),
1568                           (UINT16)(offsetof(GetCommandAuditDigest_In, inScheme)),
1569                           (UINT16)(offsetof(GetCommandAuditDigest_Out, signature))},
1570    /* types */           {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1571                           TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1572                           TPM2B_DATA_P_UNMARSHAL,
1573                           TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1574                           END_OF_LIST,
1575                           TPM2B_ATTEST_P_MARSHAL,
1576                           TPMT_SIGNATURE_P_MARSHAL,
1577                           END_OF_LIST}
1578    };
1579    #define _GetCommandAuditDigestDataAddress (&_GetCommandAuditDigestData)
1580    #else
1581    #define _GetCommandAuditDigestDataAddress 0
1582    #endif
1583    #if CC_GetTime == YES
1584    #include "GetTime_fp.h"
1585    typedef TPM_RC  (GetTime_Entry)(
1586        GetTime_In *in,
1587        GetTime_Out *out
1588    );
1589    typedef const struct {
1590        GetTime_Entry    *entry;
1591        UINT16           inSize;
1592        UINT16           outSize;
1593        UINT16           offsetOfTypes;
1594        UINT16           paramOffsets[4];
1595        BYTE             types[8];
1596    } GetTime_COMMAND_DESCRIPTOR_t;
1597    GetTime_COMMAND_DESCRIPTOR_t _GetTimeData = {
1598    /* entry  */          &TPM2_GetTime,
1599    /* inSize */          (UINT16)(sizeof(GetTime_In)),
1600    /* outSize */         (UINT16)(sizeof(GetTime_Out)),
1601    /* offsetOfTypes */   offsetof(GetTime_COMMAND_DESCRIPTOR_t, types),
1602    /* offsets */         {(UINT16)(offsetof(GetTime_In, signHandle)),
1603                           (UINT16)(offsetof(GetTime_In, qualifyingData)),
1604                           (UINT16)(offsetof(GetTime_In, inScheme)),
1605                           (UINT16)(offsetof(GetTime_Out, signature))},
1606    /* types */           {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1607                           TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1608                           TPM2B_DATA_P_UNMARSHAL,
1609                           TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1610                           END_OF_LIST,
1611                           TPM2B_ATTEST_P_MARSHAL,
1612                           TPMT_SIGNATURE_P_MARSHAL,
1613                           END_OF_LIST}
1614    };
1615    #define _GetTimeDataAddress (&_GetTimeData)
1616    #else
1617    #define _GetTimeDataAddress 0
1618    #endif
1619    #if CC_Commit == YES
1620    #include "Commit_fp.h"
1621    typedef TPM_RC  (Commit_Entry)(
1622        Commit_In *in,
1623        Commit_Out *out
1624    );
1625    typedef const struct {
1626        Commit_Entry    *entry;
1627        UINT16          inSize;
1628        UINT16          outSize;
1629        UINT16          offsetOfTypes;
1630        UINT16          paramOffsets[6];
```



```
1631        BYTE            types[10];
1632    } Commit_COMMAND_DESCRIPTOR_t;
1633    Commit_COMMAND_DESCRIPTOR_t _CommitData = {
1634    /* entry  */           &TPM2_Commit,
1635    /* inSize */           (UINT16)(sizeof(Commit_In)),
1636    /* outSize */          (UINT16)(sizeof(Commit_Out)),
1637    /* offsetOfTypes */    offsetof(Commit_COMMAND_DESCRIPTOR_t, types),
1638    /* offsets */          {(UINT16)(offsetof(Commit_In, P1)),
1639                            (UINT16)(offsetof(Commit_In, s2)),
1640                            (UINT16)(offsetof(Commit_In, y2)),
1641                            (UINT16)(offsetof(Commit_Out, L)),
1642                            (UINT16)(offsetof(Commit_Out, E)),
1643                            (UINT16)(offsetof(Commit_Out, counter))},
1644    /* types */            {TPMI_DH_OBJECT_H_UNMARSHAL,
1645                            TPM2B_ECC_POINT_P_UNMARSHAL,
1646                            TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1647                            TPM2B_ECC_PARAMETER_P_UNMARSHAL,
1648                            END_OF_LIST,
1649                            TPM2B_ECC_POINT_P_MARSHAL,
1650                            TPM2B_ECC_POINT_P_MARSHAL,
1651                            TPM2B_ECC_POINT_P_MARSHAL,
1652                            UINT16_P_MARSHAL,
1653                            END_OF_LIST}
1654    };
1655    #define _CommitDataAddress (&_CommitData)
1656    #else
1657    #define _CommitDataAddress 0
1658    #endif
1659    #if CC_EC_Ephemeral == YES
1660    #include "EC_Ephemeral_fp.h"
1661    typedef TPM_RC  (EC_Ephemeral_Entry)(
1662        EC_Ephemeral_In *in,
1663        EC_Ephemeral_Out *out
1664    );
1665    typedef const struct {
1666        EC_Ephemeral_Entry    *entry;
1667        UINT16                inSize;
1668        UINT16                outSize;
1669        UINT16                offsetOfTypes;
1670        UINT16                paramOffsets[1];
1671        BYTE                  types[5];
1672    } EC_Ephemeral_COMMAND_DESCRIPTOR_t;
1673    EC_Ephemeral_COMMAND_DESCRIPTOR_t _EC_EphemeralData = {
1674    /* entry  */           &TPM2_EC_Ephemeral,
1675    /* inSize */           (UINT16)(sizeof(EC_Ephemeral_In)),
1676    /* outSize */          (UINT16)(sizeof(EC_Ephemeral_Out)),
1677    /* offsetOfTypes */    offsetof(EC_Ephemeral_COMMAND_DESCRIPTOR_t, types),
1678    /* offsets */          {(UINT16)(offsetof(EC_Ephemeral_Out, counter))},
1679    /* types */            {TPMI_ECC_CURVE_P_UNMARSHAL,
1680                            END_OF_LIST,
1681                            TPM2B_ECC_POINT_P_MARSHAL,
1682                            UINT16_P_MARSHAL,
1683                            END_OF_LIST}
1684    };
1685    #define _EC_EphemeralDataAddress (&_EC_EphemeralData)
1686    #else
1687    #define _EC_EphemeralDataAddress 0
1688    #endif
1689    #if CC_VerifySignature == YES
1690    #include "VerifySignature_fp.h"
1691    typedef TPM_RC  (VerifySignature_Entry)(
1692        VerifySignature_In *in,
1693        VerifySignature_Out *out
1694    );
1695    typedef const struct {
1696        VerifySignature_Entry    *entry;
```

```
1697        UINT16                  inSize;
1698        UINT16                  outSize;
1699        UINT16                  offsetOfTypes;
1700        UINT16                  paramOffsets[2];
1701        BYTE                    types[6];
1702    } VerifySignature_COMMAND_DESCRIPTOR_t;
1703    VerifySignature_COMMAND_DESCRIPTOR_t _VerifySignatureData = {
1704    /* entry  */          &TPM2_VerifySignature,
1705    /* inSize */          (UINT16)(sizeof(VerifySignature_In)),
1706    /* outSize */         (UINT16)(sizeof(VerifySignature_Out)),
1707    /* offsetOfTypes */   offsetof(VerifySignature_COMMAND_DESCRIPTOR_t, types),
1708    /* offsets */         {(UINT16)(offsetof(VerifySignature_In, digest)),
1709                           (UINT16)(offsetof(VerifySignature_In, signature))},
1710    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1711                           TPM2B_DIGEST_P_UNMARSHAL,
1712                           TPMT_SIGNATURE_P_UNMARSHAL,
1713                           END_OF_LIST,
1714                           TPMT_TK_VERIFIED_P_MARSHAL,
1715                           END_OF_LIST}
1716    };
1717    #define _VerifySignatureDataAddress (&_VerifySignatureData)
1718    #else
1719    #define _VerifySignatureDataAddress 0
1720    #endif
1721    #if CC_Sign == YES
1722    #include "Sign_fp.h"
1723    typedef TPM_RC  (Sign_Entry)(
1724        Sign_In *in,
1725        Sign_Out *out
1726    );
1727    typedef const struct {
1728        Sign_Entry    *entry;
1729        UINT16        inSize;
1730        UINT16        outSize;
1731        UINT16        offsetOfTypes;
1732        UINT16        paramOffsets[3];
1733        BYTE          types[7];
1734    } Sign_COMMAND_DESCRIPTOR_t;
1735    Sign_COMMAND_DESCRIPTOR_t _SignData = {
1736    /* entry  */          &TPM2_Sign,
1737    /* inSize */          (UINT16)(sizeof(Sign_In)),
1738    /* outSize */         (UINT16)(sizeof(Sign_Out)),
1739    /* offsetOfTypes */   offsetof(Sign_COMMAND_DESCRIPTOR_t, types),
1740    /* offsets */         {(UINT16)(offsetof(Sign_In, digest)),
1741                           (UINT16)(offsetof(Sign_In, inScheme)),
1742                           (UINT16)(offsetof(Sign_In, validation))},
1743    /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1744                           TPM2B_DIGEST_P_UNMARSHAL,
1745                           TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1746                           TPMT_TK_HASHCHECK_P_UNMARSHAL,
1747                           END_OF_LIST,
1748                           TPMT_SIGNATURE_P_MARSHAL,
1749                           END_OF_LIST}
1750    };
1751    #define _SignDataAddress (&_SignData)
1752    #else
1753    #define _SignDataAddress 0
1754    #endif
1755    #if CC_SetCommandCodeAuditStatus == YES
1756    #include "SetCommandCodeAuditStatus_fp.h"
1757    typedef TPM_RC  (SetCommandCodeAuditStatus_Entry)(
1758        SetCommandCodeAuditStatus_In *in
1759    );
1760    typedef const struct {
1761        SetCommandCodeAuditStatus_Entry    *entry;
1762        UINT16                             inSize;
```

```
1763        UINT16                          outSize;
1764        UINT16                          offsetOfTypes;
1765        UINT16                          paramOffsets[3];
1766        BYTE                            types[6];
1767    } SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t;
1768    SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t _SetCommandCodeAuditStatusData = {
1769    /* entry  */          &TPM2_SetCommandCodeAuditStatus,
1770    /* inSize */          (UINT16)(sizeof(SetCommandCodeAuditStatus_In)),
1771    /* outSize */         0,
1772    /* offsetOfTypes */   offsetof(SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t, types),
1773    /* offsets */         {(UINT16)(offsetof(SetCommandCodeAuditStatus_In, auditAlg)),
1774                           (UINT16)(offsetof(SetCommandCodeAuditStatus_In, setList)),
1775                           (UINT16)(offsetof(SetCommandCodeAuditStatus_In, clearList))},
1776    /* types */           {TPMI_RH_PROVISION_H_UNMARSHAL,
1777                           TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1778                           TPML_CC_P_UNMARSHAL,
1779                           TPML_CC_P_UNMARSHAL,
1780                           END_OF_LIST,
1781                           END_OF_LIST}
1782    };
1783    #define _SetCommandCodeAuditStatusDataAddress (&_SetCommandCodeAuditStatusData)
1784    #else
1785    #define _SetCommandCodeAuditStatusDataAddress 0
1786    #endif
1787    #if CC_PCR_Extend == YES
1788    #include "PCR_Extend_fp.h"
1789    typedef TPM_RC  (PCR_Extend_Entry)(
1790        PCR_Extend_In *in
1791    );
1792    typedef const struct {
1793        PCR_Extend_Entry    *entry;
1794        UINT16              inSize;
1795        UINT16              outSize;
1796        UINT16              offsetOfTypes;
1797        UINT16              paramOffsets[1];
1798        BYTE                types[4];
1799    } PCR_Extend_COMMAND_DESCRIPTOR_t;
1800    PCR_Extend_COMMAND_DESCRIPTOR_t _PCR_ExtendData = {
1801    /* entry  */          &TPM2_PCR_Extend,
1802    /* inSize */          (UINT16)(sizeof(PCR_Extend_In)),
1803    /* outSize */         0,
1804    /* offsetOfTypes */   offsetof(PCR_Extend_COMMAND_DESCRIPTOR_t, types),
1805    /* offsets */         {(UINT16)(offsetof(PCR_Extend_In, digests))},
1806    /* types */           {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1807                           TPML_DIGEST_VALUES_P_UNMARSHAL,
1808                           END_OF_LIST,
1809                           END_OF_LIST}
1810    };
1811    #define _PCR_ExtendDataAddress (&_PCR_ExtendData)
1812    #else
1813    #define _PCR_ExtendDataAddress 0
1814    #endif
1815    #if CC_PCR_Event == YES
1816    #include "PCR_Event_fp.h"
1817    typedef TPM_RC  (PCR_Event_Entry)(
1818        PCR_Event_In *in,
1819        PCR_Event_Out *out
1820    );
1821    typedef const struct {
1822        PCR_Event_Entry    *entry;
1823        UINT16             inSize;
1824        UINT16             outSize;
1825        UINT16             offsetOfTypes;
1826        UINT16             paramOffsets[1];
1827        BYTE               types[5];
1828    } PCR_Event_COMMAND_DESCRIPTOR_t;
```

```
1829    PCR_Event_COMMAND_DESCRIPTOR_t _PCR_EventData = {
1830    /* entry  */          &TPM2_PCR_Event,
1831    /* inSize */          (UINT16)(sizeof(PCR_Event_In)),
1832    /* outSize */         (UINT16)(sizeof(PCR_Event_Out)),
1833    /* offsetOfTypes */   offsetof(PCR_Event_COMMAND_DESCRIPTOR_t, types),
1834    /* offsets */         {(UINT16)(offsetof(PCR_Event_In, eventData))},
1835    /* types */           {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1836                           TPM2B_EVENT_P_UNMARSHAL,
1837                           END_OF_LIST,
1838                           TPML_DIGEST_VALUES_P_MARSHAL,
1839                           END_OF_LIST}
1840    };
1841    #define _PCR_EventDataAddress (&_PCR_EventData)
1842    #else
1843    #define _PCR_EventDataAddress 0
1844    #endif
1845    #if CC_PCR_Read == YES
1846    #include "PCR_Read_fp.h"
1847    typedef TPM_RC  (PCR_Read_Entry)(
1848        PCR_Read_In *in,
1849        PCR_Read_Out *out
1850    );
1851    typedef const struct {
1852        PCR_Read_Entry    *entry;
1853        UINT16            inSize;
1854        UINT16            outSize;
1855        UINT16            offsetOfTypes;
1856        UINT16            paramOffsets[2];
1857        BYTE              types[6];
1858    } PCR_Read_COMMAND_DESCRIPTOR_t;
1859    PCR_Read_COMMAND_DESCRIPTOR_t _PCR_ReadData = {
1860    /* entry  */          &TPM2_PCR_Read,
1861    /* inSize */          (UINT16)(sizeof(PCR_Read_In)),
1862    /* outSize */         (UINT16)(sizeof(PCR_Read_Out)),
1863    /* offsetOfTypes */   offsetof(PCR_Read_COMMAND_DESCRIPTOR_t, types),
1864    /* offsets */         {(UINT16)(offsetof(PCR_Read_Out, pcrSelectionOut)),
1865                           (UINT16)(offsetof(PCR_Read_Out, pcrValues))},
1866    /* types */           {TPML_PCR_SELECTION_P_UNMARSHAL,
1867                           END_OF_LIST,
1868                           UINT32_P_MARSHAL,
1869                           TPML_PCR_SELECTION_P_MARSHAL,
1870                           TPML_DIGEST_P_MARSHAL,
1871                           END_OF_LIST}
1872    };
1873    #define _PCR_ReadDataAddress (&_PCR_ReadData)
1874    #else
1875    #define _PCR_ReadDataAddress 0
1876    #endif
1877    #if CC_PCR_Allocate == YES
1878    #include "PCR_Allocate_fp.h"
1879    typedef TPM_RC  (PCR_Allocate_Entry)(
1880        PCR_Allocate_In *in,
1881        PCR_Allocate_Out *out
1882    );
1883    typedef const struct {
1884        PCR_Allocate_Entry    *entry;
1885        UINT16                inSize;
1886        UINT16                outSize;
1887        UINT16                offsetOfTypes;
1888        UINT16                paramOffsets[4];
1889        BYTE                  types[8];
1890    } PCR_Allocate_COMMAND_DESCRIPTOR_t;
1891    PCR_Allocate_COMMAND_DESCRIPTOR_t _PCR_AllocateData = {
1892    /* entry  */          &TPM2_PCR_Allocate,
1893    /* inSize */          (UINT16)(sizeof(PCR_Allocate_In)),
1894    /* outSize */         (UINT16)(sizeof(PCR_Allocate_Out)),
```

```
1895    /* offsetOfTypes */   offsetof(PCR_Allocate_COMMAND_DESCRIPTOR_t, types),
1896    /* offsets */         {(UINT16)(offsetof(PCR_Allocate_In, pcrAllocation)),
1897                           (UINT16)(offsetof(PCR_Allocate_Out, maxPCR)),
1898                           (UINT16)(offsetof(PCR_Allocate_Out, sizeNeeded)),
1899                           (UINT16)(offsetof(PCR_Allocate_Out, sizeAvailable))},
1900    /* types */           {TPMI_RH_PLATFORM_H_UNMARSHAL,
1901                           TPML_PCR_SELECTION_P_UNMARSHAL,
1902                           END_OF_LIST,
1903                           UINT8_P_MARSHAL,
1904                           UINT32_P_MARSHAL,
1905                           UINT32_P_MARSHAL,
1906                           UINT32_P_MARSHAL,
1907                           END_OF_LIST}
1908    };
1909    #define _PCR_AllocateDataAddress (&_PCR_AllocateData)
1910    #else
1911    #define _PCR_AllocateDataAddress 0
1912    #endif
1913    #if CC_PCR_SetAuthPolicy == YES
1914    #include "PCR_SetAuthPolicy_fp.h"
1915    typedef TPM_RC  (PCR_SetAuthPolicy_Entry)(
1916        PCR_SetAuthPolicy_In *in
1917    );
1918    typedef const struct {
1919        PCR_SetAuthPolicy_Entry    *entry;
1920        UINT16                     inSize;
1921        UINT16                     outSize;
1922        UINT16                     offsetOfTypes;
1923        UINT16                     paramOffsets[3];
1924        BYTE                       types[6];
1925    } PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t;
1926    PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t _PCR_SetAuthPolicyData = {
1927    /* entry  */          &TPM2_PCR_SetAuthPolicy,
1928    /* inSize */          (UINT16)(sizeof(PCR_SetAuthPolicy_In)),
1929    /* outSize */         0,
1930    /* offsetOfTypes */   offsetof(PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t, types),
1931    /* offsets */         {(UINT16)(offsetof(PCR_SetAuthPolicy_In, authPolicy)),
1932                           (UINT16)(offsetof(PCR_SetAuthPolicy_In, hashAlg)),
1933                           (UINT16)(offsetof(PCR_SetAuthPolicy_In, pcrNum))},
1934    /* types */           {TPMI_RH_PLATFORM_H_UNMARSHAL,
1935                           TPM2B_DIGEST_P_UNMARSHAL,
1936                           TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1937                           TPMI_DH_PCR_P_UNMARSHAL,
1938                           END_OF_LIST,
1939                           END_OF_LIST}
1940    };
1941    #define _PCR_SetAuthPolicyDataAddress (&_PCR_SetAuthPolicyData)
1942    #else
1943    #define _PCR_SetAuthPolicyDataAddress 0
1944    #endif
1945    #if CC_PCR_SetAuthValue == YES
1946    #include "PCR_SetAuthValue_fp.h"
1947    typedef TPM_RC  (PCR_SetAuthValue_Entry)(
1948        PCR_SetAuthValue_In *in
1949    );
1950    typedef const struct {
1951        PCR_SetAuthValue_Entry    *entry;
1952        UINT16                    inSize;
1953        UINT16                    outSize;
1954        UINT16                    offsetOfTypes;
1955        UINT16                    paramOffsets[1];
1956        BYTE                      types[4];
1957    } PCR_SetAuthValue_COMMAND_DESCRIPTOR_t;
1958    PCR_SetAuthValue_COMMAND_DESCRIPTOR_t _PCR_SetAuthValueData = {
1959    /* entry  */          &TPM2_PCR_SetAuthValue,
1960    /* inSize */          (UINT16)(sizeof(PCR_SetAuthValue_In)),
```

```
1961    /* outSize */          0,
1962    /* offsetOfTypes */    offsetof(PCR_SetAuthValue_COMMAND_DESCRIPTOR_t, types),
1963    /* offsets */          {(UINT16)(offsetof(PCR_SetAuthValue_In, auth))},
1964    /* types */            {TPMI_DH_PCR_H_UNMARSHAL,
1965                            TPM2B_DIGEST_P_UNMARSHAL,
1966                            END_OF_LIST,
1967                            END_OF_LIST}
1968    };
1969    #define _PCR_SetAuthValueDataAddress (&_PCR_SetAuthValueData)
1970    #else
1971    #define _PCR_SetAuthValueDataAddress 0
1972    #endif
1973    #if CC_PCR_Reset == YES
1974    #include "PCR_Reset_fp.h"
1975    typedef TPM_RC  (PCR_Reset_Entry)(
1976        PCR_Reset_In *in
1977    );
1978    typedef const struct {
1979        PCR_Reset_Entry    *entry;
1980        UINT16              inSize;
1981        UINT16              outSize;
1982        UINT16              offsetOfTypes;
1983        BYTE                types[3];
1984    } PCR_Reset_COMMAND_DESCRIPTOR_t;
1985    PCR_Reset_COMMAND_DESCRIPTOR_t _PCR_ResetData = {
1986    /* entry  */           &TPM2_PCR_Reset,
1987    /* inSize */           (UINT16)(sizeof(PCR_Reset_In)),
1988    /* outSize */          0,
1989    /* offsetOfTypes */    offsetof(PCR_Reset_COMMAND_DESCRIPTOR_t, types),
1990    /* offsets */          // No parameter offsets
1991    /* types */            {TPMI_DH_PCR_H_UNMARSHAL,
1992                            END_OF_LIST,
1993                            END_OF_LIST}
1994    };
1995    #define _PCR_ResetDataAddress (&_PCR_ResetData)
1996    #else
1997    #define _PCR_ResetDataAddress 0
1998    #endif
1999    #if CC_PolicySigned == YES
2000    #include "PolicySigned_fp.h"
2001    typedef TPM_RC  (PolicySigned_Entry)(
2002        PolicySigned_In *in,
2003        PolicySigned_Out *out
2004    );
2005    typedef const struct {
2006        PolicySigned_Entry    *entry;
2007        UINT16                inSize;
2008        UINT16                outSize;
2009        UINT16                offsetOfTypes;
2010        UINT16                paramOffsets[7];
2011        BYTE                  types[11];
2012    } PolicySigned_COMMAND_DESCRIPTOR_t;
2013    PolicySigned_COMMAND_DESCRIPTOR_t _PolicySignedData = {
2014    /* entry  */           &TPM2_PolicySigned,
2015    /* inSize */           (UINT16)(sizeof(PolicySigned_In)),
2016    /* outSize */          (UINT16)(sizeof(PolicySigned_Out)),
2017    /* offsetOfTypes */    offsetof(PolicySigned_COMMAND_DESCRIPTOR_t, types),
2018    /* offsets */          {(UINT16)(offsetof(PolicySigned_In, policySession)),
2019                            (UINT16)(offsetof(PolicySigned_In, nonceTPM)),
2020                            (UINT16)(offsetof(PolicySigned_In, cpHashA)),
2021                            (UINT16)(offsetof(PolicySigned_In, policyRef)),
2022                            (UINT16)(offsetof(PolicySigned_In, expiration)),
2023                            (UINT16)(offsetof(PolicySigned_In, auth)),
2024                            (UINT16)(offsetof(PolicySigned_Out, policyTicket))},
2025    /* types */            {TPMI_DH_OBJECT_H_UNMARSHAL,
2026                            TPMI_SH_POLICY_H_UNMARSHAL,
```

```
                        TPM2B_DIGEST_P_UNMARSHAL,
                        TPM2B_DIGEST_P_UNMARSHAL,
                        TPM2B_DIGEST_P_UNMARSHAL,
                        UINT32_P_UNMARSHAL,
                        TPMT_SIGNATURE_P_UNMARSHAL,
                        END_OF_LIST,
                        TPM2B_DIGEST_P_MARSHAL,
                        TPMT_TK_AUTH_P_MARSHAL,
                        END_OF_LIST}
};
#define _PolicySignedDataAddress (&_PolicySignedData)
#else
#define _PolicySignedDataAddress 0
#endif
#if CC_PolicySecret == YES
#include "PolicySecret_fp.h"
typedef TPM_RC  (PolicySecret_Entry)(
    PolicySecret_In *in,
    PolicySecret_Out *out
);
typedef const struct {
    PolicySecret_Entry    *entry;
    UINT16                inSize;
    UINT16                outSize;
    UINT16                offsetOfTypes;
    UINT16                paramOffsets[6];
    BYTE                  types[10];
} PolicySecret_COMMAND_DESCRIPTOR_t;
PolicySecret_COMMAND_DESCRIPTOR_t _PolicySecretData = {
/* entry  */          &TPM2_PolicySecret,
/* inSize */          (UINT16)(sizeof(PolicySecret_In)),
/* outSize */         (UINT16)(sizeof(PolicySecret_Out)),
/* offsetOfTypes */   offsetof(PolicySecret_COMMAND_DESCRIPTOR_t, types),
/* offsets */         {(UINT16)(offsetof(PolicySecret_In, policySession)),
                       (UINT16)(offsetof(PolicySecret_In, nonceTPM)),
                       (UINT16)(offsetof(PolicySecret_In, cpHashA)),
                       (UINT16)(offsetof(PolicySecret_In, policyRef)),
                       (UINT16)(offsetof(PolicySecret_In, expiration)),
                       (UINT16)(offsetof(PolicySecret_Out, policyTicket))},
/* types */           {TPMI_DH_ENTITY_H_UNMARSHAL,
                       TPMI_SH_POLICY_H_UNMARSHAL,
                       TPM2B_DIGEST_P_UNMARSHAL,
                       TPM2B_DIGEST_P_UNMARSHAL,
                       TPM2B_DIGEST_P_UNMARSHAL,
                       UINT32_P_UNMARSHAL,
                       END_OF_LIST,
                       TPM2B_DIGEST_P_MARSHAL,
                       TPMT_TK_AUTH_P_MARSHAL,
                       END_OF_LIST}
};
#define _PolicySecretDataAddress (&_PolicySecretData)
#else
#define _PolicySecretDataAddress 0
#endif
#if CC_PolicyTicket == YES
#include "PolicyTicket_fp.h"
typedef TPM_RC  (PolicyTicket_Entry)(
    PolicyTicket_In *in
);
typedef const struct {
    PolicyTicket_Entry    *entry;
    UINT16                inSize;
    UINT16                outSize;
    UINT16                offsetOfTypes;
    UINT16                paramOffsets[5];
    BYTE                  types[8];
```

```
2093    } PolicyTicket_COMMAND_DESCRIPTOR_t;
2094    PolicyTicket_COMMAND_DESCRIPTOR_t _PolicyTicketData = {
2095    /* entry  */          &TPM2_PolicyTicket,
2096    /* inSize */          (UINT16)(sizeof(PolicyTicket_In)),
2097    /* outSize */         0,
2098    /* offsetOfTypes */   offsetof(PolicyTicket_COMMAND_DESCRIPTOR_t, types),
2099    /* offsets */         {(UINT16)(offsetof(PolicyTicket_In, timeout)),
2100                           (UINT16)(offsetof(PolicyTicket_In, cpHashA)),
2101                           (UINT16)(offsetof(PolicyTicket_In, policyRef)),
2102                           (UINT16)(offsetof(PolicyTicket_In, authName)),
2103                           (UINT16)(offsetof(PolicyTicket_In, ticket))},
2104    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2105                           TPM2B_DIGEST_P_UNMARSHAL,
2106                           TPM2B_DIGEST_P_UNMARSHAL,
2107                           TPM2B_DIGEST_P_UNMARSHAL,
2108                           TPM2B_NAME_P_UNMARSHAL,
2109                           TPMT_TK_AUTH_P_UNMARSHAL,
2110                           END_OF_LIST,
2111                           END_OF_LIST}
2112    };
2113    #define _PolicyTicketDataAddress (&_PolicyTicketData)
2114    #else
2115    #define _PolicyTicketDataAddress 0
2116    #endif
2117    #if CC_PolicyOR == YES
2118    #include "PolicyOR_fp.h"
2119    typedef TPM_RC  (PolicyOR_Entry)(
2120        PolicyOR_In *in
2121    );
2122    typedef const struct {
2123        PolicyOR_Entry    *entry;
2124        UINT16            inSize;
2125        UINT16            outSize;
2126        UINT16            offsetOfTypes;
2127        UINT16            paramOffsets[1];
2128        BYTE              types[4];
2129    } PolicyOR_COMMAND_DESCRIPTOR_t;
2130    PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
2131    /* entry  */          &TPM2_PolicyOR,
2132    /* inSize */          (UINT16)(sizeof(PolicyOR_In)),
2133    /* outSize */         0,
2134    /* offsetOfTypes */   offsetof(PolicyOR_COMMAND_DESCRIPTOR_t, types),
2135    /* offsets */         {(UINT16)(offsetof(PolicyOR_In, pHashList))},
2136    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2137                           TPML_DIGEST_P_UNMARSHAL,
2138                           END_OF_LIST,
2139                           END_OF_LIST}
2140    };
2141    #define _PolicyORDataAddress (&_PolicyORData)
2142    #else
2143    #define _PolicyORDataAddress 0
2144    #endif
2145    #if CC_PolicyPCR == YES
2146    #include "PolicyPCR_fp.h"
2147    typedef TPM_RC  (PolicyPCR_Entry)(
2148        PolicyPCR_In *in
2149    );
2150    typedef const struct {
2151        PolicyPCR_Entry    *entry;
2152        UINT16             inSize;
2153        UINT16             outSize;
2154        UINT16             offsetOfTypes;
2155        UINT16             paramOffsets[2];
2156        BYTE               types[5];
2157    } PolicyPCR_COMMAND_DESCRIPTOR_t;
2158    PolicyPCR_COMMAND_DESCRIPTOR_t _PolicyPCRData = {
```

```
2159    /* entry  */          &TPM2_PolicyPCR,
2160    /* inSize */          (UINT16)(sizeof(PolicyPCR_In)),
2161    /* outSize */         0,
2162    /* offsetOfTypes */   offsetof(PolicyPCR_COMMAND_DESCRIPTOR_t, types),
2163    /* offsets */         {(UINT16)(offsetof(PolicyPCR_In, pcrDigest)),
2164                           (UINT16)(offsetof(PolicyPCR_In, pcrs))},
2165    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2166                           TPM2B_DIGEST_P_UNMARSHAL,
2167                           TPML_PCR_SELECTION_P_UNMARSHAL,
2168                           END_OF_LIST,
2169                           END_OF_LIST}
2170    };
2171    #define _PolicyPCRDataAddress (&_PolicyPCRData)
2172    #else
2173    #define _PolicyPCRDataAddress 0
2174    #endif
2175    #if CC_PolicyLocality == YES
2176    #include "PolicyLocality_fp.h"
2177    typedef TPM_RC  (PolicyLocality_Entry)(
2178        PolicyLocality_In *in
2179    );
2180    typedef const struct {
2181        PolicyLocality_Entry    *entry;
2182        UINT16                  inSize;
2183        UINT16                  outSize;
2184        UINT16                  offsetOfTypes;
2185        UINT16                  paramOffsets[1];
2186        BYTE                    types[4];
2187    } PolicyLocality_COMMAND_DESCRIPTOR_t;
2188    PolicyLocality_COMMAND_DESCRIPTOR_t _PolicyLocalityData = {
2189    /* entry  */          &TPM2_PolicyLocality,
2190    /* inSize */          (UINT16)(sizeof(PolicyLocality_In)),
2191    /* outSize */         0,
2192    /* offsetOfTypes */   offsetof(PolicyLocality_COMMAND_DESCRIPTOR_t, types),
2193    /* offsets */         {(UINT16)(offsetof(PolicyLocality_In, locality))},
2194    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2195                           UINT8_P_UNMARSHAL,
2196                           END_OF_LIST,
2197                           END_OF_LIST}
2198    };
2199    #define _PolicyLocalityDataAddress (&_PolicyLocalityData)
2200    #else
2201    #define _PolicyLocalityDataAddress 0
2202    #endif
2203    #if CC_PolicyNV == YES
2204    #include "PolicyNV_fp.h"
2205    typedef TPM_RC  (PolicyNV_Entry)(
2206        PolicyNV_In *in
2207    );
2208    typedef const struct {
2209        PolicyNV_Entry    *entry;
2210        UINT16            inSize;
2211        UINT16            outSize;
2212        UINT16            offsetOfTypes;
2213        UINT16            paramOffsets[5];
2214        BYTE              types[8];
2215    } PolicyNV_COMMAND_DESCRIPTOR_t;
2216    PolicyNV_COMMAND_DESCRIPTOR_t _PolicyNVData = {
2217    /* entry  */          &TPM2_PolicyNV,
2218    /* inSize */          (UINT16)(sizeof(PolicyNV_In)),
2219    /* outSize */         0,
2220    /* offsetOfTypes */   offsetof(PolicyNV_COMMAND_DESCRIPTOR_t, types),
2221    /* offsets */         {(UINT16)(offsetof(PolicyNV_In, nvIndex)),
2222                           (UINT16)(offsetof(PolicyNV_In, policySession)),
2223                           (UINT16)(offsetof(PolicyNV_In, operandB)),
2224                           (UINT16)(offsetof(PolicyNV_In, offset)),
```

```
2225                        (UINT16)(offsetof(PolicyNV_In, operation))},
2226    /* types */         {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2227                         TPMI_RH_NV_INDEX_H_UNMARSHAL,
2228                         TPMI_SH_POLICY_H_UNMARSHAL,
2229                         TPM2B_DIGEST_P_UNMARSHAL,
2230                         UINT16_P_UNMARSHAL,
2231                         TPM_EO_P_UNMARSHAL,
2232                         END_OF_LIST,
2233                         END_OF_LIST}
2234    };
2235    #define _PolicyNVDataAddress (&_PolicyNVData)
2236    #else
2237    #define _PolicyNVDataAddress 0
2238    #endif
2239    #if CC_PolicyCounterTimer == YES
2240    #include "PolicyCounterTimer_fp.h"
2241    typedef TPM_RC  (PolicyCounterTimer_Entry)(
2242        PolicyCounterTimer_In *in
2243    );
2244    typedef const struct {
2245        PolicyCounterTimer_Entry    *entry;
2246        UINT16                      inSize;
2247        UINT16                      outSize;
2248        UINT16                      offsetOfTypes;
2249        UINT16                      paramOffsets[3];
2250        BYTE                        types[6];
2251    } PolicyCounterTimer_COMMAND_DESCRIPTOR_t;
2252    PolicyCounterTimer_COMMAND_DESCRIPTOR_t _PolicyCounterTimerData = {
2253    /* entry  */        &TPM2_PolicyCounterTimer,
2254    /* inSize */        (UINT16)(sizeof(PolicyCounterTimer_In)),
2255    /* outSize */       0,
2256    /* offsetOfTypes */ offsetof(PolicyCounterTimer_COMMAND_DESCRIPTOR_t, types),
2257    /* offsets */       {(UINT16)(offsetof(PolicyCounterTimer_In, operandB)),
2258                         (UINT16)(offsetof(PolicyCounterTimer_In, offset)),
2259                         (UINT16)(offsetof(PolicyCounterTimer_In, operation))},
2260    /* types */         {TPMI_SH_POLICY_H_UNMARSHAL,
2261                         TPM2B_DIGEST_P_UNMARSHAL,
2262                         UINT16_P_UNMARSHAL,
2263                         TPM_EO_P_UNMARSHAL,
2264                         END_OF_LIST,
2265                         END_OF_LIST}
2266    };
2267    #define _PolicyCounterTimerDataAddress (&_PolicyCounterTimerData)
2268    #else
2269    #define _PolicyCounterTimerDataAddress 0
2270    #endif
2271    #if CC_PolicyCommandCode == YES
2272    #include "PolicyCommandCode_fp.h"
2273    typedef TPM_RC  (PolicyCommandCode_Entry)(
2274        PolicyCommandCode_In *in
2275    );
2276    typedef const struct {
2277        PolicyCommandCode_Entry    *entry;
2278        UINT16                     inSize;
2279        UINT16                     outSize;
2280        UINT16                     offsetOfTypes;
2281        UINT16                     paramOffsets[1];
2282        BYTE                       types[4];
2283    } PolicyCommandCode_COMMAND_DESCRIPTOR_t;
2284    PolicyCommandCode_COMMAND_DESCRIPTOR_t _PolicyCommandCodeData = {
2285    /* entry  */        &TPM2_PolicyCommandCode,
2286    /* inSize */        (UINT16)(sizeof(PolicyCommandCode_In)),
2287    /* outSize */       0,
2288    /* offsetOfTypes */ offsetof(PolicyCommandCode_COMMAND_DESCRIPTOR_t, types),
2289    /* offsets */       {(UINT16)(offsetof(PolicyCommandCode_In, code))},
2290    /* types */         {TPMI_SH_POLICY_H_UNMARSHAL,
```

```
                        UINT32_P_UNMARSHAL,
                        END_OF_LIST,
                        END_OF_LIST}
};
#define _PolicyCommandCodeDataAddress (&_PolicyCommandCodeData)
#else
#define _PolicyCommandCodeDataAddress 0
#endif
#if CC_PolicyPhysicalPresence == YES
#include "PolicyPhysicalPresence_fp.h"
typedef TPM_RC  (PolicyPhysicalPresence_Entry)(
    PolicyPhysicalPresence_In *in
);
typedef const struct {
    PolicyPhysicalPresence_Entry    *entry;
    UINT16                          inSize;
    UINT16                          outSize;
    UINT16                          offsetOfTypes;
    BYTE                            types[3];
} PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t;
PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t _PolicyPhysicalPresenceData = {
/* entry  */          &TPM2_PolicyPhysicalPresence,
/* inSize */          (UINT16)(sizeof(PolicyPhysicalPresence_In)),
/* outSize */         0,
/* offsetOfTypes */   offsetof(PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t, types),
/* offsets */         // No parameter offsets
/* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
                       END_OF_LIST,
                       END_OF_LIST}
};
#define _PolicyPhysicalPresenceDataAddress (&_PolicyPhysicalPresenceData)
#else
#define _PolicyPhysicalPresenceDataAddress 0
#endif
#if CC_PolicyCpHash == YES
#include "PolicyCpHash_fp.h"
typedef TPM_RC  (PolicyCpHash_Entry)(
    PolicyCpHash_In *in
);
typedef const struct {
    PolicyCpHash_Entry    *entry;
    UINT16                inSize;
    UINT16                outSize;
    UINT16                offsetOfTypes;
    UINT16                paramOffsets[1];
    BYTE                  types[4];
} PolicyCpHash_COMMAND_DESCRIPTOR_t;
PolicyCpHash_COMMAND_DESCRIPTOR_t _PolicyCpHashData = {
/* entry  */          &TPM2_PolicyCpHash,
/* inSize */          (UINT16)(sizeof(PolicyCpHash_In)),
/* outSize */         0,
/* offsetOfTypes */   offsetof(PolicyCpHash_COMMAND_DESCRIPTOR_t, types),
/* offsets */         {(UINT16)(offsetof(PolicyCpHash_In, cpHashA))},
/* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
                       TPM2B_DIGEST_P_UNMARSHAL,
                       END_OF_LIST,
                       END_OF_LIST}
};
#define _PolicyCpHashDataAddress (&_PolicyCpHashData)
#else
#define _PolicyCpHashDataAddress 0
#endif
#if CC_PolicyNameHash == YES
#include "PolicyNameHash_fp.h"
typedef TPM_RC  (PolicyNameHash_Entry)(
    PolicyNameHash_In *in
```

```
2357    );
2358    typedef const struct {
2359        PolicyNameHash_Entry    *entry;
2360        UINT16                  inSize;
2361        UINT16                  outSize;
2362        UINT16                  offsetOfTypes;
2363        UINT16                  paramOffsets[1];
2364        BYTE                    types[4];
2365    } PolicyNameHash_COMMAND_DESCRIPTOR_t;
2366    PolicyNameHash_COMMAND_DESCRIPTOR_t _PolicyNameHashData = {
2367    /* entry  */          &TPM2_PolicyNameHash,
2368    /* inSize */          (UINT16)(sizeof(PolicyNameHash_In)),
2369    /* outSize */         0,
2370    /* offsetOfTypes */   offsetof(PolicyNameHash_COMMAND_DESCRIPTOR_t, types),
2371    /* offsets */         {(UINT16)(offsetof(PolicyNameHash_In, nameHash))},
2372    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2373                           TPM2B_DIGEST_P_UNMARSHAL,
2374                           END_OF_LIST,
2375                           END_OF_LIST}
2376    };
2377    #define _PolicyNameHashDataAddress (&_PolicyNameHashData)
2378    #else
2379    #define _PolicyNameHashDataAddress 0
2380    #endif
2381    #if CC_PolicyDuplicationSelect == YES
2382    #include "PolicyDuplicationSelect_fp.h"
2383    typedef TPM_RC  (PolicyDuplicationSelect_Entry)(
2384        PolicyDuplicationSelect_In *in
2385    );
2386    typedef const struct {
2387        PolicyDuplicationSelect_Entry    *entry;
2388        UINT16                           inSize;
2389        UINT16                           outSize;
2390        UINT16                           offsetOfTypes;
2391        UINT16                           paramOffsets[3];
2392        BYTE                             types[6];
2393    } PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t;
2394    PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t _PolicyDuplicationSelectData = {
2395    /* entry  */          &TPM2_PolicyDuplicationSelect,
2396    /* inSize */          (UINT16)(sizeof(PolicyDuplicationSelect_In)),
2397    /* outSize */         0,
2398    /* offsetOfTypes */   offsetof(PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t, types),
2399    /* offsets */         {(UINT16)(offsetof(PolicyDuplicationSelect_In, objectName)),
2400                           (UINT16)(offsetof(PolicyDuplicationSelect_In, newParentName)),
2401                           (UINT16)(offsetof(PolicyDuplicationSelect_In, includeObject))},
2402    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2403                           TPM2B_NAME_P_UNMARSHAL,
2404                           TPM2B_NAME_P_UNMARSHAL,
2405                           TPMI_YES_NO_P_UNMARSHAL,
2406                           END_OF_LIST,
2407                           END_OF_LIST}
2408    };
2409    #define _PolicyDuplicationSelectDataAddress (&_PolicyDuplicationSelectData)
2410    #else
2411    #define _PolicyDuplicationSelectDataAddress 0
2412    #endif
2413    #if CC_PolicyAuthorize == YES
2414    #include "PolicyAuthorize_fp.h"
2415    typedef TPM_RC  (PolicyAuthorize_Entry)(
2416        PolicyAuthorize_In *in
2417    );
2418    typedef const struct {
2419        PolicyAuthorize_Entry    *entry;
2420        UINT16                   inSize;
2421        UINT16                   outSize;
2422        UINT16                   offsetOfTypes;
```

```
2423        UINT16                  paramOffsets[4];
2424        BYTE                    types[7];
2425    } PolicyAuthorize_COMMAND_DESCRIPTOR_t;
2426    PolicyAuthorize_COMMAND_DESCRIPTOR_t _PolicyAuthorizeData = {
2427    /* entry  */            &TPM2_PolicyAuthorize,
2428    /* inSize */            (UINT16)(sizeof(PolicyAuthorize_In)),
2429    /* outSize */           0,
2430    /* offsetOfTypes */     offsetof(PolicyAuthorize_COMMAND_DESCRIPTOR_t, types),
2431    /* offsets */           {(UINT16)(offsetof(PolicyAuthorize_In, approvedPolicy)),
2432                             (UINT16)(offsetof(PolicyAuthorize_In, policyRef)),
2433                             (UINT16)(offsetof(PolicyAuthorize_In, keySign)),
2434                             (UINT16)(offsetof(PolicyAuthorize_In, checkTicket))},
2435    /* types */             {TPMI_SH_POLICY_H_UNMARSHAL,
2436                             TPM2B_DIGEST_P_UNMARSHAL,
2437                             TPM2B_DIGEST_P_UNMARSHAL,
2438                             TPM2B_NAME_P_UNMARSHAL,
2439                             TPMT_TK_VERIFIED_P_UNMARSHAL,
2440                             END_OF_LIST,
2441                             END_OF_LIST}
2442    };
2443    #define _PolicyAuthorizeDataAddress (&_PolicyAuthorizeData)
2444    #else
2445    #define _PolicyAuthorizeDataAddress 0
2446    #endif
2447    #if CC_PolicyAuthValue == YES
2448    #include "PolicyAuthValue_fp.h"
2449    typedef TPM_RC  (PolicyAuthValue_Entry)(
2450        PolicyAuthValue_In *in
2451    );
2452    typedef const struct {
2453        PolicyAuthValue_Entry    *entry;
2454        UINT16                   inSize;
2455        UINT16                   outSize;
2456        UINT16                   offsetOfTypes;
2457        BYTE                     types[3];
2458    } PolicyAuthValue_COMMAND_DESCRIPTOR_t;
2459    PolicyAuthValue_COMMAND_DESCRIPTOR_t _PolicyAuthValueData = {
2460    /* entry  */            &TPM2_PolicyAuthValue,
2461    /* inSize */            (UINT16)(sizeof(PolicyAuthValue_In)),
2462    /* outSize */           0,
2463    /* offsetOfTypes */     offsetof(PolicyAuthValue_COMMAND_DESCRIPTOR_t, types),
2464    /* offsets */           // No parameter offsets
2465    /* types */             {TPMI_SH_POLICY_H_UNMARSHAL,
2466                             END_OF_LIST,
2467                             END_OF_LIST}
2468    };
2469    #define _PolicyAuthValueDataAddress (&_PolicyAuthValueData)
2470    #else
2471    #define _PolicyAuthValueDataAddress 0
2472    #endif
2473    #if CC_PolicyPassword == YES
2474    #include "PolicyPassword_fp.h"
2475    typedef TPM_RC  (PolicyPassword_Entry)(
2476        PolicyPassword_In *in
2477    );
2478    typedef const struct {
2479        PolicyPassword_Entry    *entry;
2480        UINT16                  inSize;
2481        UINT16                  outSize;
2482        UINT16                  offsetOfTypes;
2483        BYTE                    types[3];
2484    } PolicyPassword_COMMAND_DESCRIPTOR_t;
2485    PolicyPassword_COMMAND_DESCRIPTOR_t _PolicyPasswordData = {
2486    /* entry  */            &TPM2_PolicyPassword,
2487    /* inSize */            (UINT16)(sizeof(PolicyPassword_In)),
2488    /* outSize */           0,
```

```
2489    /* offsetOfTypes */   offsetof(PolicyPassword_COMMAND_DESCRIPTOR_t, types),
2490    /* offsets */         // No parameter offsets
2491    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2492                           END_OF_LIST,
2493                           END_OF_LIST}
2494    };
2495    #define _PolicyPasswordDataAddress (&_PolicyPasswordData)
2496    #else
2497    #define _PolicyPasswordDataAddress 0
2498    #endif
2499    #if CC_PolicyGetDigest == YES
2500    #include "PolicyGetDigest_fp.h"
2501    typedef TPM_RC  (PolicyGetDigest_Entry)(
2502        PolicyGetDigest_In *in,
2503        PolicyGetDigest_Out *out
2504    );
2505    typedef const struct {
2506        PolicyGetDigest_Entry    *entry;
2507        UINT16                   inSize;
2508        UINT16                   outSize;
2509        UINT16                   offsetOfTypes;
2510        BYTE                     types[4];
2511    } PolicyGetDigest_COMMAND_DESCRIPTOR_t;
2512    PolicyGetDigest_COMMAND_DESCRIPTOR_t _PolicyGetDigestData = {
2513    /* entry  */          &TPM2_PolicyGetDigest,
2514    /* inSize */          (UINT16)(sizeof(PolicyGetDigest_In)),
2515    /* outSize */         (UINT16)(sizeof(PolicyGetDigest_Out)),
2516    /* offsetOfTypes */   offsetof(PolicyGetDigest_COMMAND_DESCRIPTOR_t, types),
2517    /* offsets */         // No parameter offsets
2518    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2519                           END_OF_LIST,
2520                           TPM2B_DIGEST_P_MARSHAL,
2521                           END_OF_LIST}
2522    };
2523    #define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
2524    #else
2525    #define _PolicyGetDigestDataAddress 0
2526    #endif
2527    #if CC_PolicyNvWritten == YES
2528    #include "PolicyNvWritten_fp.h"
2529    typedef TPM_RC  (PolicyNvWritten_Entry)(
2530        PolicyNvWritten_In *in
2531    );
2532    typedef const struct {
2533        PolicyNvWritten_Entry    *entry;
2534        UINT16                   inSize;
2535        UINT16                   outSize;
2536        UINT16                   offsetOfTypes;
2537        UINT16                   paramOffsets[1];
2538        BYTE                     types[4];
2539    } PolicyNvWritten_COMMAND_DESCRIPTOR_t;
2540    PolicyNvWritten_COMMAND_DESCRIPTOR_t _PolicyNvWrittenData = {
2541    /* entry  */          &TPM2_PolicyNvWritten,
2542    /* inSize */          (UINT16)(sizeof(PolicyNvWritten_In)),
2543    /* outSize */         0,
2544    /* offsetOfTypes */   offsetof(PolicyNvWritten_COMMAND_DESCRIPTOR_t, types),
2545    /* offsets */         {(UINT16)(offsetof(PolicyNvWritten_In, writtenSet))},
2546    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2547                           TPMI_YES_NO_P_UNMARSHAL,
2548                           END_OF_LIST,
2549                           END_OF_LIST}
2550    };
2551    #define _PolicyNvWrittenDataAddress (&_PolicyNvWrittenData)
2552    #else
2553    #define _PolicyNvWrittenDataAddress 0
2554    #endif
```

```
2555    #if CC_PolicyTemplate == YES
2556    #include "PolicyTemplate_fp.h"
2557    typedef TPM_RC  (PolicyTemplate_Entry)(
2558        PolicyTemplate_In *in
2559    );
2560    typedef const struct {
2561        PolicyTemplate_Entry    *entry;
2562        UINT16                  inSize;
2563        UINT16                  outSize;
2564        UINT16                  offsetOfTypes;
2565        UINT16                  paramOffsets[1];
2566        BYTE                    types[4];
2567    } PolicyTemplate_COMMAND_DESCRIPTOR_t;
2568    PolicyTemplate_COMMAND_DESCRIPTOR_t _PolicyTemplateData = {
2569    /* entry  */          &TPM2_PolicyTemplate,
2570    /* inSize */          (UINT16)(sizeof(PolicyTemplate_In)),
2571    /* outSize */         0,
2572    /* offsetOfTypes */   offsetof(PolicyTemplate_COMMAND_DESCRIPTOR_t, types),
2573    /* offsets */         {(UINT16)(offsetof(PolicyTemplate_In, templateHash))},
2574    /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2575                           TPM2B_DIGEST_P_UNMARSHAL,
2576                           END_OF_LIST,
2577                           END_OF_LIST}
2578    };
2579    #define _PolicyTemplateDataAddress (&_PolicyTemplateData)
2580    #else
2581    #define _PolicyTemplateDataAddress 0
2582    #endif
2583    #if CC_PolicyAuthorizeNV == YES
2584    #include "PolicyAuthorizeNV_fp.h"
2585    typedef TPM_RC  (PolicyAuthorizeNV_Entry)(
2586        PolicyAuthorizeNV_In *in
2587    );
2588    typedef const struct {
2589        PolicyAuthorizeNV_Entry    *entry;
2590        UINT16                     inSize;
2591        UINT16                     outSize;
2592        UINT16                     offsetOfTypes;
2593        UINT16                     paramOffsets[2];
2594        BYTE                       types[5];
2595    } PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t;
2596    PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t _PolicyAuthorizeNVData = {
2597    /* entry  */          &TPM2_PolicyAuthorizeNV,
2598    /* inSize */          (UINT16)(sizeof(PolicyAuthorizeNV_In)),
2599    /* outSize */         0,
2600    /* offsetOfTypes */   offsetof(PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t, types),
2601    /* offsets */         {(UINT16)(offsetof(PolicyAuthorizeNV_In, nvIndex)),
2602                           (UINT16)(offsetof(PolicyAuthorizeNV_In, policySession))},
2603    /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2604                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
2605                           TPMI_SH_POLICY_H_UNMARSHAL,
2606                           END_OF_LIST,
2607                           END_OF_LIST}
2608    };
2609    #define _PolicyAuthorizeNVDataAddress (&_PolicyAuthorizeNVData)
2610    #else
2611    #define _PolicyAuthorizeNVDataAddress 0
2612    #endif
2613    #if CC_CreatePrimary == YES
2614    #include "CreatePrimary_fp.h"
2615    typedef TPM_RC  (CreatePrimary_Entry)(
2616        CreatePrimary_In *in,
2617        CreatePrimary_Out *out
2618    );
2619    typedef const struct {
2620        CreatePrimary_Entry    *entry;
```

```
2621        UINT16                  inSize;
2622        UINT16                  outSize;
2623        UINT16                  offsetOfTypes;
2624        UINT16                  paramOffsets[9];
2625        BYTE                    types[13];
2626    } CreatePrimary_COMMAND_DESCRIPTOR_t;
2627    CreatePrimary_COMMAND_DESCRIPTOR_t _CreatePrimaryData = {
2628    /* entry  */            &TPM2_CreatePrimary,
2629    /* inSize */            (UINT16)(sizeof(CreatePrimary_In)),
2630    /* outSize */           (UINT16)(sizeof(CreatePrimary_Out)),
2631    /* offsetOfTypes */     offsetof(CreatePrimary_COMMAND_DESCRIPTOR_t, types),
2632    /* offsets */           {(UINT16)(offsetof(CreatePrimary_In, inSensitive)),
2633                             (UINT16)(offsetof(CreatePrimary_In, inPublic)),
2634                             (UINT16)(offsetof(CreatePrimary_In, outsideInfo)),
2635                             (UINT16)(offsetof(CreatePrimary_In, creationPCR)),
2636                             (UINT16)(offsetof(CreatePrimary_Out, outPublic)),
2637                             (UINT16)(offsetof(CreatePrimary_Out, creationData)),
2638                             (UINT16)(offsetof(CreatePrimary_Out, creationHash)),
2639                             (UINT16)(offsetof(CreatePrimary_Out, creationTicket)),
2640                             (UINT16)(offsetof(CreatePrimary_Out, name))},
2641    /* types */             {TPMI_RH_HIERARCHY_H_UNMARSHAL + ADD_FLAG,
2642                             TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
2643                             TPM2B_PUBLIC_P_UNMARSHAL,
2644                             TPM2B_DATA_P_UNMARSHAL,
2645                             TPML_PCR_SELECTION_P_UNMARSHAL,
2646                             END_OF_LIST,
2647                             UINT32_H_MARSHAL,
2648                             TPM2B_PUBLIC_P_MARSHAL,
2649                             TPM2B_CREATION_DATA_P_MARSHAL,
2650                             TPM2B_DIGEST_P_MARSHAL,
2651                             TPMT_TK_CREATION_P_MARSHAL,
2652                             TPM2B_NAME_P_MARSHAL,
2653                             END_OF_LIST}
2654    };
2655    #define _CreatePrimaryDataAddress (&_CreatePrimaryData)
2656    #else
2657    #define _CreatePrimaryDataAddress 0
2658    #endif
2659    #if CC_HierarchyControl == YES
2660    #include "HierarchyControl_fp.h"
2661    typedef TPM_RC  (HierarchyControl_Entry)(
2662        HierarchyControl_In *in
2663    );
2664    typedef const struct {
2665        HierarchyControl_Entry    *entry;
2666        UINT16                    inSize;
2667        UINT16                    outSize;
2668        UINT16                    offsetOfTypes;
2669        UINT16                    paramOffsets[2];
2670        BYTE                      types[5];
2671    } HierarchyControl_COMMAND_DESCRIPTOR_t;
2672    HierarchyControl_COMMAND_DESCRIPTOR_t _HierarchyControlData = {
2673    /* entry  */            &TPM2_HierarchyControl,
2674    /* inSize */            (UINT16)(sizeof(HierarchyControl_In)),
2675    /* outSize */           0,
2676    /* offsetOfTypes */     offsetof(HierarchyControl_COMMAND_DESCRIPTOR_t, types),
2677    /* offsets */           {(UINT16)(offsetof(HierarchyControl_In, enable)),
2678                             (UINT16)(offsetof(HierarchyControl_In, state))},
2679    /* types */             {TPMI_RH_HIERARCHY_H_UNMARSHAL,
2680                             TPMI_RH_ENABLES_P_UNMARSHAL,
2681                             TPMI_YES_NO_P_UNMARSHAL,
2682                             END_OF_LIST,
2683                             END_OF_LIST}
2684    };
2685    #define _HierarchyControlDataAddress (&_HierarchyControlData)
2686    #else
```



```
2687    #define _HierarchyControlDataAddress 0
2688    #endif
2689    #if CC_SetPrimaryPolicy == YES
2690    #include "SetPrimaryPolicy_fp.h"
2691    typedef TPM_RC  (SetPrimaryPolicy_Entry)(
2692        SetPrimaryPolicy_In *in
2693    );
2694    typedef const struct {
2695        SetPrimaryPolicy_Entry    *entry;
2696        UINT16                    inSize;
2697        UINT16                    outSize;
2698        UINT16                    offsetOfTypes;
2699        UINT16                    paramOffsets[2];
2700        BYTE                      types[5];
2701    } SetPrimaryPolicy_COMMAND_DESCRIPTOR_t;
2702    SetPrimaryPolicy_COMMAND_DESCRIPTOR_t _SetPrimaryPolicyData = {
2703    /* entry  */          &TPM2_SetPrimaryPolicy,
2704    /* inSize */          (UINT16)(sizeof(SetPrimaryPolicy_In)),
2705    /* outSize */         0,
2706    /* offsetOfTypes */   offsetof(SetPrimaryPolicy_COMMAND_DESCRIPTOR_t, types),
2707    /* offsets */         {(UINT16)(offsetof(SetPrimaryPolicy_In, authPolicy)),
2708                           (UINT16)(offsetof(SetPrimaryPolicy_In, hashAlg))},
2709    /* types */           {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
2710                           TPM2B_DIGEST_P_UNMARSHAL,
2711                           TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2712                           END_OF_LIST,
2713                           END_OF_LIST}
2714    };
2715    #define _SetPrimaryPolicyDataAddress (&_SetPrimaryPolicyData)
2716    #else
2717    #define _SetPrimaryPolicyDataAddress 0
2718    #endif
2719    #if CC_ChangePPS == YES
2720    #include "ChangePPS_fp.h"
2721    typedef TPM_RC  (ChangePPS_Entry)(
2722        ChangePPS_In *in
2723    );
2724    typedef const struct {
2725        ChangePPS_Entry    *entry;
2726        UINT16             inSize;
2727        UINT16             outSize;
2728        UINT16             offsetOfTypes;
2729        BYTE               types[3];
2730    } ChangePPS_COMMAND_DESCRIPTOR_t;
2731    ChangePPS_COMMAND_DESCRIPTOR_t _ChangePPSData = {
2732    /* entry  */          &TPM2_ChangePPS,
2733    /* inSize */          (UINT16)(sizeof(ChangePPS_In)),
2734    /* outSize */         0,
2735    /* offsetOfTypes */   offsetof(ChangePPS_COMMAND_DESCRIPTOR_t, types),
2736    /* offsets */         // No parameter offsets
2737    /* types */           {TPMI_RH_PLATFORM_H_UNMARSHAL,
2738                           END_OF_LIST,
2739                           END_OF_LIST}
2740    };
2741    #define _ChangePPSDataAddress (&_ChangePPSData)
2742    #else
2743    #define _ChangePPSDataAddress 0
2744    #endif
2745    #if CC_ChangeEPS == YES
2746    #include "ChangeEPS_fp.h"
2747    typedef TPM_RC  (ChangeEPS_Entry)(
2748        ChangeEPS_In *in
2749    );
2750    typedef const struct {
2751        ChangeEPS_Entry    *entry;
2752        UINT16             inSize;
```

```
2753        UINT16              outSize;
2754        UINT16              offsetOfTypes;
2755        BYTE                types[3];
2756    } ChangeEPS_COMMAND_DESCRIPTOR_t;
2757    ChangeEPS_COMMAND_DESCRIPTOR_t _ChangeEPSData = {
2758    /* entry  */           &TPM2_ChangeEPS,
2759    /* inSize */           (UINT16)(sizeof(ChangeEPS_In)),
2760    /* outSize */          0,
2761    /* offsetOfTypes */    offsetof(ChangeEPS_COMMAND_DESCRIPTOR_t, types),
2762    /* offsets */          // No parameter offsets
2763    /* types */            {TPMI_RH_PLATFORM_H_UNMARSHAL,
2764                            END_OF_LIST,
2765                            END_OF_LIST}
2766    };
2767    #define _ChangeEPSDataAddress (&_ChangeEPSData)
2768    #else
2769    #define _ChangeEPSDataAddress 0
2770    #endif
2771    #if CC_Clear == YES
2772    #include "Clear_fp.h"
2773    typedef TPM_RC  (Clear_Entry)(
2774        Clear_In *in
2775    );
2776    typedef const struct {
2777        Clear_Entry    *entry;
2778        UINT16         inSize;
2779        UINT16         outSize;
2780        UINT16         offsetOfTypes;
2781        BYTE           types[3];
2782    } Clear_COMMAND_DESCRIPTOR_t;
2783    Clear_COMMAND_DESCRIPTOR_t _ClearData = {
2784    /* entry  */           &TPM2_Clear,
2785    /* inSize */           (UINT16)(sizeof(Clear_In)),
2786    /* outSize */          0,
2787    /* offsetOfTypes */    offsetof(Clear_COMMAND_DESCRIPTOR_t, types),
2788    /* offsets */          // No parameter offsets
2789    /* types */            {TPMI_RH_CLEAR_H_UNMARSHAL,
2790                            END_OF_LIST,
2791                            END_OF_LIST}
2792    };
2793    #define _ClearDataAddress (&_ClearData)
2794    #else
2795    #define _ClearDataAddress 0
2796    #endif
2797    #if CC_ClearControl == YES
2798    #include "ClearControl_fp.h"
2799    typedef TPM_RC  (ClearControl_Entry)(
2800        ClearControl_In *in
2801    );
2802    typedef const struct {
2803        ClearControl_Entry    *entry;
2804        UINT16                inSize;
2805        UINT16                outSize;
2806        UINT16                offsetOfTypes;
2807        UINT16                paramOffsets[1];
2808        BYTE                  types[4];
2809    } ClearControl_COMMAND_DESCRIPTOR_t;
2810    ClearControl_COMMAND_DESCRIPTOR_t _ClearControlData = {
2811    /* entry  */           &TPM2_ClearControl,
2812    /* inSize */           (UINT16)(sizeof(ClearControl_In)),
2813    /* outSize */          0,
2814    /* offsetOfTypes */    offsetof(ClearControl_COMMAND_DESCRIPTOR_t, types),
2815    /* offsets */          {(UINT16)(offsetof(ClearControl_In, disable))},
2816    /* types */            {TPMI_RH_CLEAR_H_UNMARSHAL,
2817                            TPMI_YES_NO_P_UNMARSHAL,
2818                            END_OF_LIST,
```

```
2819                        END_OF_LIST}
2820    };
2821    #define _ClearControlDataAddress (&_ClearControlData)
2822    #else
2823    #define _ClearControlDataAddress 0
2824    #endif
2825    #if CC_HierarchyChangeAuth == YES
2826    #include "HierarchyChangeAuth_fp.h"
2827    typedef TPM_RC  (HierarchyChangeAuth_Entry)(
2828        HierarchyChangeAuth_In *in
2829    );
2830    typedef const struct {
2831        HierarchyChangeAuth_Entry    *entry;
2832        UINT16                       inSize;
2833        UINT16                       outSize;
2834        UINT16                       offsetOfTypes;
2835        UINT16                       paramOffsets[1];
2836        BYTE                         types[4];
2837    } HierarchyChangeAuth_COMMAND_DESCRIPTOR_t;
2838    HierarchyChangeAuth_COMMAND_DESCRIPTOR_t _HierarchyChangeAuthData = {
2839    /* entry  */          &TPM2_HierarchyChangeAuth,
2840    /* inSize */          (UINT16)(sizeof(HierarchyChangeAuth_In)),
2841    /* outSize */         0,
2842    /* offsetOfTypes */   offsetof(HierarchyChangeAuth_COMMAND_DESCRIPTOR_t, types),
2843    /* offsets */         {(UINT16)(offsetof(HierarchyChangeAuth_In, newAuth))},
2844    /* types */           {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
2845                           TPM2B_DIGEST_P_UNMARSHAL,
2846                           END_OF_LIST,
2847                           END_OF_LIST}
2848    };
2849    #define _HierarchyChangeAuthDataAddress (&_HierarchyChangeAuthData)
2850    #else
2851    #define _HierarchyChangeAuthDataAddress 0
2852    #endif
2853    #if CC_DictionaryAttackLockReset == YES
2854    #include "DictionaryAttackLockReset_fp.h"
2855    typedef TPM_RC  (DictionaryAttackLockReset_Entry)(
2856        DictionaryAttackLockReset_In *in
2857    );
2858    typedef const struct {
2859        DictionaryAttackLockReset_Entry    *entry;
2860        UINT16                             inSize;
2861        UINT16                             outSize;
2862        UINT16                             offsetOfTypes;
2863        BYTE                               types[3];
2864    } DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t;
2865    DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t _DictionaryAttackLockResetData = {
2866    /* entry  */          &TPM2_DictionaryAttackLockReset,
2867    /* inSize */          (UINT16)(sizeof(DictionaryAttackLockReset_In)),
2868    /* outSize */         0,
2869    /* offsetOfTypes */   offsetof(DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t, types),
2870    /* offsets */         // No parameter offsets
2871    /* types */           {TPMI_RH_LOCKOUT_H_UNMARSHAL,
2872                           END_OF_LIST,
2873                           END_OF_LIST}
2874    };
2875    #define _DictionaryAttackLockResetDataAddress (&_DictionaryAttackLockResetData)
2876    #else
2877    #define _DictionaryAttackLockResetDataAddress 0
2878    #endif
2879    #if CC_DictionaryAttackParameters == YES
2880    #include "DictionaryAttackParameters_fp.h"
2881    typedef TPM_RC  (DictionaryAttackParameters_Entry)(
2882        DictionaryAttackParameters_In *in
2883    );
2884    typedef const struct {
```

```
2885        DictionaryAttackParameters_Entry    *entry;
2886        UINT16                              inSize;
2887        UINT16                              outSize;
2888        UINT16                              offsetOfTypes;
2889        UINT16                              paramOffsets[3];
2890        BYTE                                types[6];
2891    } DictionaryAttackParameters_COMMAND_DESCRIPTOR_t;
2892    DictionaryAttackParameters_COMMAND_DESCRIPTOR_t _DictionaryAttackParametersData = {
2893    /* entry  */          &TPM2_DictionaryAttackParameters,
2894    /* inSize */          (UINT16)(sizeof(DictionaryAttackParameters_In)),
2895    /* outSize */         0,
2896    /* offsetOfTypes */   offsetof(DictionaryAttackParameters_COMMAND_DESCRIPTOR_t,
        types),
2897    /* offsets */         {(UINT16)(offsetof(DictionaryAttackParameters_In, newMaxTries)),
2898                           (UINT16)(offsetof(DictionaryAttackParameters_In,
        newRecoveryTime)),
2899                           (UINT16)(offsetof(DictionaryAttackParameters_In,
        lockoutRecovery))},
2900    /* types */           {TPMI_RH_LOCKOUT_H_UNMARSHAL,
2901                           UINT32_P_UNMARSHAL,
2902                           UINT32_P_UNMARSHAL,
2903                           UINT32_P_UNMARSHAL,
2904                           END_OF_LIST,
2905                           END_OF_LIST}
2906    };
2907    #define _DictionaryAttackParametersDataAddress (&_DictionaryAttackParametersData)
2908    #else
2909    #define _DictionaryAttackParametersDataAddress 0
2910    #endif
2911    #if CC_PP_Commands == YES
2912    #include "PP_Commands_fp.h"
2913    typedef TPM_RC  (PP_Commands_Entry)(
2914        PP_Commands_In *in
2915    );
2916    typedef const struct {
2917        PP_Commands_Entry    *entry;
2918        UINT16               inSize;
2919        UINT16               outSize;
2920        UINT16               offsetOfTypes;
2921        UINT16               paramOffsets[2];
2922        BYTE                 types[5];
2923    } PP_Commands_COMMAND_DESCRIPTOR_t;
2924    PP_Commands_COMMAND_DESCRIPTOR_t _PP_CommandsData = {
2925    /* entry  */          &TPM2_PP_Commands,
2926    /* inSize */          (UINT16)(sizeof(PP_Commands_In)),
2927    /* outSize */         0,
2928    /* offsetOfTypes */   offsetof(PP_Commands_COMMAND_DESCRIPTOR_t, types),
2929    /* offsets */         {(UINT16)(offsetof(PP_Commands_In, setList)),
2930                           (UINT16)(offsetof(PP_Commands_In, clearList))},
2931    /* types */           {TPMI_RH_PLATFORM_H_UNMARSHAL,
2932                           TPML_CC_P_UNMARSHAL,
2933                           TPML_CC_P_UNMARSHAL,
2934                           END_OF_LIST,
2935                           END_OF_LIST}
2936    };
2937    #define _PP_CommandsDataAddress (&_PP_CommandsData)
2938    #else
2939    #define _PP_CommandsDataAddress 0
2940    #endif
2941    #if CC_SetAlgorithmSet == YES
2942    #include "SetAlgorithmSet_fp.h"
2943    typedef TPM_RC  (SetAlgorithmSet_Entry)(
2944        SetAlgorithmSet_In *in
2945    );
2946    typedef const struct {
2947        SetAlgorithmSet_Entry    *entry;
```

```
2948        UINT16                  inSize;
2949        UINT16                  outSize;
2950        UINT16                  offsetOfTypes;
2951        UINT16                  paramOffsets[1];
2952        BYTE                    types[4];
2953    } SetAlgorithmSet_COMMAND_DESCRIPTOR_t;
2954    SetAlgorithmSet_COMMAND_DESCRIPTOR_t _SetAlgorithmSetData = {
2955    /* entry  */          &TPM2_SetAlgorithmSet,
2956    /* inSize */          (UINT16)(sizeof(SetAlgorithmSet_In)),
2957    /* outSize */         0,
2958    /* offsetOfTypes */   offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
2959    /* offsets */         {(UINT16)(offsetof(SetAlgorithmSet_In, algorithmSet))},
2960    /* types */           {TPMI_RH_PLATFORM_H_UNMARSHAL,
2961                           UINT32_P_UNMARSHAL,
2962                           END_OF_LIST,
2963                           END_OF_LIST}
2964    };
2965    #define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
2966    #else
2967    #define _SetAlgorithmSetDataAddress 0
2968    #endif
2969    #if CC_FieldUpgradeStart == YES
2970    #include "FieldUpgradeStart_fp.h"
2971    typedef TPM_RC  (FieldUpgradeStart_Entry)(
2972        FieldUpgradeStart_In *in
2973    );
2974    typedef const struct {
2975        FieldUpgradeStart_Entry    *entry;
2976        UINT16                     inSize;
2977        UINT16                     outSize;
2978        UINT16                     offsetOfTypes;
2979        UINT16                     paramOffsets[3];
2980        BYTE                       types[6];
2981    } FieldUpgradeStart_COMMAND_DESCRIPTOR_t;
2982    FieldUpgradeStart_COMMAND_DESCRIPTOR_t _FieldUpgradeStartData = {
2983    /* entry  */          &TPM2_FieldUpgradeStart,
2984    /* inSize */          (UINT16)(sizeof(FieldUpgradeStart_In)),
2985    /* outSize */         0,
2986    /* offsetOfTypes */   offsetof(FieldUpgradeStart_COMMAND_DESCRIPTOR_t, types),
2987    /* offsets */         {(UINT16)(offsetof(FieldUpgradeStart_In, keyHandle)),
2988                           (UINT16)(offsetof(FieldUpgradeStart_In, fuDigest)),
2989                           (UINT16)(offsetof(FieldUpgradeStart_In, manifestSignature))},
2990    /* types */           {TPMI_RH_PLATFORM_H_UNMARSHAL,
2991                           TPMI_DH_OBJECT_H_UNMARSHAL,
2992                           TPM2B_DIGEST_P_UNMARSHAL,
2993                           TPMT_SIGNATURE_P_UNMARSHAL,
2994                           END_OF_LIST,
2995                           END_OF_LIST}
2996    };
2997    #define _FieldUpgradeStartDataAddress (&_FieldUpgradeStartData)
2998    #else
2999    #define _FieldUpgradeStartDataAddress 0
3000    #endif
3001    #if CC_FieldUpgradeData == YES
3002    #include "FieldUpgradeData_fp.h"
3003    typedef TPM_RC  (FieldUpgradeData_Entry)(
3004        FieldUpgradeData_In *in,
3005        FieldUpgradeData_Out *out
3006    );
3007    typedef const struct {
3008        FieldUpgradeData_Entry    *entry;
3009        UINT16                    inSize;
3010        UINT16                    outSize;
3011        UINT16                    offsetOfTypes;
3012        UINT16                    paramOffsets[1];
3013        BYTE                      types[5];
```

```
3014    } FieldUpgradeData_COMMAND_DESCRIPTOR_t;
3015    FieldUpgradeData_COMMAND_DESCRIPTOR_t _FieldUpgradeDataData = {
3016    /* entry  */          &TPM2_FieldUpgradeData,
3017    /* inSize */          (UINT16)(sizeof(FieldUpgradeData_In)),
3018    /* outSize */         (UINT16)(sizeof(FieldUpgradeData_Out)),
3019    /* offsetOfTypes */   offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t, types),
3020    /* offsets */         {(UINT16)(offsetof(FieldUpgradeData_Out, firstDigest))},
3021    /* types */           {TPM2B_MAX_BUFFER_P_UNMARSHAL,
3022                           END_OF_LIST,
3023                           TPMT_HA_P_MARSHAL,
3024                           TPMT_HA_P_MARSHAL,
3025                           END_OF_LIST}
3026    };
3027    #define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
3028    #else
3029    #define _FieldUpgradeDataDataAddress 0
3030    #endif
3031    #if CC_FirmwareRead == YES
3032    #include "FirmwareRead_fp.h"
3033    typedef TPM_RC  (FirmwareRead_Entry)(
3034        FirmwareRead_In *in,
3035        FirmwareRead_Out *out
3036    );
3037    typedef const struct {
3038        FirmwareRead_Entry    *entry;
3039        UINT16                inSize;
3040        UINT16                outSize;
3041        UINT16                offsetOfTypes;
3042        BYTE                  types[4];
3043    } FirmwareRead_COMMAND_DESCRIPTOR_t;
3044    FirmwareRead_COMMAND_DESCRIPTOR_t _FirmwareReadData = {
3045    /* entry  */          &TPM2_FirmwareRead,
3046    /* inSize */          (UINT16)(sizeof(FirmwareRead_In)),
3047    /* outSize */         (UINT16)(sizeof(FirmwareRead_Out)),
3048    /* offsetOfTypes */   offsetof(FirmwareRead_COMMAND_DESCRIPTOR_t, types),
3049    /* offsets */         // No parameter offsets
3050    /* types */           {UINT32_P_UNMARSHAL,
3051                           END_OF_LIST,
3052                           TPM2B_MAX_BUFFER_P_MARSHAL,
3053                           END_OF_LIST}
3054    };
3055    #define _FirmwareReadDataAddress (&_FirmwareReadData)
3056    #else
3057    #define _FirmwareReadDataAddress 0
3058    #endif
3059    #if CC_ContextSave == YES
3060    #include "ContextSave_fp.h"
3061    typedef TPM_RC  (ContextSave_Entry)(
3062        ContextSave_In *in,
3063        ContextSave_Out *out
3064    );
3065    typedef const struct {
3066        ContextSave_Entry    *entry;
3067        UINT16               inSize;
3068        UINT16               outSize;
3069        UINT16               offsetOfTypes;
3070        BYTE                 types[4];
3071    } ContextSave_COMMAND_DESCRIPTOR_t;
3072    ContextSave_COMMAND_DESCRIPTOR_t _ContextSaveData = {
3073    /* entry  */          &TPM2_ContextSave,
3074    /* inSize */          (UINT16)(sizeof(ContextSave_In)),
3075    /* outSize */         (UINT16)(sizeof(ContextSave_Out)),
3076    /* offsetOfTypes */   offsetof(ContextSave_COMMAND_DESCRIPTOR_t, types),
3077    /* offsets */         // No parameter offsets
3078    /* types */           {TPMI_DH_CONTEXT_H_UNMARSHAL,
3079                           END_OF_LIST,
```

```
3080                        TPMS_CONTEXT_P_MARSHAL,
3081                        END_OF_LIST}
3082    };
3083    #define _ContextSaveDataAddress (&_ContextSaveData)
3084    #else
3085    #define _ContextSaveDataAddress 0
3086    #endif
3087    #if CC_ContextLoad == YES
3088    #include "ContextLoad_fp.h"
3089    typedef TPM_RC  (ContextLoad_Entry)(
3090        ContextLoad_In *in,
3091        ContextLoad_Out *out
3092    );
3093    typedef const struct {
3094        ContextLoad_Entry    *entry;
3095        UINT16               inSize;
3096        UINT16               outSize;
3097        UINT16               offsetOfTypes;
3098        BYTE                 types[4];
3099    } ContextLoad_COMMAND_DESCRIPTOR_t;
3100    ContextLoad_COMMAND_DESCRIPTOR_t _ContextLoadData = {
3101    /* entry  */          &TPM2_ContextLoad,
3102    /* inSize */          (UINT16)(sizeof(ContextLoad_In)),
3103    /* outSize */         (UINT16)(sizeof(ContextLoad_Out)),
3104    /* offsetOfTypes */   offsetof(ContextLoad_COMMAND_DESCRIPTOR_t, types),
3105    /* offsets */         // No parameter offsets
3106    /* types */           {TPMS_CONTEXT_P_UNMARSHAL,
3107                           END_OF_LIST,
3108                           UINT32_H_MARSHAL,
3109                           END_OF_LIST}
3110    };
3111    #define _ContextLoadDataAddress (&_ContextLoadData)
3112    #else
3113    #define _ContextLoadDataAddress 0
3114    #endif
3115    #if CC_FlushContext == YES
3116    #include "FlushContext_fp.h"
3117    typedef TPM_RC  (FlushContext_Entry)(
3118        FlushContext_In *in
3119    );
3120    typedef const struct {
3121        FlushContext_Entry    *entry;
3122        UINT16                inSize;
3123        UINT16                outSize;
3124        UINT16                offsetOfTypes;
3125        BYTE                  types[3];
3126    } FlushContext_COMMAND_DESCRIPTOR_t;
3127    FlushContext_COMMAND_DESCRIPTOR_t _FlushContextData = {
3128    /* entry  */          &TPM2_FlushContext,
3129    /* inSize */          (UINT16)(sizeof(FlushContext_In)),
3130    /* outSize */         0,
3131    /* offsetOfTypes */   offsetof(FlushContext_COMMAND_DESCRIPTOR_t, types),
3132    /* offsets */         // No parameter offsets
3133    /* types */           {TPMI_DH_CONTEXT_P_UNMARSHAL,
3134                           END_OF_LIST,
3135                           END_OF_LIST}
3136    };
3137    #define _FlushContextDataAddress (&_FlushContextData)
3138    #else
3139    #define _FlushContextDataAddress 0
3140    #endif
3141    #if CC_EvictControl == YES
3142    #include "EvictControl_fp.h"
3143    typedef TPM_RC  (EvictControl_Entry)(
3144        EvictControl_In *in
3145    );
```

```
3146    typedef const struct {
3147        EvictControl_Entry    *entry;
3148        UINT16                inSize;
3149        UINT16                outSize;
3150        UINT16                offsetOfTypes;
3151        UINT16                paramOffsets[2];
3152        BYTE                  types[5];
3153    } EvictControl_COMMAND_DESCRIPTOR_t;
3154    EvictControl_COMMAND_DESCRIPTOR_t _EvictControlData = {
3155    /* entry  */          &TPM2_EvictControl,
3156    /* inSize */          (UINT16)(sizeof(EvictControl_In)),
3157    /* outSize */         0,
3158    /* offsetOfTypes */   offsetof(EvictControl_COMMAND_DESCRIPTOR_t, types),
3159    /* offsets */         {(UINT16)(offsetof(EvictControl_In, objectHandle)),
3160                           (UINT16)(offsetof(EvictControl_In, persistentHandle))},
3161    /* types */           {TPMI_RH_PROVISION_H_UNMARSHAL,
3162                           TPMI_DH_OBJECT_H_UNMARSHAL,
3163                           TPMI_DH_PERSISTENT_P_UNMARSHAL,
3164                           END_OF_LIST,
3165                           END_OF_LIST}
3166    };
3167    #define _EvictControlDataAddress (&_EvictControlData)
3168    #else
3169    #define _EvictControlDataAddress 0
3170    #endif
3171    #if CC_ReadClock == YES
3172    #include "ReadClock_fp.h"
3173    typedef TPM_RC  (ReadClock_Entry)(
3174        ReadClock_Out *out
3175    );
3176    typedef const struct {
3177        ReadClock_Entry    *entry;
3178        UINT16             inSize;
3179        UINT16             outSize;
3180        UINT16             offsetOfTypes;
3181        BYTE               types[3];
3182    } ReadClock_COMMAND_DESCRIPTOR_t;
3183    ReadClock_COMMAND_DESCRIPTOR_t _ReadClockData = {
3184    /* entry  */          &TPM2_ReadClock,
3185    /* inSize */          0,
3186    /* outSize */         (UINT16)(sizeof(ReadClock_Out)),
3187    /* offsetOfTypes */   offsetof(ReadClock_COMMAND_DESCRIPTOR_t, types),
3188    /* offsets */         // No parameter offsets
3189    /* types */           {END_OF_LIST,
3190                           TPMS_TIME_INFO_P_MARSHAL,
3191                           END_OF_LIST}
3192    };
3193    #define _ReadClockDataAddress (&_ReadClockData)
3194    #else
3195    #define _ReadClockDataAddress 0
3196    #endif
3197    #if CC_ClockSet == YES
3198    #include "ClockSet_fp.h"
3199    typedef TPM_RC  (ClockSet_Entry)(
3200        ClockSet_In *in
3201    );
3202    typedef const struct {
3203        ClockSet_Entry    *entry;
3204        UINT16            inSize;
3205        UINT16            outSize;
3206        UINT16            offsetOfTypes;
3207        UINT16            paramOffsets[1];
3208        BYTE              types[4];
3209    } ClockSet_COMMAND_DESCRIPTOR_t;
3210    ClockSet_COMMAND_DESCRIPTOR_t _ClockSetData = {
3211    /* entry  */          &TPM2_ClockSet,
```

```
3212    /* inSize */          (UINT16)(sizeof(ClockSet_In)),
3213    /* outSize */         0,
3214    /* offsetOfTypes */   offsetof(ClockSet_COMMAND_DESCRIPTOR_t, types),
3215    /* offsets */         {(UINT16)(offsetof(ClockSet_In, newTime))},
3216    /* types */           {TPMI_RH_PROVISION_H_UNMARSHAL,
3217                           UINT64_P_UNMARSHAL,
3218                           END_OF_LIST,
3219                           END_OF_LIST}
3220    };
3221    #define _ClockSetDataAddress (&_ClockSetData)
3222    #else
3223    #define _ClockSetDataAddress 0
3224    #endif
3225    #if CC_ClockRateAdjust == YES
3226    #include "ClockRateAdjust_fp.h"
3227    typedef TPM_RC  (ClockRateAdjust_Entry)(
3228        ClockRateAdjust_In *in
3229    );
3230    typedef const struct {
3231        ClockRateAdjust_Entry    *entry;
3232        UINT16                   inSize;
3233        UINT16                   outSize;
3234        UINT16                   offsetOfTypes;
3235        UINT16                   paramOffsets[1];
3236        BYTE                     types[4];
3237    } ClockRateAdjust_COMMAND_DESCRIPTOR_t;
3238    ClockRateAdjust_COMMAND_DESCRIPTOR_t _ClockRateAdjustData = {
3239    /* entry  */          &TPM2_ClockRateAdjust,
3240    /* inSize */          (UINT16)(sizeof(ClockRateAdjust_In)),
3241    /* outSize */         0,
3242    /* offsetOfTypes */   offsetof(ClockRateAdjust_COMMAND_DESCRIPTOR_t, types),
3243    /* offsets */         {(UINT16)(offsetof(ClockRateAdjust_In, rateAdjust))},
3244    /* types */           {TPMI_RH_PROVISION_H_UNMARSHAL,
3245                           TPM_CLOCK_ADJUST_P_UNMARSHAL,
3246                           END_OF_LIST,
3247                           END_OF_LIST}
3248    };
3249    #define _ClockRateAdjustDataAddress (&_ClockRateAdjustData)
3250    #else
3251    #define _ClockRateAdjustDataAddress 0
3252    #endif
3253    #if CC_GetCapability == YES
3254    #include "GetCapability_fp.h"
3255    typedef TPM_RC  (GetCapability_Entry)(
3256        GetCapability_In *in,
3257        GetCapability_Out *out
3258    );
3259    typedef const struct {
3260        GetCapability_Entry    *entry;
3261        UINT16                 inSize;
3262        UINT16                 outSize;
3263        UINT16                 offsetOfTypes;
3264        UINT16                 paramOffsets[3];
3265        BYTE                   types[7];
3266    } GetCapability_COMMAND_DESCRIPTOR_t;
3267    GetCapability_COMMAND_DESCRIPTOR_t _GetCapabilityData = {
3268    /* entry  */          &TPM2_GetCapability,
3269    /* inSize */          (UINT16)(sizeof(GetCapability_In)),
3270    /* outSize */         (UINT16)(sizeof(GetCapability_Out)),
3271    /* offsetOfTypes */   offsetof(GetCapability_COMMAND_DESCRIPTOR_t, types),
3272    /* offsets */         {(UINT16)(offsetof(GetCapability_In, property)),
3273                           (UINT16)(offsetof(GetCapability_In, propertyCount)),
3274                           (UINT16)(offsetof(GetCapability_Out, capabilityData))},
3275    /* types */           {TPM_CAP_P_UNMARSHAL,
3276                           UINT32_P_UNMARSHAL,
3277                           UINT32_P_UNMARSHAL,
```

```
3278                        END_OF_LIST,
3279                        UINT8_P_MARSHAL,
3280                        TPMS_CAPABILITY_DATA_P_MARSHAL,
3281                        END_OF_LIST}
3282    };
3283    #define _GetCapabilityDataAddress (&_GetCapabilityData)
3284    #else
3285    #define _GetCapabilityDataAddress 0
3286    #endif
3287    #if CC_TestParms == YES
3288    #include "TestParms_fp.h"
3289    typedef TPM_RC  (TestParms_Entry)(
3290        TestParms_In *in
3291    );
3292    typedef const struct {
3293        TestParms_Entry    *entry;
3294        UINT16             inSize;
3295        UINT16             outSize;
3296        UINT16             offsetOfTypes;
3297        BYTE               types[3];
3298    } TestParms_COMMAND_DESCRIPTOR_t;
3299    TestParms_COMMAND_DESCRIPTOR_t _TestParmsData = {
3300    /* entry  */          &TPM2_TestParms,
3301    /* inSize */          (UINT16)(sizeof(TestParms_In)),
3302    /* outSize */         0,
3303    /* offsetOfTypes */   offsetof(TestParms_COMMAND_DESCRIPTOR_t, types),
3304    /* offsets */         // No parameter offsets
3305    /* types */           {TPMT_PUBLIC_PARMS_P_UNMARSHAL,
3306                           END_OF_LIST,
3307                           END_OF_LIST}
3308    };
3309    #define _TestParmsDataAddress (&_TestParmsData)
3310    #else
3311    #define _TestParmsDataAddress 0
3312    #endif
3313    #if CC_NV_DefineSpace == YES
3314    #include "NV_DefineSpace_fp.h"
3315    typedef TPM_RC  (NV_DefineSpace_Entry)(
3316        NV_DefineSpace_In *in
3317    );
3318    typedef const struct {
3319        NV_DefineSpace_Entry    *entry;
3320        UINT16                  inSize;
3321        UINT16                  outSize;
3322        UINT16                  offsetOfTypes;
3323        UINT16                  paramOffsets[2];
3324        BYTE                    types[5];
3325    } NV_DefineSpace_COMMAND_DESCRIPTOR_t;
3326    NV_DefineSpace_COMMAND_DESCRIPTOR_t _NV_DefineSpaceData = {
3327    /* entry  */          &TPM2_NV_DefineSpace,
3328    /* inSize */          (UINT16)(sizeof(NV_DefineSpace_In)),
3329    /* outSize */         0,
3330    /* offsetOfTypes */   offsetof(NV_DefineSpace_COMMAND_DESCRIPTOR_t, types),
3331    /* offsets */         {(UINT16)(offsetof(NV_DefineSpace_In, auth)),
3332                           (UINT16)(offsetof(NV_DefineSpace_In, publicInfo))},
3333    /* types */           {TPMI_RH_PROVISION_H_UNMARSHAL,
3334                           TPM2B_DIGEST_P_UNMARSHAL,
3335                           TPM2B_NV_PUBLIC_P_UNMARSHAL,
3336                           END_OF_LIST,
3337                           END_OF_LIST}
3338    };
3339    #define _NV_DefineSpaceDataAddress (&_NV_DefineSpaceData)
3340    #else
3341    #define _NV_DefineSpaceDataAddress 0
3342    #endif
3343    #if CC_NV_UndefineSpace == YES
```

```
3344    #include "NV_UndefineSpace_fp.h"
3345    typedef TPM_RC  (NV_UndefineSpace_Entry)(
3346        NV_UndefineSpace_In *in
3347    );
3348    typedef const struct {
3349        NV_UndefineSpace_Entry    *entry;
3350        UINT16                    inSize;
3351        UINT16                    outSize;
3352        UINT16                    offsetOfTypes;
3353        UINT16                    paramOffsets[1];
3354        BYTE                      types[4];
3355    } NV_UndefineSpace_COMMAND_DESCRIPTOR_t;
3356    NV_UndefineSpace_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceData = {
3357    /* entry  */          &TPM2_NV_UndefineSpace,
3358    /* inSize */          (UINT16)(sizeof(NV_UndefineSpace_In)),
3359    /* outSize */         0,
3360    /* offsetOfTypes */   offsetof(NV_UndefineSpace_COMMAND_DESCRIPTOR_t, types),
3361    /* offsets */         {(UINT16)(offsetof(NV_UndefineSpace_In, nvIndex))},
3362    /* types */           {TPMI_RH_PROVISION_H_UNMARSHAL,
3363                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
3364                           END_OF_LIST,
3365                           END_OF_LIST}
3366    };
3367    #define _NV_UndefineSpaceDataAddress (&_NV_UndefineSpaceData)
3368    #else
3369    #define _NV_UndefineSpaceDataAddress 0
3370    #endif
3371    #if CC_NV_UndefineSpaceSpecial == YES
3372    #include "NV_UndefineSpaceSpecial_fp.h"
3373    typedef TPM_RC  (NV_UndefineSpaceSpecial_Entry)(
3374        NV_UndefineSpaceSpecial_In *in
3375    );
3376    typedef const struct {
3377        NV_UndefineSpaceSpecial_Entry    *entry;
3378        UINT16                           inSize;
3379        UINT16                           outSize;
3380        UINT16                           offsetOfTypes;
3381        UINT16                           paramOffsets[1];
3382        BYTE                             types[4];
3383    } NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t;
3384    NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceSpecialData = {
3385    /* entry  */          &TPM2_NV_UndefineSpaceSpecial,
3386    /* inSize */          (UINT16)(sizeof(NV_UndefineSpaceSpecial_In)),
3387    /* outSize */         0,
3388    /* offsetOfTypes */   offsetof(NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t, types),
3389    /* offsets */         {(UINT16)(offsetof(NV_UndefineSpaceSpecial_In, platform))},
3390    /* types */           {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3391                           TPMI_RH_PLATFORM_H_UNMARSHAL,
3392                           END_OF_LIST,
3393                           END_OF_LIST}
3394    };
3395    #define _NV_UndefineSpaceSpecialDataAddress (&_NV_UndefineSpaceSpecialData)
3396    #else
3397    #define _NV_UndefineSpaceSpecialDataAddress 0
3398    #endif
3399    #if CC_NV_ReadPublic == YES
3400    #include "NV_ReadPublic_fp.h"
3401    typedef TPM_RC  (NV_ReadPublic_Entry)(
3402        NV_ReadPublic_In *in,
3403        NV_ReadPublic_Out *out
3404    );
3405    typedef const struct {
3406        NV_ReadPublic_Entry    *entry;
3407        UINT16                 inSize;
3408        UINT16                 outSize;
3409        UINT16                 offsetOfTypes;
```

```
3410        UINT16                  paramOffsets[1];
3411        BYTE                    types[5];
3412    } NV_ReadPublic_COMMAND_DESCRIPTOR_t;
3413    NV_ReadPublic_COMMAND_DESCRIPTOR_t _NV_ReadPublicData = {
3414    /* entry  */            &TPM2_NV_ReadPublic,
3415    /* inSize */            (UINT16)(sizeof(NV_ReadPublic_In)),
3416    /* outSize */           (UINT16)(sizeof(NV_ReadPublic_Out)),
3417    /* offsetOfTypes */     offsetof(NV_ReadPublic_COMMAND_DESCRIPTOR_t, types),
3418    /* offsets */           {(UINT16)(offsetof(NV_ReadPublic_Out, nvName))},
3419    /* types */             {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3420                             END_OF_LIST,
3421                             TPM2B_NV_PUBLIC_P_MARSHAL,
3422                             TPM2B_NAME_P_MARSHAL,
3423                             END_OF_LIST}
3424    };
3425    #define _NV_ReadPublicDataAddress (&_NV_ReadPublicData)
3426    #else
3427    #define _NV_ReadPublicDataAddress 0
3428    #endif
3429    #if CC_NV_Write == YES
3430    #include "NV_Write_fp.h"
3431    typedef TPM_RC  (NV_Write_Entry)(
3432        NV_Write_In *in
3433    );
3434    typedef const struct {
3435        NV_Write_Entry    *entry;
3436        UINT16            inSize;
3437        UINT16            outSize;
3438        UINT16            offsetOfTypes;
3439        UINT16            paramOffsets[3];
3440        BYTE              types[6];
3441    } NV_Write_COMMAND_DESCRIPTOR_t;
3442    NV_Write_COMMAND_DESCRIPTOR_t _NV_WriteData = {
3443    /* entry  */            &TPM2_NV_Write,
3444    /* inSize */            (UINT16)(sizeof(NV_Write_In)),
3445    /* outSize */           0,
3446    /* offsetOfTypes */     offsetof(NV_Write_COMMAND_DESCRIPTOR_t, types),
3447    /* offsets */           {(UINT16)(offsetof(NV_Write_In, nvIndex)),
3448                             (UINT16)(offsetof(NV_Write_In, data)),
3449                             (UINT16)(offsetof(NV_Write_In, offset))},
3450    /* types */             {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3451                             TPMI_RH_NV_INDEX_H_UNMARSHAL,
3452                             TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3453                             UINT16_P_UNMARSHAL,
3454                             END_OF_LIST,
3455                             END_OF_LIST}
3456    };
3457    #define _NV_WriteDataAddress (&_NV_WriteData)
3458    #else
3459    #define _NV_WriteDataAddress 0
3460    #endif
3461    #if CC_NV_Increment == YES
3462    #include "NV_Increment_fp.h"
3463    typedef TPM_RC  (NV_Increment_Entry)(
3464        NV_Increment_In *in
3465    );
3466    typedef const struct {
3467        NV_Increment_Entry    *entry;
3468        UINT16                inSize;
3469        UINT16                outSize;
3470        UINT16                offsetOfTypes;
3471        UINT16                paramOffsets[1];
3472        BYTE                  types[4];
3473    } NV_Increment_COMMAND_DESCRIPTOR_t;
3474    NV_Increment_COMMAND_DESCRIPTOR_t _NV_IncrementData = {
3475    /* entry  */            &TPM2_NV_Increment,
```

```
3476    /* inSize */          (UINT16)(sizeof(NV_Increment_In)),
3477    /* outSize */         0,
3478    /* offsetOfTypes */   offsetof(NV_Increment_COMMAND_DESCRIPTOR_t, types),
3479    /* offsets */         {(UINT16)(offsetof(NV_Increment_In, nvIndex))},
3480    /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3481                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
3482                           END_OF_LIST,
3483                           END_OF_LIST}
3484    };
3485    #define _NV_IncrementDataAddress (&_NV_IncrementData)
3486    #else
3487    #define _NV_IncrementDataAddress 0
3488    #endif
3489    #if CC_NV_Extend == YES
3490    #include "NV_Extend_fp.h"
3491    typedef TPM_RC  (NV_Extend_Entry)(
3492        NV_Extend_In *in
3493    );
3494    typedef const struct {
3495        NV_Extend_Entry    *entry;
3496        UINT16              inSize;
3497        UINT16              outSize;
3498        UINT16              offsetOfTypes;
3499        UINT16              paramOffsets[2];
3500        BYTE                types[5];
3501    } NV_Extend_COMMAND_DESCRIPTOR_t;
3502    NV_Extend_COMMAND_DESCRIPTOR_t _NV_ExtendData = {
3503    /* entry  */          &TPM2_NV_Extend,
3504    /* inSize */          (UINT16)(sizeof(NV_Extend_In)),
3505    /* outSize */         0,
3506    /* offsetOfTypes */   offsetof(NV_Extend_COMMAND_DESCRIPTOR_t, types),
3507    /* offsets */         {(UINT16)(offsetof(NV_Extend_In, nvIndex)),
3508                           (UINT16)(offsetof(NV_Extend_In, data))},
3509    /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3510                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
3511                           TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3512                           END_OF_LIST,
3513                           END_OF_LIST}
3514    };
3515    #define _NV_ExtendDataAddress (&_NV_ExtendData)
3516    #else
3517    #define _NV_ExtendDataAddress 0
3518    #endif
3519    #if CC_NV_SetBits == YES
3520    #include "NV_SetBits_fp.h"
3521    typedef TPM_RC  (NV_SetBits_Entry)(
3522        NV_SetBits_In *in
3523    );
3524    typedef const struct {
3525        NV_SetBits_Entry    *entry;
3526        UINT16              inSize;
3527        UINT16              outSize;
3528        UINT16              offsetOfTypes;
3529        UINT16              paramOffsets[2];
3530        BYTE                types[5];
3531    } NV_SetBits_COMMAND_DESCRIPTOR_t;
3532    NV_SetBits_COMMAND_DESCRIPTOR_t _NV_SetBitsData = {
3533    /* entry  */          &TPM2_NV_SetBits,
3534    /* inSize */          (UINT16)(sizeof(NV_SetBits_In)),
3535    /* outSize */         0,
3536    /* offsetOfTypes */   offsetof(NV_SetBits_COMMAND_DESCRIPTOR_t, types),
3537    /* offsets */         {(UINT16)(offsetof(NV_SetBits_In, nvIndex)),
3538                           (UINT16)(offsetof(NV_SetBits_In, bits))},
3539    /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3540                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
3541                           UINT64_P_UNMARSHAL,
```

```
3542                        END_OF_LIST,
3543                        END_OF_LIST}
3544    };
3545    #define _NV_SetBitsDataAddress (&_NV_SetBitsData)
3546    #else
3547    #define _NV_SetBitsDataAddress 0
3548    #endif
3549    #if CC_NV_WriteLock == YES
3550    #include "NV_WriteLock_fp.h"
3551    typedef TPM_RC  (NV_WriteLock_Entry)(
3552        NV_WriteLock_In *in
3553    );
3554    typedef const struct {
3555        NV_WriteLock_Entry    *entry;
3556        UINT16                inSize;
3557        UINT16                outSize;
3558        UINT16                offsetOfTypes;
3559        UINT16                paramOffsets[1];
3560        BYTE                  types[4];
3561    } NV_WriteLock_COMMAND_DESCRIPTOR_t;
3562    NV_WriteLock_COMMAND_DESCRIPTOR_t _NV_WriteLockData = {
3563    /* entry  */          &TPM2_NV_WriteLock,
3564    /* inSize */          (UINT16)(sizeof(NV_WriteLock_In)),
3565    /* outSize */         0,
3566    /* offsetOfTypes */   offsetof(NV_WriteLock_COMMAND_DESCRIPTOR_t, types),
3567    /* offsets */         {(UINT16)(offsetof(NV_WriteLock_In, nvIndex))},
3568    /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3569                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
3570                           END_OF_LIST,
3571                           END_OF_LIST}
3572    };
3573    #define _NV_WriteLockDataAddress (&_NV_WriteLockData)
3574    #else
3575    #define _NV_WriteLockDataAddress 0
3576    #endif
3577    #if CC_NV_GlobalWriteLock == YES
3578    #include "NV_GlobalWriteLock_fp.h"
3579    typedef TPM_RC  (NV_GlobalWriteLock_Entry)(
3580        NV_GlobalWriteLock_In *in
3581    );
3582    typedef const struct {
3583        NV_GlobalWriteLock_Entry    *entry;
3584        UINT16                      inSize;
3585        UINT16                      outSize;
3586        UINT16                      offsetOfTypes;
3587        BYTE                        types[3];
3588    } NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t;
3589    NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t _NV_GlobalWriteLockData = {
3590    /* entry  */          &TPM2_NV_GlobalWriteLock,
3591    /* inSize */          (UINT16)(sizeof(NV_GlobalWriteLock_In)),
3592    /* outSize */         0,
3593    /* offsetOfTypes */   offsetof(NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t, types),
3594    /* offsets */         // No parameter offsets
3595    /* types */           {TPMI_RH_PROVISION_H_UNMARSHAL,
3596                           END_OF_LIST,
3597                           END_OF_LIST}
3598    };
3599    #define _NV_GlobalWriteLockDataAddress (&_NV_GlobalWriteLockData)
3600    #else
3601    #define _NV_GlobalWriteLockDataAddress 0
3602    #endif
3603    #if CC_NV_Read == YES
3604    #include "NV_Read_fp.h"
3605    typedef TPM_RC  (NV_Read_Entry)(
3606        NV_Read_In *in,
3607        NV_Read_Out *out
```

```
3608    );
3609    typedef const struct {
3610        NV_Read_Entry    *entry;
3611        UINT16           inSize;
3612        UINT16           outSize;
3613        UINT16           offsetOfTypes;
3614        UINT16           paramOffsets[3];
3615        BYTE             types[7];
3616    } NV_Read_COMMAND_DESCRIPTOR_t;
3617    NV_Read_COMMAND_DESCRIPTOR_t _NV_ReadData = {
3618    /* entry  */          &TPM2_NV_Read,
3619    /* inSize */          (UINT16)(sizeof(NV_Read_In)),
3620    /* outSize */         (UINT16)(sizeof(NV_Read_Out)),
3621    /* offsetOfTypes */   offsetof(NV_Read_COMMAND_DESCRIPTOR_t, types),
3622    /* offsets */         {(UINT16)(offsetof(NV_Read_In, nvIndex)),
3623                           (UINT16)(offsetof(NV_Read_In, size)),
3624                           (UINT16)(offsetof(NV_Read_In, offset))},
3625    /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3626                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
3627                           UINT16_P_UNMARSHAL,
3628                           UINT16_P_UNMARSHAL,
3629                           END_OF_LIST,
3630                           TPM2B_MAX_NV_BUFFER_P_MARSHAL,
3631                           END_OF_LIST}
3632    };
3633    #define _NV_ReadDataAddress (&_NV_ReadData)
3634    #else
3635    #define _NV_ReadDataAddress 0
3636    #endif
3637    #if CC_NV_ReadLock == YES
3638    #include "NV_ReadLock_fp.h"
3639    typedef TPM_RC  (NV_ReadLock_Entry)(
3640        NV_ReadLock_In *in
3641    );
3642    typedef const struct {
3643        NV_ReadLock_Entry    *entry;
3644        UINT16               inSize;
3645        UINT16               outSize;
3646        UINT16               offsetOfTypes;
3647        UINT16               paramOffsets[1];
3648        BYTE                 types[4];
3649    } NV_ReadLock_COMMAND_DESCRIPTOR_t;
3650    NV_ReadLock_COMMAND_DESCRIPTOR_t _NV_ReadLockData = {
3651    /* entry  */          &TPM2_NV_ReadLock,
3652    /* inSize */          (UINT16)(sizeof(NV_ReadLock_In)),
3653    /* outSize */         0,
3654    /* offsetOfTypes */   offsetof(NV_ReadLock_COMMAND_DESCRIPTOR_t, types),
3655    /* offsets */         {(UINT16)(offsetof(NV_ReadLock_In, nvIndex))},
3656    /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3657                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
3658                           END_OF_LIST,
3659                           END_OF_LIST}
3660    };
3661    #define _NV_ReadLockDataAddress (&_NV_ReadLockData)
3662    #else
3663    #define _NV_ReadLockDataAddress 0
3664    #endif
3665    #if CC_NV_ChangeAuth == YES
3666    #include "NV_ChangeAuth_fp.h"
3667    typedef TPM_RC  (NV_ChangeAuth_Entry)(
3668        NV_ChangeAuth_In *in
3669    );
3670    typedef const struct {
3671        NV_ChangeAuth_Entry    *entry;
3672        UINT16                 inSize;
3673        UINT16                 outSize;
```

```
3674        UINT16                  offsetOfTypes;
3675        UINT16                  paramOffsets[1];
3676        BYTE                    types[4];
3677    } NV_ChangeAuth_COMMAND_DESCRIPTOR_t;
3678    NV_ChangeAuth_COMMAND_DESCRIPTOR_t _NV_ChangeAuthData = {
3679    /* entry  */           &TPM2_NV_ChangeAuth,
3680    /* inSize */           (UINT16)(sizeof(NV_ChangeAuth_In)),
3681    /* outSize */          0,
3682    /* offsetOfTypes */    offsetof(NV_ChangeAuth_COMMAND_DESCRIPTOR_t, types),
3683    /* offsets */          {(UINT16)(offsetof(NV_ChangeAuth_In, newAuth))},
3684    /* types */            {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3685                            TPM2B_DIGEST_P_UNMARSHAL,
3686                            END_OF_LIST,
3687                            END_OF_LIST}
3688    };
3689    #define _NV_ChangeAuthDataAddress (&_NV_ChangeAuthData)
3690    #else
3691    #define _NV_ChangeAuthDataAddress 0
3692    #endif
3693    #if CC_NV_Certify == YES
3694    #include "NV_Certify_fp.h"
3695    typedef TPM_RC  (NV_Certify_Entry)(
3696        NV_Certify_In *in,
3697        NV_Certify_Out *out
3698    );
3699    typedef const struct {
3700        NV_Certify_Entry    *entry;
3701        UINT16              inSize;
3702        UINT16              outSize;
3703        UINT16              offsetOfTypes;
3704        UINT16              paramOffsets[7];
3705        BYTE                types[11];
3706    } NV_Certify_COMMAND_DESCRIPTOR_t;
3707    NV_Certify_COMMAND_DESCRIPTOR_t _NV_CertifyData = {
3708    /* entry  */           &TPM2_NV_Certify,
3709    /* inSize */           (UINT16)(sizeof(NV_Certify_In)),
3710    /* outSize */          (UINT16)(sizeof(NV_Certify_Out)),
3711    /* offsetOfTypes */    offsetof(NV_Certify_COMMAND_DESCRIPTOR_t, types),
3712    /* offsets */          {(UINT16)(offsetof(NV_Certify_In, authHandle)),
3713                            (UINT16)(offsetof(NV_Certify_In, nvIndex)),
3714                            (UINT16)(offsetof(NV_Certify_In, qualifyingData)),
3715                            (UINT16)(offsetof(NV_Certify_In, inScheme)),
3716                            (UINT16)(offsetof(NV_Certify_In, size)),
3717                            (UINT16)(offsetof(NV_Certify_In, offset)),
3718                            (UINT16)(offsetof(NV_Certify_Out, signature))},
3719    /* types */            {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
3720                            TPMI_RH_NV_AUTH_H_UNMARSHAL,
3721                            TPMI_RH_NV_INDEX_H_UNMARSHAL,
3722                            TPM2B_DATA_P_UNMARSHAL,
3723                            TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
3724                            UINT16_P_UNMARSHAL,
3725                            UINT16_P_UNMARSHAL,
3726                            END_OF_LIST,
3727                            TPM2B_ATTEST_P_MARSHAL,
3728                            TPMT_SIGNATURE_P_MARSHAL,
3729                            END_OF_LIST}
3730    };
3731    #define _NV_CertifyDataAddress (&_NV_CertifyData)
3732    #else
3733    #define _NV_CertifyDataAddress 0
3734    #endif
3735    #if CC_Vendor_TCG_Test == YES
3736    #include "Vendor_TCG_Test_fp.h"
3737    typedef TPM_RC  (Vendor_TCG_Test_Entry)(
3738        Vendor_TCG_Test_In *in,
3739        Vendor_TCG_Test_Out *out
```

```
3740    );
3741    typedef const struct {
3742        Vendor_TCG_Test_Entry    *entry;
3743        UINT16                   inSize;
3744        UINT16                   outSize;
3745        UINT16                   offsetOfTypes;
3746        BYTE                     types[4];
3747    } Vendor_TCG_Test_COMMAND_DESCRIPTOR_t;
3748    Vendor_TCG_Test_COMMAND_DESCRIPTOR_t _Vendor_TCG_TestData = {
3749    /* entry  */          &TPM2_Vendor_TCG_Test,
3750    /* inSize */          (UINT16)(sizeof(Vendor_TCG_Test_In)),
3751    /* outSize */         (UINT16)(sizeof(Vendor_TCG_Test_Out)),
3752    /* offsetOfTypes */   offsetof(Vendor_TCG_Test_COMMAND_DESCRIPTOR_t, types),
3753    /* offsets */         // No parameter offsets
3754    /* types */           {TPM2B_DATA_P_UNMARSHAL,
3755                           END_OF_LIST,
3756                           TPM2B_DATA_P_MARSHAL,
3757                           END_OF_LIST}
3758    };
3759    #define _Vendor_TCG_TestDataAddress (&_Vendor_TCG_TestData)
3760    #else
3761    #define _Vendor_TCG_TestDataAddress 0
3762    #endif
3763    COMMAND_DESCRIPTOR_t *s_CommandDataArray[] = {
3764    #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
3765            (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceSpecialDataAddress,
3766    #endif
3767    #if (PAD_LIST || CC_EvictControl)
3768            (COMMAND_DESCRIPTOR_t *)_EvictControlDataAddress,
3769    #endif
3770    #if (PAD_LIST || CC_HierarchyControl)
3771            (COMMAND_DESCRIPTOR_t *)_HierarchyControlDataAddress,
3772    #endif
3773    #if (PAD_LIST || CC_NV_UndefineSpace)
3774            (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceDataAddress,
3775    #endif
3776    #if (PAD_LIST)
3777            (COMMAND_DESCRIPTOR_t *)0,
3778    #endif
3779    #if (PAD_LIST || CC_ChangeEPS)
3780            (COMMAND_DESCRIPTOR_t *)_ChangeEPSDataAddress,
3781    #endif
3782    #if (PAD_LIST || CC_ChangePPS)
3783            (COMMAND_DESCRIPTOR_t *)_ChangePPSDataAddress,
3784    #endif
3785    #if (PAD_LIST || CC_Clear)
3786            (COMMAND_DESCRIPTOR_t *)_ClearDataAddress,
3787    #endif
3788    #if (PAD_LIST || CC_ClearControl)
3789            (COMMAND_DESCRIPTOR_t *)_ClearControlDataAddress,
3790    #endif
3791    #if (PAD_LIST || CC_ClockSet)
3792            (COMMAND_DESCRIPTOR_t *)_ClockSetDataAddress,
3793    #endif
3794    #if (PAD_LIST || CC_HierarchyChangeAuth)
3795            (COMMAND_DESCRIPTOR_t *)_HierarchyChangeAuthDataAddress,
3796    #endif
3797    #if (PAD_LIST || CC_NV_DefineSpace)
3798            (COMMAND_DESCRIPTOR_t *)_NV_DefineSpaceDataAddress,
3799    #endif
3800    #if (PAD_LIST || CC_PCR_Allocate)
3801            (COMMAND_DESCRIPTOR_t *)_PCR_AllocateDataAddress,
3802    #endif
3803    #if (PAD_LIST || CC_PCR_SetAuthPolicy)
3804            (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthPolicyDataAddress,
3805    #endif
```

```
3806    #if (PAD_LIST || CC_PP_Commands)
3807            (COMMAND_DESCRIPTOR_t *)_PP_CommandsDataAddress,
3808    #endif
3809    #if (PAD_LIST || CC_SetPrimaryPolicy)
3810            (COMMAND_DESCRIPTOR_t *)_SetPrimaryPolicyDataAddress,
3811    #endif
3812    #if (PAD_LIST || CC_FieldUpgradeStart)
3813            (COMMAND_DESCRIPTOR_t *)_FieldUpgradeStartDataAddress,
3814    #endif
3815    #if (PAD_LIST || CC_ClockRateAdjust)
3816            (COMMAND_DESCRIPTOR_t *)_ClockRateAdjustDataAddress,
3817    #endif
3818    #if (PAD_LIST || CC_CreatePrimary)
3819            (COMMAND_DESCRIPTOR_t *)_CreatePrimaryDataAddress,
3820    #endif
3821    #if (PAD_LIST || CC_NV_GlobalWriteLock)
3822            (COMMAND_DESCRIPTOR_t *)_NV_GlobalWriteLockDataAddress,
3823    #endif
3824    #if (PAD_LIST || CC_GetCommandAuditDigest)
3825            (COMMAND_DESCRIPTOR_t *)_GetCommandAuditDigestDataAddress,
3826    #endif
3827    #if (PAD_LIST || CC_NV_Increment)
3828            (COMMAND_DESCRIPTOR_t *)_NV_IncrementDataAddress,
3829    #endif
3830    #if (PAD_LIST || CC_NV_SetBits)
3831            (COMMAND_DESCRIPTOR_t *)_NV_SetBitsDataAddress,
3832    #endif
3833    #if (PAD_LIST || CC_NV_Extend)
3834            (COMMAND_DESCRIPTOR_t *)_NV_ExtendDataAddress,
3835    #endif
3836    #if (PAD_LIST || CC_NV_Write)
3837            (COMMAND_DESCRIPTOR_t *)_NV_WriteDataAddress,
3838    #endif
3839    #if (PAD_LIST || CC_NV_WriteLock)
3840            (COMMAND_DESCRIPTOR_t *)_NV_WriteLockDataAddress,
3841    #endif
3842    #if (PAD_LIST || CC_DictionaryAttackLockReset)
3843            (COMMAND_DESCRIPTOR_t *)_DictionaryAttackLockResetDataAddress,
3844    #endif
3845    #if (PAD_LIST || CC_DictionaryAttackParameters)
3846            (COMMAND_DESCRIPTOR_t *)_DictionaryAttackParametersDataAddress,
3847    #endif
3848    #if (PAD_LIST || CC_NV_ChangeAuth)
3849            (COMMAND_DESCRIPTOR_t *)_NV_ChangeAuthDataAddress,
3850    #endif
3851    #if (PAD_LIST || CC_PCR_Event)
3852            (COMMAND_DESCRIPTOR_t *)_PCR_EventDataAddress,
3853    #endif
3854    #if (PAD_LIST || CC_PCR_Reset)
3855            (COMMAND_DESCRIPTOR_t *)_PCR_ResetDataAddress,
3856    #endif
3857    #if (PAD_LIST || CC_SequenceComplete)
3858            (COMMAND_DESCRIPTOR_t *)_SequenceCompleteDataAddress,
3859    #endif
3860    #if (PAD_LIST || CC_SetAlgorithmSet)
3861            (COMMAND_DESCRIPTOR_t *)_SetAlgorithmSetDataAddress,
3862    #endif
3863    #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
3864            (COMMAND_DESCRIPTOR_t *)_SetCommandCodeAuditStatusDataAddress,
3865    #endif
3866    #if (PAD_LIST || CC_FieldUpgradeData)
3867            (COMMAND_DESCRIPTOR_t *)_FieldUpgradeDataDataAddress,
3868    #endif
3869    #if (PAD_LIST || CC_IncrementalSelfTest)
3870            (COMMAND_DESCRIPTOR_t *)_IncrementalSelfTestDataAddress,
3871    #endif
```



```
3872    #if (PAD_LIST || CC_SelfTest)
3873            (COMMAND_DESCRIPTOR_t *)_SelfTestDataAddress,
3874    #endif
3875    #if (PAD_LIST || CC_Startup)
3876            (COMMAND_DESCRIPTOR_t *)_StartupDataAddress,
3877    #endif
3878    #if (PAD_LIST || CC_Shutdown)
3879            (COMMAND_DESCRIPTOR_t *)_ShutdownDataAddress,
3880    #endif
3881    #if (PAD_LIST || CC_StirRandom)
3882            (COMMAND_DESCRIPTOR_t *)_StirRandomDataAddress,
3883    #endif
3884    #if (PAD_LIST || CC_ActivateCredential)
3885            (COMMAND_DESCRIPTOR_t *)_ActivateCredentialDataAddress,
3886    #endif
3887    #if (PAD_LIST || CC_Certify)
3888            (COMMAND_DESCRIPTOR_t *)_CertifyDataAddress,
3889    #endif
3890    #if (PAD_LIST || CC_PolicyNV)
3891            (COMMAND_DESCRIPTOR_t *)_PolicyNVDataAddress,
3892    #endif
3893    #if (PAD_LIST || CC_CertifyCreation)
3894            (COMMAND_DESCRIPTOR_t *)_CertifyCreationDataAddress,
3895    #endif
3896    #if (PAD_LIST || CC_Duplicate)
3897            (COMMAND_DESCRIPTOR_t *)_DuplicateDataAddress,
3898    #endif
3899    #if (PAD_LIST || CC_GetTime)
3900            (COMMAND_DESCRIPTOR_t *)_GetTimeDataAddress,
3901    #endif
3902    #if (PAD_LIST || CC_GetSessionAuditDigest)
3903            (COMMAND_DESCRIPTOR_t *)_GetSessionAuditDigestDataAddress,
3904    #endif
3905    #if (PAD_LIST || CC_NV_Read)
3906            (COMMAND_DESCRIPTOR_t *)_NV_ReadDataAddress,
3907    #endif
3908    #if (PAD_LIST || CC_NV_ReadLock)
3909            (COMMAND_DESCRIPTOR_t *)_NV_ReadLockDataAddress,
3910    #endif
3911    #if (PAD_LIST || CC_ObjectChangeAuth)
3912            (COMMAND_DESCRIPTOR_t *)_ObjectChangeAuthDataAddress,
3913    #endif
3914    #if (PAD_LIST || CC_PolicySecret)
3915            (COMMAND_DESCRIPTOR_t *)_PolicySecretDataAddress,
3916    #endif
3917    #if (PAD_LIST || CC_Rewrap)
3918            (COMMAND_DESCRIPTOR_t *)_RewrapDataAddress,
3919    #endif
3920    #if (PAD_LIST || CC_Create)
3921            (COMMAND_DESCRIPTOR_t *)_CreateDataAddress,
3922    #endif
3923    #if (PAD_LIST || CC_ECDH_ZGen)
3924            (COMMAND_DESCRIPTOR_t *)_ECDH_ZGenDataAddress,
3925    #endif
3926    #if (PAD_LIST || CC_HMAC)
3927            (COMMAND_DESCRIPTOR_t *)_HMACDataAddress,
3928    #endif
3929    #if (PAD_LIST || CC_Import)
3930            (COMMAND_DESCRIPTOR_t *)_ImportDataAddress,
3931    #endif
3932    #if (PAD_LIST || CC_Load)
3933            (COMMAND_DESCRIPTOR_t *)_LoadDataAddress,
3934    #endif
3935    #if (PAD_LIST || CC_Quote)
3936            (COMMAND_DESCRIPTOR_t *)_QuoteDataAddress,
3937    #endif
```



```
3938    #if (PAD_LIST || CC_RSA_Decrypt)
3939            (COMMAND_DESCRIPTOR_t *)_RSA_DecryptDataAddress,
3940    #endif
3941    #if (PAD_LIST)
3942            (COMMAND_DESCRIPTOR_t *)0,
3943    #endif
3944    #if (PAD_LIST || CC_HMAC_Start)
3945            (COMMAND_DESCRIPTOR_t *)_HMAC_StartDataAddress,
3946    #endif
3947    #if (PAD_LIST || CC_SequenceUpdate)
3948            (COMMAND_DESCRIPTOR_t *)_SequenceUpdateDataAddress,
3949    #endif
3950    #if (PAD_LIST || CC_Sign)
3951            (COMMAND_DESCRIPTOR_t *)_SignDataAddress,
3952    #endif
3953    #if (PAD_LIST || CC_Unseal)
3954            (COMMAND_DESCRIPTOR_t *)_UnsealDataAddress,
3955    #endif
3956    #if (PAD_LIST)
3957            (COMMAND_DESCRIPTOR_t *)0,
3958    #endif
3959    #if (PAD_LIST || CC_PolicySigned)
3960            (COMMAND_DESCRIPTOR_t *)_PolicySignedDataAddress,
3961    #endif
3962    #if (PAD_LIST || CC_ContextLoad)
3963            (COMMAND_DESCRIPTOR_t *)_ContextLoadDataAddress,
3964    #endif
3965    #if (PAD_LIST || CC_ContextSave)
3966            (COMMAND_DESCRIPTOR_t *)_ContextSaveDataAddress,
3967    #endif
3968    #if (PAD_LIST || CC_ECDH_KeyGen)
3969            (COMMAND_DESCRIPTOR_t *)_ECDH_KeyGenDataAddress,
3970    #endif
3971    #if (PAD_LIST || CC_EncryptDecrypt)
3972            (COMMAND_DESCRIPTOR_t *)_EncryptDecryptDataAddress,
3973    #endif
3974    #if (PAD_LIST || CC_FlushContext)
3975            (COMMAND_DESCRIPTOR_t *)_FlushContextDataAddress,
3976    #endif
3977    #if (PAD_LIST)
3978            (COMMAND_DESCRIPTOR_t *)0,
3979    #endif
3980    #if (PAD_LIST || CC_LoadExternal)
3981            (COMMAND_DESCRIPTOR_t *)_LoadExternalDataAddress,
3982    #endif
3983    #if (PAD_LIST || CC_MakeCredential)
3984            (COMMAND_DESCRIPTOR_t *)_MakeCredentialDataAddress,
3985    #endif
3986    #if (PAD_LIST || CC_NV_ReadPublic)
3987            (COMMAND_DESCRIPTOR_t *)_NV_ReadPublicDataAddress,
3988    #endif
3989    #if (PAD_LIST || CC_PolicyAuthorize)
3990            (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeDataAddress,
3991    #endif
3992    #if (PAD_LIST || CC_PolicyAuthValue)
3993            (COMMAND_DESCRIPTOR_t *)_PolicyAuthValueDataAddress,
3994    #endif
3995    #if (PAD_LIST || CC_PolicyCommandCode)
3996            (COMMAND_DESCRIPTOR_t *)_PolicyCommandCodeDataAddress,
3997    #endif
3998    #if (PAD_LIST || CC_PolicyCounterTimer)
3999            (COMMAND_DESCRIPTOR_t *)_PolicyCounterTimerDataAddress,
4000    #endif
4001    #if (PAD_LIST || CC_PolicyCpHash)
4002            (COMMAND_DESCRIPTOR_t *)_PolicyCpHashDataAddress,
4003    #endif
```

```
4004    #if (PAD_LIST || CC_PolicyLocality)
4005            (COMMAND_DESCRIPTOR_t *)_PolicyLocalityDataAddress,
4006    #endif
4007    #if (PAD_LIST || CC_PolicyNameHash)
4008            (COMMAND_DESCRIPTOR_t *)_PolicyNameHashDataAddress,
4009    #endif
4010    #if (PAD_LIST || CC_PolicyOR)
4011            (COMMAND_DESCRIPTOR_t *)_PolicyORDataAddress,
4012    #endif
4013    #if (PAD_LIST || CC_PolicyTicket)
4014            (COMMAND_DESCRIPTOR_t *)_PolicyTicketDataAddress,
4015    #endif
4016    #if (PAD_LIST || CC_ReadPublic)
4017            (COMMAND_DESCRIPTOR_t *)_ReadPublicDataAddress,
4018    #endif
4019    #if (PAD_LIST || CC_RSA_Encrypt)
4020            (COMMAND_DESCRIPTOR_t *)_RSA_EncryptDataAddress,
4021    #endif
4022    #if (PAD_LIST)
4023            (COMMAND_DESCRIPTOR_t *)0,
4024    #endif
4025    #if (PAD_LIST || CC_StartAuthSession)
4026            (COMMAND_DESCRIPTOR_t *)_StartAuthSessionDataAddress,
4027    #endif
4028    #if (PAD_LIST || CC_VerifySignature)
4029            (COMMAND_DESCRIPTOR_t *)_VerifySignatureDataAddress,
4030    #endif
4031    #if (PAD_LIST || CC_ECC_Parameters)
4032            (COMMAND_DESCRIPTOR_t *)_ECC_ParametersDataAddress,
4033    #endif
4034    #if (PAD_LIST || CC_FirmwareRead)
4035            (COMMAND_DESCRIPTOR_t *)_FirmwareReadDataAddress,
4036    #endif
4037    #if (PAD_LIST || CC_GetCapability)
4038            (COMMAND_DESCRIPTOR_t *)_GetCapabilityDataAddress,
4039    #endif
4040    #if (PAD_LIST || CC_GetRandom)
4041            (COMMAND_DESCRIPTOR_t *)_GetRandomDataAddress,
4042    #endif
4043    #if (PAD_LIST || CC_GetTestResult)
4044            (COMMAND_DESCRIPTOR_t *)_GetTestResultDataAddress,
4045    #endif
4046    #if (PAD_LIST || CC_Hash)
4047            (COMMAND_DESCRIPTOR_t *)_HashDataAddress,
4048    #endif
4049    #if (PAD_LIST || CC_PCR_Read)
4050            (COMMAND_DESCRIPTOR_t *)_PCR_ReadDataAddress,
4051    #endif
4052    #if (PAD_LIST || CC_PolicyPCR)
4053            (COMMAND_DESCRIPTOR_t *)_PolicyPCRDataAddress,
4054    #endif
4055    #if (PAD_LIST || CC_PolicyRestart)
4056            (COMMAND_DESCRIPTOR_t *)_PolicyRestartDataAddress,
4057    #endif
4058    #if (PAD_LIST || CC_ReadClock)
4059            (COMMAND_DESCRIPTOR_t *)_ReadClockDataAddress,
4060    #endif
4061    #if (PAD_LIST || CC_PCR_Extend)
4062            (COMMAND_DESCRIPTOR_t *)_PCR_ExtendDataAddress,
4063    #endif
4064    #if (PAD_LIST || CC_PCR_SetAuthValue)
4065            (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthValueDataAddress,
4066    #endif
4067    #if (PAD_LIST || CC_NV_Certify)
4068            (COMMAND_DESCRIPTOR_t *)_NV_CertifyDataAddress,
4069    #endif
```

```
4070    #if (PAD_LIST || CC_EventSequenceComplete)
4071            (COMMAND_DESCRIPTOR_t *)_EventSequenceCompleteDataAddress,
4072    #endif
4073    #if (PAD_LIST || CC_HashSequenceStart)
4074            (COMMAND_DESCRIPTOR_t *)_HashSequenceStartDataAddress,
4075    #endif
4076    #if (PAD_LIST || CC_PolicyPhysicalPresence)
4077            (COMMAND_DESCRIPTOR_t *)_PolicyPhysicalPresenceDataAddress,
4078    #endif
4079    #if (PAD_LIST || CC_PolicyDuplicationSelect)
4080            (COMMAND_DESCRIPTOR_t *)_PolicyDuplicationSelectDataAddress,
4081    #endif
4082    #if (PAD_LIST || CC_PolicyGetDigest)
4083            (COMMAND_DESCRIPTOR_t *)_PolicyGetDigestDataAddress,
4084    #endif
4085    #if (PAD_LIST || CC_TestParms)
4086            (COMMAND_DESCRIPTOR_t *)_TestParmsDataAddress,
4087    #endif
4088    #if (PAD_LIST || CC_Commit)
4089            (COMMAND_DESCRIPTOR_t *)_CommitDataAddress,
4090    #endif
4091    #if (PAD_LIST || CC_PolicyPassword)
4092            (COMMAND_DESCRIPTOR_t *)_PolicyPasswordDataAddress,
4093    #endif
4094    #if (PAD_LIST || CC_ZGen_2Phase)
4095            (COMMAND_DESCRIPTOR_t *)_ZGen_2PhaseDataAddress,
4096    #endif
4097    #if (PAD_LIST || CC_EC_Ephemeral)
4098            (COMMAND_DESCRIPTOR_t *)_EC_EphemeralDataAddress,
4099    #endif
4100    #if (PAD_LIST || CC_PolicyNvWritten)
4101            (COMMAND_DESCRIPTOR_t *)_PolicyNvWrittenDataAddress,
4102    #endif
4103    #if (PAD_LIST || CC_PolicyTemplate)
4104            (COMMAND_DESCRIPTOR_t *)_PolicyTemplateDataAddress,
4105    #endif
4106    #if (PAD_LIST || CC_CreateLoaded)
4107            (COMMAND_DESCRIPTOR_t *)_CreateLoadedDataAddress,
4108    #endif
4109    #if (PAD_LIST || CC_PolicyAuthorizeNV)
4110            (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeNVDataAddress,
4111    #endif
4112    #if (PAD_LIST || CC_EncryptDecrypt2)
4113            (COMMAND_DESCRIPTOR_t *)_EncryptDecrypt2DataAddress,
4114    #endif
4115    #if (PAD_LIST || CC_Vendor_TCG_Test)
4116            (COMMAND_DESCRIPTOR_t *)_Vendor_TCG_TestDataAddress,
4117    #endif
4118            0
4119    };
4120    #endif  // _COMMAND_TABLE_DISPATCH_
```


5.9 Commands.h

```
#ifndef _COMMANDS_H
#define _COMMANDS_H
```

Start-up

```
#ifdef TPM_CC_Startup
    #include "Startup_fp.h"
#endif
#ifdef TPM_CC_Shutdown
    #include "Shutdown_fp.h"
#endif
```

Testing

```
#ifdef TPM_CC_SelfTest
    #include "SelfTest_fp.h"
#endif
#ifdef TPM_CC_IncrementalSelfTest
    #include "IncrementalSelfTest_fp.h"
#endif
#ifdef TPM_CC_GetTestResult
    #include "GetTestResult_fp.h"
#endif
```

Session Commands

```
#ifdef TPM_CC_StartAuthSession
    #include "StartAuthSession_fp.h"
#endif
#ifdef TPM_CC_PolicyRestart
    #include "PolicyRestart_fp.h"
#endif
```

Object Commands

```
#ifdef TPM_CC_Create
    #include "Create_fp.h"
#endif
#ifdef TPM_CC_Load
    #include "Load_fp.h"
#endif
#ifdef TPM_CC_LoadExternal
    #include "LoadExternal_fp.h"
#endif
#ifdef TPM_CC_ReadPublic
    #include "ReadPublic_fp.h"
#endif
#ifdef TPM_CC_ActivateCredential
    #include "ActivateCredential_fp.h"
#endif
#ifdef TPM_CC_MakeCredential
    #include "MakeCredential_fp.h"
#endif
#ifdef TPM_CC_Unseal
    #include "Unseal_fp.h"
#endif
#ifdef TPM_CC_ObjectChangeAuth
    #include "ObjectChangeAuth_fp.h"
#endif
#ifdef TPM_CC_CreateLoaded
    #include "CreateLoaded_fp.h"
```

```
50  #endif
```

Duplication Commands

```
51  #ifdef TPM_CC_Duplicate
52      #include "Duplicate_fp.h"
53  #endif
54  #ifdef TPM_CC_Rewrap
55      #include "Rewrap_fp.h"
56  #endif
57  #ifdef TPM_CC_Import
58      #include "Import_fp.h"
59  #endif
```

Asymmetric Primitives

```
60  #ifdef TPM_CC_RSA_Encrypt
61      #include "RSA_Encrypt_fp.h"
62  #endif
63  #ifdef TPM_CC_RSA_Decrypt
64      #include "RSA_Decrypt_fp.h"
65  #endif
66  #ifdef TPM_CC_ECDH_KeyGen
67      #include "ECDH_KeyGen_fp.h"
68  #endif
69  #ifdef TPM_CC_ECDH_ZGen
70      #include "ECDH_ZGen_fp.h"
71  #endif
72  #ifdef TPM_CC_ECC_Parameters
73      #include "ECC_Parameters_fp.h"
74  #endif
75  #ifdef TPM_CC_ZGen_2Phase
76      #include "ZGen_2Phase_fp.h"
77  #endif
```

Symmetric Primitives

```
78  #ifdef TPM_CC_EncryptDecrypt
79      #include "EncryptDecrypt_fp.h"
80  #endif
81  #ifdef TPM_CC_EncryptDecrypt2
82      #include "EncryptDecrypt2_fp.h"
83  #endif
84  #ifdef TPM_CC_Hash
85      #include "Hash_fp.h"
86  #endif
87  #ifdef TPM_CC_HMAC
88      #include "HMAC_fp.h"
89  #endif
```

Random Number Generator

```
90  #ifdef TPM_CC_GetRandom
91      #include "GetRandom_fp.h"
92  #endif
93  #ifdef TPM_CC_StirRandom
94      #include "StirRandom_fp.h"
95  #endif
```

Hash/HMAC/Event Sequences

```
96  #ifdef TPM_CC_HMAC_Start
97      #include "HMAC_Start_fp.h"
```



```
 98  #endif
 99  #ifdef TPM_CC_HashSequenceStart
100      #include "HashSequenceStart_fp.h"
101  #endif
102  #ifdef TPM_CC_SequenceUpdate
103      #include "SequenceUpdate_fp.h"
104  #endif
105  #ifdef TPM_CC_SequenceComplete
106      #include "SequenceComplete_fp.h"
107  #endif
108  #ifdef TPM_CC_EventSequenceComplete
109      #include "EventSequenceComplete_fp.h"
110  #endif
```

Attestation Commands

```
111  #ifdef TPM_CC_Certify
112      #include "Certify_fp.h"
113  #endif
114  #ifdef TPM_CC_CertifyCreation
115      #include "CertifyCreation_fp.h"
116  #endif
117  #ifdef TPM_CC_Quote
118      #include "Quote_fp.h"
119  #endif
120  #ifdef TPM_CC_GetSessionAuditDigest
121      #include "GetSessionAuditDigest_fp.h"
122  #endif
123  #ifdef TPM_CC_GetCommandAuditDigest
124      #include "GetCommandAuditDigest_fp.h"
125  #endif
126  #ifdef TPM_CC_GetTime
127      #include "GetTime_fp.h"
128  #endif
```

Ephemeral EC Keys

```
129  #ifdef TPM_CC_Commit
130      #include "Commit_fp.h"
131  #endif
132  #ifdef TPM_CC_EC_Ephemeral
133      #include "EC_Ephemeral_fp.h"
134  #endif
```

Signing and Signature Verification

```
135  #ifdef TPM_CC_VerifySignature
136      #include "VerifySignature_fp.h"
137  #endif
138  #ifdef TPM_CC_Sign
139      #include "Sign_fp.h"
140  #endif
```

Command Audit

```
141  #ifdef TPM_CC_SetCommandCodeAuditStatus
142      #include "SetCommandCodeAuditStatus_fp.h"
143  #endif
```

Integrity Collection (PCR)

```
144  #ifdef TPM_CC_PCR_Extend
145      #include "PCR_Extend_fp.h"
```



```
#endif
#ifdef TPM_CC_PCR_Event
    #include "PCR_Event_fp.h"
#endif
#ifdef TPM_CC_PCR_Read
    #include "PCR_Read_fp.h"
#endif
#ifdef TPM_CC_PCR_Allocate
    #include "PCR_Allocate_fp.h"
#endif
#ifdef TPM_CC_PCR_SetAuthPolicy
    #include "PCR_SetAuthPolicy_fp.h"
#endif
#ifdef TPM_CC_PCR_SetAuthValue
    #include "PCR_SetAuthValue_fp.h"
#endif
#ifdef TPM_CC_PCR_Reset
    #include "PCR_Reset_fp.h"
#endif
```

Enhanced Authorization (EA) Commands

```
#ifdef TPM_CC_PolicySigned
    #include "PolicySigned_fp.h"
#endif
#ifdef TPM_CC_PolicySecret
    #include "PolicySecret_fp.h"
#endif
#ifdef TPM_CC_PolicyTicket
    #include "PolicyTicket_fp.h"
#endif
#ifdef TPM_CC_PolicyOR
    #include "PolicyOR_fp.h"
#endif
#ifdef TPM_CC_PolicyPCR
    #include "PolicyPCR_fp.h"
#endif
#ifdef TPM_CC_PolicyLocality
    #include "PolicyLocality_fp.h"
#endif
#ifdef TPM_CC_PolicyNV
    #include "PolicyNV_fp.h"
#endif
#ifdef TPM_CC_PolicyCounterTimer
    #include "PolicyCounterTimer_fp.h"
#endif
#ifdef TPM_CC_PolicyCommandCode
    #include "PolicyCommandCode_fp.h"
#endif
#ifdef TPM_CC_PolicyPhysicalPresence
    #include "PolicyPhysicalPresence_fp.h"
#endif
#ifdef TPM_CC_PolicyCpHash
    #include "PolicyCpHash_fp.h"
#endif
#ifdef TPM_CC_PolicyNameHash
    #include "PolicyNameHash_fp.h"
#endif
#ifdef TPM_CC_PolicyDuplicationSelect
    #include "PolicyDuplicationSelect_fp.h"
#endif
#ifdef TPM_CC_PolicyAuthorize
    #include "PolicyAuthorize_fp.h"
#endif
#ifdef TPM_CC_PolicyAuthValue
```

```
208         #include "PolicyAuthValue_fp.h"
209     #endif
210     #ifdef TPM_CC_PolicyPassword
211         #include "PolicyPassword_fp.h"
212     #endif
213     #ifdef TPM_CC_PolicyGetDigest
214         #include "PolicyGetDigest_fp.h"
215     #endif
216     #ifdef TPM_CC_PolicyNvWritten
217         #include "PolicyNvWritten_fp.h"
218     #endif
219     #ifdef TPM_CC_PolicyTemplate
220         #include "PolicyTemplate_fp.h"
221     #endif
222     #ifdef TPM_CC_PolicyAuthorizeNV
223         #include "PolicyAuthorizeNV_fp.h"
224     #endif
```

Hierarchy Commands

```
225     #ifdef TPM_CC_CreatePrimary
226         #include "CreatePrimary_fp.h"
227     #endif
228     #ifdef TPM_CC_HierarchyControl
229         #include "HierarchyControl_fp.h"
230     #endif
231     #ifdef TPM_CC_SetPrimaryPolicy
232         #include "SetPrimaryPolicy_fp.h"
233     #endif
234     #ifdef TPM_CC_ChangePPS
235         #include "ChangePPS_fp.h"
236     #endif
237     #ifdef TPM_CC_ChangeEPS
238         #include "ChangeEPS_fp.h"
239     #endif
240     #ifdef TPM_CC_Clear
241         #include "Clear_fp.h"
242     #endif
243     #ifdef TPM_CC_ClearControl
244         #include "ClearControl_fp.h"
245     #endif
246     #ifdef TPM_CC_HierarchyChangeAuth
247         #include "HierarchyChangeAuth_fp.h"
248     #endif
```

Dictionary Attack Functions

```
249     #ifdef TPM_CC_DictionaryAttackLockReset
250         #include "DictionaryAttackLockReset_fp.h"
251     #endif
252     #ifdef TPM_CC_DictionaryAttackParameters
253         #include "DictionaryAttackParameters_fp.h"
254     #endif
```

Miscellaneous Management Functions

```
255     #ifdef TPM_CC_PP_Commands
256         #include "PP_Commands_fp.h"
257     #endif
258     #ifdef TPM_CC_SetAlgorithmSet
259         #include "SetAlgorithmSet_fp.h"
260     #endif
```

Field Upgrade

```
261 #ifdef TPM_CC_FieldUpgradeStart
262     #include "FieldUpgradeStart_fp.h"
263 #endif
264 #ifdef TPM_CC_FieldUpgradeData
265     #include "FieldUpgradeData_fp.h"
266 #endif
267 #ifdef TPM_CC_FirmwareRead
268     #include "FirmwareRead_fp.h"
269 #endif
```

Context Management

```
270 #ifdef TPM_CC_ContextSave
271     #include "ContextSave_fp.h"
272 #endif
273 #ifdef TPM_CC_ContextLoad
274     #include "ContextLoad_fp.h"
275 #endif
276 #ifdef TPM_CC_FlushContext
277     #include "FlushContext_fp.h"
278 #endif
279 #ifdef TPM_CC_EvictControl
280     #include "EvictControl_fp.h"
281 #endif
```

Clocks and Timers

```
282 #ifdef TPM_CC_ReadClock
283     #include "ReadClock_fp.h"
284 #endif
285 #ifdef TPM_CC_ClockSet
286     #include "ClockSet_fp.h"
287 #endif
288 #ifdef TPM_CC_ClockRateAdjust
289     #include "ClockRateAdjust_fp.h"
290 #endif
```

Capability Commands

```
291 #ifdef TPM_CC_GetCapability
292     #include "GetCapability_fp.h"
293 #endif
294 #ifdef TPM_CC_TestParms
295     #include "TestParms_fp.h"
296 #endif
```

Non-volatile Storage

```
297 #ifdef TPM_CC_NV_DefineSpace
298     #include "NV_DefineSpace_fp.h"
299 #endif
300 #ifdef TPM_CC_NV_UndefineSpace
301     #include "NV_UndefineSpace_fp.h"
302 #endif
303 #ifdef TPM_CC_NV_UndefineSpaceSpecial
304     #include "NV_UndefineSpaceSpecial_fp.h"
305 #endif
306 #ifdef TPM_CC_NV_ReadPublic
307     #include "NV_ReadPublic_fp.h"
308 #endif
309 #ifdef TPM_CC_NV_Write
310     #include "NV_Write_fp.h"
311 #endif
312 #ifdef TPM_CC_NV_Increment
```

```
313         #include "NV_Increment_fp.h"
314     #endif
315     #ifdef TPM_CC_NV_Extend
316         #include "NV_Extend_fp.h"
317     #endif
318     #ifdef TPM_CC_NV_SetBits
319         #include "NV_SetBits_fp.h"
320     #endif
321     #ifdef TPM_CC_NV_WriteLock
322         #include "NV_WriteLock_fp.h"
323     #endif
324     #ifdef TPM_CC_NV_GlobalWriteLock
325         #include "NV_GlobalWriteLock_fp.h"
326     #endif
327     #ifdef TPM_CC_NV_Read
328         #include "NV_Read_fp.h"
329     #endif
330     #ifdef TPM_CC_NV_ReadLock
331         #include "NV_ReadLock_fp.h"
332     #endif
333     #ifdef TPM_CC_NV_ChangeAuth
334         #include "NV_ChangeAuth_fp.h"
335     #endif
336     #ifdef TPM_CC_NV_Certify
337         #include "NV_Certify_fp.h"
338     #endif
```

Vendor Specific

```
339     #ifdef TPM_CC_Vendor_TCG_Test
340         #include "Vendor_TCG_Test_fp.h"
341     #endif
342     #endif
```


5.10 CompilerDependencies.h

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

```
#ifndef _COMPILER_DEPENDENCIES_H_
#define _COMPILER_DEPENDENCIES_H_
#ifdef GCC
#   undef _MSC_VER
#   undef WIN32
#endif
```

If in-line functions are not being used, define INLINE as null. If INLINE_FUNCTIONS is defined, then need to define INLINE for each compiler.

```
#ifndef INLINE_FUNCTIONS
#   define INLINE
#endif
#ifdef _MSC_VER
```

These definitions are for the Microsoft compiler Endian conversion for aligned structures

```
#   define REVERSE_ENDIAN_16(_Number) _byteswap_ushort(_Number)
#   define REVERSE_ENDIAN_32(_Number) _byteswap_ulong(_Number)
#   define REVERSE_ENDIAN_64(_Number) _byteswap_uint64(_Number)
```

Handling of INLINE macro

```
#   ifdef INLINE_FUNCTIONS
#    define INLINE   static __inline
#   endif
```

Avoid compiler warning for in line of stdio (or not)

```
//#define _NO_CRT_STDIO_INLINE
```

This macro is used to handle LIB_EXPORT of function and variable names in lieu of a .def file. Visual Studio requires that functions be explicitly exported and imported.

```
#   define LIB_EXPORT __declspec(dllexport) // VS compatible version
#   define LIB_IMPORT __declspec(dllimport)
```

This is defined to indicate a function that does not return. Microsoft compilers do not support the _Noretrun() function parameter.

```
#   define NORETURN  __declspec(noreturn)
#   if _MSC_VER >= 1400     // SAL processing when needed
#       include <sal.h>
#   endif
#   ifdef _WIN64
#       define _INTPTR 2
#    else
#       define _INTPTR 1
#    endif
#define NOT_REFERENCED(x)   (x)
```

Lower the compiler error warning for system include files. They tend not to be that clean and there is no reason to sort through all the spurious errors that they generate when the normal error level is set to /Wall

```
#   define _REDUCE_WARNING_LEVEL_(n)                   \
```

```
31  __pragma(warning(push, n))
```

Restore the compiler warning level

```
32  #   define _NORMAL_WARNING_LEVEL_                          \
33  __pragma(warning(pop))
34  #   include <stdint.h>
35  #endif
36  #ifndef _MSC_VER
37  #   define WINAPI
38  #   define __pragama(x)
39  #   define REVERSE_ENDIAN_16(_Number) __builtin_bswap16(_Number)
40  #   define REVERSE_ENDIAN_32(_Number) __builtin_bswap32(_Number)
41  #   define REVERSE_ENDIAN_64(_Number) __builtin_bswap64(_Number)
42  #   ifdef INLINE_FUNCTIONS
43  #   define INLINE static inline
44  #endif
45  #if defined(__GNUC__)
46  #   define NORETURN                    __attribute__((noreturn))
47  #   include <stdint.h>
48  #  else
49  #   define NORETURN
50  #  endif
51  #   define LIB_EXPORT
52  #   define LIB_IMPORT
53  #   define _REDUCE_WARNING_LEVEL_
54  #   define _NORMAL_WARNING_LEVEL_
55  #   define  NOT_REFERENCED(x) (x = x)
56  #endif
57  #ifdef _POSIX_
58  typedef int SOCKET;
59  #endif
60  #endif // _COMPILER_DEPENDENCIES_H_
```

5.11 Global.h

```
#if !defined _TPM_H_
#error "Should not be called"
#endif
```

5.11.1 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.

5.11.2 Includes

```
#ifndef         GLOBAL_H
#define         GLOBAL_H
//#define SELF_TEST
_REDUCE_WARNING_LEVEL_(2)
#include <string.h>
//#include <setjmp.h>
#include <stddef.h>
_NORMAL_WARNING_LEVEL_
#ifdef SIMULATION
#undef CONTEXT_SLOT
#  define CONTEXT_SLOT    UINT8
#endif
#include "Capabilities.h"
#include "TpmTypes.h"
#include "CommandAttributes.h"
#include "CryptTest.h"
#include "BnValues.h"
#include "CryptHash.h"
#include "CryptRand.h"
#include "CryptEcc.h"
#include "CryptRsa.h"
#include "CryptTest.h"
#include "TpmError.h"
#include "NV.h"
//** Defines and Types
//*** Crypto Self-Test Values
extern ALGORITHM_VECTOR     g_implementedAlgorithms;
extern ALGORITHM_VECTOR     g_toTest;
//*** Size Types
// These types are used to differentiate the two different size values used.
//
// NUMBYTES is used when a size is a number of bytes (usually a TPM2B)
typedef UINT16  NUMBYTES;
//*** Other Types
// An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)
typedef BYTE    AUTH_VALUE[sizeof(TPMU_HA)];
```

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

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

A NAME is a BYTE array that can contain a TPMU_NAME

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

A CLOCK_NONCE is used to tag the time value in the authorization session and in the ticket computation so that the ticket expires when there is a time discontinuity. When the clock stops during normal operation, the nonce is 64-bit value kept in RAM but it is a 32-bit counter when the clock only stops during power events.

```
42  #ifdef CLOCK_STOPS
43  typedef UINT64          CLOCK_NONCE;
44  #else
45  typedef UINT32          CLOCK_NONCE;
46  #endif
```

5.11.3 Loaded Object Structures

5.11.3.1 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.

5.11.3.2 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.

```
47  typedef struct
48  {
49      unsigned            publicOnly : 1;     //0) SET if only the public portion of
50                                              //   an object is loaded
51      unsigned            epsHierarchy : 1;   //1) SET if the object belongs to EPS
52                                              //   Hierarchy
53      unsigned            ppsHierarchy : 1;   //2) SET if the object belongs to PPS
54                                              //   Hierarchy
55      unsigned            spsHierarchy : 1;   //3) SET f the object belongs to SPS
56                                              //   Hierarchy
57      unsigned            evict : 1;          //4) SET if the object is a platform or
58                                              //   owner evict object.  Platform-
59                                              //   evict object belongs to PPS
60                                              //   hierarchy, owner-evict object
61                                              //   belongs to SPS or EPS hierarchy.
62                                              //   This bit is also used to mark a
63                                              //   completed sequence object so it
64                                              //   will be flush when the
65                                              //   SequenceComplete command succeeds.
66      unsigned            primary : 1;        //5) SET for a primary object
67      unsigned            temporary : 1;      //6) SET for a temporary object
68      unsigned            stClear : 1;        //7) SET for an stClear object
69      unsigned            hmacSeq : 1;        //8) SET for an HMAC sequence object
70      unsigned            hashSeq : 1;        //9) SET for a hash sequence object
71      unsigned            eventSeq : 1;       //10) SET for an event sequence object
72      unsigned            ticketSafe : 1;     //11) SET if a ticket is safe to create
73                                              //    for hash sequence object
74      unsigned            firstBlock : 1;     //12) SET if the first block of hash
75                                              //    data has been received.  It
76                                              //    works with ticketSafe bit
77      unsigned            isParent : 1;       //13) SET if the key has the proper
```

```
78                                          //    attributes to be a parent key
79      unsigned            privateExp : 1;     //14) SET when the private exponent
80                                          //    of an RSA key has been validated.
81      unsigned            occupied : 1;       //15) SET when the slot is occupied.
82      unsigned            derivation : 1;     //16) SET when the key is a derivation
83                                          //        parent
84      unsigned            external : 1;       //17) SET when the object is loaded with
85                                          //    TPM2_LoadExternal();
86  } OBJECT_ATTRIBUTES;
```

5.11.3.3 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.

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

5.11.3.4 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.

NOTE: In a future version, this will probably be renamed as SEQUENCE_OBJECT

```
105 typedef struct HASH_OBJECT
106 {
107     OBJECT_ATTRIBUTES   attributes;         // The attributes of the HASH object
108     TPMI_ALG_PUBLIC     type;               // algorithm
109     TPMI_ALG_HASH       nameAlg;            // name algorithm
110     TPMA_OBJECT         objectAttributes;   // object attributes
111     // The data below is unique to a sequence object
112     TPM2B_AUTH          auth;               // authorization for use of sequence
113     union
114     {
115         HASH_STATE      hashState[HASH_COUNT];
116         HMAC_STATE      hmacState;
117     }                   state;
118 } HASH_OBJECT;
```

```
119   typedef BYTE  HASH_OBJECT_BUFFER[sizeof(HASH_OBJECT)];
```

5.11.3.5 ANY_OBJECT

This is the union for holding either a sequence object or a regular object. for ContextSave() and ContextLoad()

```
120   typedef union ANY_OBJECT
121   {
122       OBJECT              entity;
123       HASH_OBJECT         hash;
124   } ANY_OBJECT;
125   typedef BYTE    ANY_OBJECT_BUFFER[sizeof(ANY_OBJECT)];
```

5.11.4 AUTH_DUP Types

These values are used in the authorization processing.

```
126   typedef UINT32          AUTH_ROLE;
127   #define AUTH_NONE       ((AUTH_ROLE)(0))
128   #define AUTH_USER       ((AUTH_ROLE)(1))
129   #define AUTH_ADMIN      ((AUTH_ROLE)(2))
130   #define AUTH_DUP        ((AUTH_ROLE)(3))
```

5.11.5 Active Session Context

5.11.5.1 Description

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

5.11.5.2 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.

```
131   typedef struct SESSION_ATTRIBUTES
132   {
133       unsigned    isPolicy : 1;           //1) SET if the session may only be used
134                                           //   for policy
135       unsigned    isAudit : 1;            //2) SET if the session is used for audit
136       unsigned    isBound : 1;            //3) SET if the session is bound to with an
137                                           //   entity. This attribute will be CLEAR
138                                           //   if either isPolicy or isAudit is SET.
139       unsigned    isCpHashDefined : 1;    //3) SET if the cpHash has been defined
140                                           //   This attribute is not SET unless
141                                           //   'isPolicy' is SET.
142       unsigned    isAuthValueNeeded : 1;  //5) SET if the authValue is required for
143                                           //   computing the session HMAC. This
144                                           //   attribute is not SET unless 'isPolicy'
145                                           //   is SET.
146       unsigned    isPasswordNeeded : 1;   //6) SET if a password authValue is required
147                                           //   for authorization This attribute is not
148                                           //   SET unless 'isPolicy' is SET.
149       unsigned    isPPRequired : 1;       //7) SET if physical presence is required to
150                                           //   be asserted when the authorization is
151                                           //   checked. This attribute is not SET
152                                           //   unless 'isPolicy' is SET.
153       unsigned    isTrialPolicy : 1;      //8) SET if the policy session is created
```

```
154                                         //   for trial of the policy's policyHash
155                                         //   generation. This attribute is not SET
156                                         //   unless 'isPolicy' is SET.
157     unsigned    isDaBound : 1;          //9) SET if the bind entity had noDA CLEAR.
158                                         //   If this is SET, then an authorization
159                                         //   failure using this session will count
160                                         //   against lockout even if the object
161                                         //   being authorized is exempt from DA.
162     unsigned    isLockoutBound : 1;     //10) SET if the session is bound to
163                                         //    lockoutAuth.
164     unsigned    includeAuth : 1;        //11) This attribute is SET when the
165                                         //    authValue of an object is to be
166                                         //    included in the computation of the
167                                         //    HMAC key for the command and response
168                                         //    computations. (was 'requestWasBound')
169     unsigned    checkNvWritten : 1;     //12) SET if the TPMA_NV_WRITTEN attribute
170                                         //    needs to be checked when the policy is
171                                         //    used for authorization for NV access.
172                                         //    If this is SET for any other type, the
173                                         //    policy will fail.
174     unsigned    nvWrittenState : 1;     //13) SET if TPMA_NV_WRITTEN is required to
175                                         //    be SET. Used when 'checkNvWritten' is
176                                         //    SET
177     unsigned    isTemplateSet : 1;      //14) SET if the templateHash needs to be
178                                         //    checked for Create, CreatePrimary, or
179                                         //    CreateLoaded.
180 } SESSION_ATTRIBUTES;
```

5.11.5.3 SESSION Structure

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

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

```
181 typedef struct SESSION
182 {
183     SESSION_ATTRIBUTES  attributes;         // session attributes
184     UINT32              pcrCounter;         // PCR counter value when PCR is
185                                             // included (policy session)
186                                             // If no PCR is included, this
187                                             // value is 0.
188     UINT64              startTime;          // The value in g_time when the session
189                                             // was started (policy session)
190     UINT64              timeout;            // The timeout relative to g_time
191                                             // There is no timeout if this value
192                                             // is 0.
193     CLOCK_NONCE         epoch;              // The g_clockEpoch value when the
194                                             // session was started. If g_clockEpoch
195                                             // does not match this value when the
196                                             // timeout is used, then
197                                             // then the command will fail.
198     TPM_CC              commandCode;        // command code (policy session)
199     TPM_ALG_ID          authHashAlg;        // session hash algorithm
200     TPMA_LOCALITY       commandLocality;    // command locality (policy session)
201     TPMT_SYM_DEF        symmetric;          // session symmetric algorithm (if any)
202     TPM2B_AUTH          sessionKey;         // session secret value used for
203                                             // this session
204     TPM2B_NONCE         nonceTPM;           // last TPM-generated nonce for
205                                             // generating HMAC and encryption keys
206    union
207     {
208         TPM2B_NAME      boundEntity;        // value used to track the entity to
209                                             // which the session is bound
210         TPM2B_DIGEST    cpHash;             // the required cpHash value for the
```

```
                                            // command being authorized
        TPM2B_DIGEST    nameHash;           // the required nameHash
        TPM2B_DIGEST    templateHash;       // the required template for creation
    } u1;
    union
    {
        TPM2B_DIGEST    auditDigest;        // audit session digest
        TPM2B_DIGEST    policyDigest;       // policyHash
    } u2;                                   // audit log and policyHash may
                                            // share space to save memory
} SESSION;
#define     EXPIRES_ON_RESET    INT32_MIN
#define     TIMEOUT_ON_RESET    UINT64_MAX
#define     EXPIRES_ON_RESTART  (INT32_MIN + 1)
#define     TIMEOUT_ON_RESTART  (UINT64_MAX - 1)
typedef BYTE        SESSION_BUF[sizeof(SESSION)];
```

5.11.6 PCR

5.11.6.1 PCR_SAVE Structure

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

```
typedef struct PCR_SAVE
{
#ifdef TPM_ALG_SHA1
    BYTE                sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA256
    BYTE                sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA384
    BYTE                sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA512
    BYTE                sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SM3_256
    BYTE                sm3_256[NUM_STATIC_PCR][SM3_256_DIGEST_SIZE];
#endif
    // This counter increments whenever the PCR are updated.
    // NOTE: A platform-specific specification may designate
    //       certain PCR changes as not causing this counter
    //       to increment.
    UINT32              pcrCounter;
} PCR_SAVE;
```

5.11.6.2 PCR_POLICY

```
#if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
```

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

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

```
256   } PCR_POLICY;
257   #endif
```

5.11.6.3 PCR_AUTHVALUE

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

```
258   typedef struct PCR_AUTH_VALUE
259   {
260       TPM2B_DIGEST        auth[NUM_AUTHVALUE_PCR_GROUP];
261   } PCR_AUTHVALUE;
```

5.11.7 Startup

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

```
262   typedef enum
263   {
264       SU_RESET,
265       SU_RESTART,
266       SU_RESUME
267   } STARTUP_TYPE;
```

5.11.8 NV

5.11.8.1 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.

```
268   typedef struct NV_INDEX
269   {
270       TPMS_NV_PUBLIC      publicArea;
271       TPM2B_AUTH          authValue;
272   } NV_INDEX;
```

5.11.8.2 NV_REF

An NV_REF is an opaque value returned by the NV subsystem. It is used to reference and NV Index in a relatively efficient way. Rather than having to continually search for an Index, its reference value may be used. In this implementation, an NV_REF is a byte pointer that points to the copy of the NV memory that is kept in RAM.

```
273   typedef UINT32           NV_REF;
274   typedef BYTE            *NV_RAM_REF;
```

5.11.8.3 NV_PIN

This structure deals with the possible endianess differences between the canonical form of the TPMS_NV_PIN_COUNTER_PARAMETERS structure and the internal value. The structures allow the data in a PIN index to be read as an 8-octet value using NvReadUINT64Data(). That function will byte swap all the values on a little endian system. This will put the bytes with the 4-octet values in the correct order but will swap the *pinLimit* and *pinCount* values. When written, the PIN index is simply handled as a normal index with the octets in canonical order.

```
275  #if BIG_ENDIAN_TPM == YES
276  typedef struct
277  {
278      UINT32      pinCount;
279      UINT32      pinLimit;
280  } PIN_DATA;
281  #else
282  typedef struct
283  {
284      UINT32      pinLimit;
285      UINT32      pinCount;
286  } PIN_DATA;
287  #endif
288  typedef union
289  {
290      UINT64     intVal;
291      PIN_DATA   pin;
292  } NV_PIN;
```

5.11.9 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.

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

5.11.10 RAM Global Values

5.11.10.1 Description

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

5.11.10.2 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.

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

5.11.10.3 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.

```
297  extern TPM_HANDLE       g_exclusiveAuditSession;
```

5.11.10.4 g_time

This is the value in which we keep the current command time. This is initialized at the start of each command. The time is in *mS*.

```
298  extern  UINT64          g_time;
```


5.11.10.5 g_timeEpoch

This value contains the current clock Epoch. It changes when there is a clock discontinuity. It may be necessary to place this in NV should the timer be able to run across a power down of the TPM but not in all cases (e.g. dead battery). If the nonce is placed in NV, it should go in gp because it should be changing slowly.

```
299    #ifdef CLOCK_STOPS
300    extern CLOCK_NONCE       g_timeEpoch;
301    #else
302    #define g_timeEpoch      gp.timeEpoch
303    #endif
```

5.11.10.6 g_timeNewEpochNeeded

This flag is SET at startup if a new timer nonce is needed. This flag will cause a new *g_timeEpoch* to be generated if it is needed by any of the ticket functions.

```
304    extern BOOL              g_timeNewEpochNeeded;
```

5.11.10.7 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.

```
305    extern BOOL              g_phEnable;
```

5.11.10.8 g_pcrReConfig

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).

```
306    extern BOOL              g_pcrReConfig;
```

5.11.10.9 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().

```
307    extern TPMI_DH_OBJECT    g_DRTMHandle;
```

5.11.10.10 g_DrtmPreStartup

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

```
308    extern  BOOL             g_DrtmPreStartup;
```

5.11.10.11 g_StartupLocality3

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


```
309   extern  BOOL            g_StartupLocality3;
```

5.11.10.12 TPM_SU_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.

```
310   #define SU_NONE_VALUE           (0xFFFF)
311   #define TPM_SU_NONE             (TPM_SU)(SU_NONE_VALUE)
```

5.11.10.13 TPM_SU_DA_USED

As with TPM_SU_NONE, this value is added to allow indication that the shutdown was not orderly and that a DA=protected object was reference during the previous cycle.

```
312   #define SU_DA_USED_VALUE    (SU_NONE_VALUE - 1)
313   #define TPM_SU_DA_USED      (TPM_SU)(SU_DA_USED_VALUE)
```

5.11.10.14 Startup Flags

These flags are included in gp.*orderlyState*. These are hacks and are being used to avoid having to change the layout of gp. The PRE_STARTUP_FLAG indicates that a _TPM_Hash_Start()/_Data()/_End() sequence was received after _TPM_Init() but before TPM2_StartUp(). STARTUP_LOCALITY_3 indicates that the last TPM2_Startup() was received at locality 3. These flags are only relevant if after a TPM2_Shutdown(STATE).

```
314   #define PRE_STARTUP_FLAG    0x8000
315   #define STARTUP_LOCALITY_3   0x4000
316   #ifdef USE_DA_USED
```

5.11.10.15 g_daUsed

This location indicates if a DA-protected value is accessed during a boot cycle. If none has, then there is no need to increment *failedTries* on the next non-orderly startup. This bit is merged with gp.*orderlyState* when that gp.orderly is set to SU_NONE_VALUE

```
317   extern BOOL               g_daUsed;
318   #endif
```

5.11.10.16 g_updateNV

This flag indicates if NV should be updated at the end of a command. This flag is set to UT_NONE 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 not UT_NONE, any pending NV writes will be committed to NV. UT_ORDERLY causes any RAM data to be written to the orderly space for staging the write to NV.

```
319   typedef BYTE        UPDATE_TYPE;
320   #define UT_NONE     (UPDATE_TYPE)0
321   #define UT_NV       (UPDATE_TYPE)1
322   #define UT_ORDERLY  (UPDATE_TYPE)(UT_NV + 2)
323   extern UPDATE_TYPE          g_updateNV;
```

5.11.10.17 g_powerWasLost

This flag is used to indicate if the power was lost. It is SET in _TPM__Init(). This flag is cleared by TPM2_Startup() after all power-lost activities are completed.

NOTE: When power is applied, this value can come up as anything. However, _plat__WasPowerLost() will provide the proper indication in that case. So, when power is actually lost, we get the correct answer. When power was not lost, but the power-lost processing has not been completed before the next _TPM_Init(), then the TPM still does the correct thing.

```
324 extern BOOL             g_powerWasLost;
```

5.11.10.18 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 SU_NONE_VALUE, then the orderly state in NV memory will be changed to SU_NONE_VALUE or SU_DA_USED_VALUE.

```
325 extern BOOL             g_clearOrderly;
```

5.11.10.19 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.

```
326 extern TPM_SU           g_prevOrderlyState;
```

5.11.10.20 g_nvOk

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

```
327 extern BOOL             g_nvOk;
```

NV availability is sampled as the start of each command and stored here so that its value remains consistent during the command execution

```
328 extern TPM_RC           g_NvStatus;
```

5.11.10.21 g_platformUnique

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

```
329 extern TPM2B_AUTH       g_platformUniqueAuthorities; // Reserved for RNG
330 extern TPM2B_AUTH       g_platformUniqueDetails;   // referenced by VENDOR_PERMANENT
```

5.11.11 Persistent Global Values

5.11.11.1 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.

5.11.11.2 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().

```
331  typedef struct
332  {
333  //*****************************************************************************
334  //          Hierarchy
335  //*****************************************************************************
336  // The values in this section are related to the hierarchies.
337      BOOL                disableClear;       // TRUE if TPM2_Clear() using
338                                              // lockoutAuth is disabled
339      // Hierarchy authPolicies
340      TPMI_ALG_HASH       ownerAlg;
341      TPMI_ALG_HASH       endorsementAlg;
342      TPMI_ALG_HASH       lockoutAlg;
343      TPM2B_DIGEST        ownerPolicy;
344      TPM2B_DIGEST        endorsementPolicy;
345      TPM2B_DIGEST        lockoutPolicy;
346      // Hierarchy authValues
347      TPM2B_AUTH          ownerAuth;
348      TPM2B_AUTH          endorsementAuth;
349      TPM2B_AUTH          lockoutAuth;
350      // Primary Seeds
351      TPM2B_SEED          EPSeed;
352      TPM2B_SEED          SPSeed;
353      TPM2B_SEED          PPSeed;
354      // Note there is a nullSeed in the state_reset memory.
355      // Hierarchy proofs
356      TPM2B_AUTH          phProof;
357      TPM2B_AUTH          shProof;
358      TPM2B_AUTH          ehProof;
359      // Note there is a nullProof in the state_reset memory.
360  //*****************************************************************************
361  //          Reset Events
362  //*****************************************************************************
363  // A count that increments at each TPM reset and never get reset during the life
364  // time of TPM.  The value of this counter is initialized to 1 during TPM
365  // manufacture process. It is used to invalidate all saved contexts after a TPM
366  // Reset.
367      UINT64              totalResetCount;
368  // This counter increments on each TPM Reset. The counter is reset by
369  // TPM2_Clear().
370      UINT32              resetCount;
371  //*****************************************************************************
372  //          PCR
373  //*****************************************************************************
374  // This structure hold the policies for those PCR that have an update policy.
375  // This implementation only supports a single group of PCR controlled by
376  // policy. If more are required, then this structure would be changed to
377  // an array.
378  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
379      PCR_POLICY          pcrPolicies;
380  #endif
381  // This structure indicates the allocation of PCR. The structure contains a
```

```
382   // list of PCR allocations for each implemented algorithm. If no PCR are
383   // allocated for an algorithm, a list entry still exists but the bit map
384   // will contain no SET bits.
385       TPML_PCR_SELECTION  pcrAllocated;
386   //*****************************************************************************
387   //          Physical Presence
388   //*****************************************************************************
389   // The PP_LIST type contains a bit map of the commands that require physical
390   // to be asserted when the authorization is evaluated. Physical presence will be
391   // checked if the corresponding bit in the array is SET and if the authorization
392   // handle is TPM_RH_PLATFORM.
393   //
394   // These bits may be changed with TPM2_PP_Commands().
395       BYTE                ppList[(COMMAND_COUNT + 7) / 8];
396   //*****************************************************************************
397   //          Dictionary attack values
398   //*****************************************************************************
399   // These values are used for dictionary attack tracking and control.
400       UINT32              failedTries;        // the current count of unexpired
401                                               // authorization failures
402       UINT32              maxTries;           // number of unexpired authorization
403                                               // failures before the TPM is in
404                                               // lockout
405       UINT32              recoveryTime;       // time between authorization failures
406                                               // before failedTries is decremented
407       UINT32              lockoutRecovery;    // time that must expire between
408                                               // authorization failures associated
409                                               // with lockoutAuth
410       BOOL                lockOutAuthEnabled; // TRUE if use of lockoutAuth is
411                                               // allowed
412   //*****************************************************************************
413   //           Orderly State
414   //*****************************************************************************
415   // The orderly state for current cycle
416       TPM_SU              orderlyState;
417   //*****************************************************************************
418   //           Command audit values.
419   //*****************************************************************************
420       BYTE                auditCommands[((COMMAND_COUNT + 1) + 7) / 8];
421       TPMI_ALG_HASH       auditHashAlg;
422       UINT64              auditCounter;
423   //*****************************************************************************
424   //           Algorithm selection
425   //*****************************************************************************
426   //
427   // The 'algorithmSet' value indicates the collection of algorithms that are
428   // currently in used on the TPM.  The interpretation of value is vendor dependent.
429       UINT32              algorithmSet;
430   //*****************************************************************************
431   //           Firmware version
432   //*****************************************************************************
433   // The firmwareV1 and firmwareV2 values are instanced in TimeStamp.c. This is
434   // a scheme used in development to allow determination of the linker build time
435   // of the TPM. An actual implementation would implement these values in a way that
436   // is consistent with vendor needs. The values are maintained in RAM for simplified
437   // access with a master version in NV.  These values are modified in a
438   // vendor-specific way.
439   // g_firmwareV1 contains the more significant 32-bits of the vendor version number.
440   // In the reference implementation, if this value is printed as a hex
441   // value, it will have the format of YYYYMMDD
442       UINT32              firmwareV1;
443   // g_firmwareV1 contains the less significant 32-bits of the vendor version number.
444   // In the reference implementation, if this value is printed as a hex
445   // value, it will have the format of 00 HH MM SS
446       UINT32              firmwareV2;
447   //*****************************************************************************
```

```
448   //           Timer Epoch
449   //***************************************************************************
450   // timeEpoch contains a nonce that has a vendor=specific size (should not be
451   // less than 8 bytes. This nonce changes when the clock epoch changes. The clock
452   // epoch changes when there is a discontinuity in the timing of the TPM.
453   #ifndef CLOCK_STOPS
454       CLOCK_NONCE         timeEpoch;
455   #endif
456   } PERSISTENT_DATA;
457   extern PERSISTENT_DATA  gp;
```

5.11.11.3 ORDERLY_DATA

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

```
458   typedef struct orderly_data
459   {
460   //***************************************************************************
461   //           TIME
462   //***************************************************************************
463   // Clock has two parts. One is the state save part and one is the NV part. The
464   // state save version is updated on each command. When the clock rolls over, the
465   // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
466   // orderly way, then the sClock value is used to initialize the clock. If the
467   // TPM shutdown was not orderly, then the persistent value is used and the safe
468   // attribute is clear.
469       UINT64              clock;              // The orderly version of clock
470       TPMI_YES_NO         clockSafe;          // Indicates if the clock value is
471                                               // safe.
472       // In many implementations, the quality of the entropy available is not that
473       // high. To compensate, the current value of the drbgState can be saved and
474       // restored on each power cycle. This prevents the internal state from reverting
475       // to the initial state on each power cycle and starting with a limited amount
476       // of entropy. By keeping the old state and adding entropy, the entropy will
477       // accumulate.
478       DRBG_STATE          drbgState;
479   } ORDERLY_DATA;
480   #  define drbgDefault go.drbgState
481   extern ORDERLY_DATA     go;
```

5.11.11.4 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).

```
482   typedef struct state_clear_data
483   {
484   //***************************************************************************
485   //           Hierarchy Control
486   //***************************************************************************
487       BOOL                shEnable;           // default reset is SET
488       BOOL                ehEnable;           // default reset is SET
489       BOOL                phEnableNV;         // default reset is SET
490       TPMI_ALG_HASH       platformAlg;        // default reset is TPM_ALG_NULL
491       TPM2B_DIGEST        platformPolicy;     // default reset is an Empty Buffer
492       TPM2B_AUTH          platformAuth;       // default reset is an Empty Buffer
493   //***************************************************************************
494   //           PCR
495   //***************************************************************************
```

```
496   // The set of PCR to be saved on Shutdown(STATE)
497       PCR_SAVE            pcrSave;            // default reset is 0...0
498   // This structure hold the authorization values for those PCR that have an
499   // update authorization.
500   // This implementation only supports a single group of PCR controlled by
501   // authorization. If more are required, then this structure would be changed to
502   // an array.
503       PCR_AUTHVALUE       pcrAuthValues;
504   } STATE_CLEAR_DATA;
505   extern STATE_CLEAR_DATA gc;
```

5.11.11.5 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.

```
506   typedef struct state_reset_data
507   {
508   //*****************************************************************************
509   //          Hierarchy Control
510   //*****************************************************************************
511       TPM2B_AUTH          nullProof;          // The proof value associated with
512                                               // the TPM_RH_NULL hierarchy. The
513                                               // default reset value is from the RNG.
514       TPM2B_SEED          nullSeed;           // The seed value for the TPM_RN_NULL
515                                               // hierarchy. The default reset value
516                                               // is from the RNG.
517   //*****************************************************************************
518   //           Context
519   //*****************************************************************************
520   // The 'clearCount' counter is incremented each time the TPM successfully executes
521   // a TPM Resume. The counter is included in each saved context that has 'stClear'
522   // SET (including descendants of keys that have 'stClear' SET). This prevents these
523   // objects from being loaded after a TPM Resume.
524   // If 'clearCount' is at its maximum value when the TPM receives a Shutdown(STATE),
525   // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
526       UINT32              clearCount;         // The default reset value is 0.
527       UINT64              objectContextID;    // This is the context ID for a saved
528                                               //  object context. The default reset
529                                               //  value is 0.
530   #ifndef NDEBUG
531   #undef  CONTEXT_SLOT
532   #define CONTEXT_SLOT     BYTE
533   #endif
534       CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];    // This array contains
535                                               // contains the values used to track
536                                               // the version numbers of saved
537                                               // contexts (see
538                                               // Session.c in for details). The
539                                               // default reset value is {0}.
540       CONTEXT_COUNTER     contextCounter;     // This is the value from which the
541                                               // 'contextID' is derived. The
542                                               // default reset value is {0}.
543   //*****************************************************************************
544   //           Command Audit
545   //*****************************************************************************
546   // When an audited command completes, ExecuteCommand() checks the return
547   // value.  If it is TPM_RC_SUCCESS, and the command is an audited command, the
548   // TPM will extend the cpHash and rpHash for the command to this value. If this
549   // digest was the Zero Digest before the cpHash was extended, the audit counter
550   // is incremented.
551       TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest
552                                               // by TPM2_GetCommandAuditDigest() or a
```

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

5.11.12 NV Layout

The NV data organization is

a) a PERSISTENT_DATA structure

b) a STATE_RESET_DATA structure

c) a STATE_CLEAR_DATA structure

d) an ORDERLY_DATA structure

e) the user defined NV index space

```
587     #define NV_PERSISTENT_DATA  (0)
588     #define NV_STATE_RESET_DATA (NV_PERSISTENT_DATA + sizeof(PERSISTENT_DATA))
589     #define NV_STATE_CLEAR_DATA (NV_STATE_RESET_DATA + sizeof(STATE_RESET_DATA))
590     #define NV_ORDERLY_DATA     (NV_STATE_CLEAR_DATA + sizeof(STATE_CLEAR_DATA))
591     #define NV_INDEX_RAM_DATA   (NV_ORDERLY_DATA + sizeof(ORDERLY_DATA))
592     #define NV_USER_DYNAMIC     (NV_INDEX_RAM_DATA + sizeof(s_indexOrderlyRam))
593     #define NV_USER_DYNAMIC_END     NV_MEMORY_SIZE
```

5.11.13 Global Macro Definitions

The NV_READ_PERSISTENT and NV_WRITE_PERSISTENT macros are used to access members of the PERSISTENT_DATA structure in NV.

```
594     #define NV_READ_PERSISTENT(to, from)      \
595                 NvRead(&to, offsetof(PERSISTENT_DATA, from), sizeof(to))
```

```
596  #define NV_WRITE_PERSISTENT(to, from)     \
597              NvWrite(offsetof(PERSISTENT_DATA, to), sizeof(gp.to), &from)
598  #define CLEAR_PERSISTENT(item)                          \
599      NvClearPersistent(offsetof(PERSISTENT_DATA, item), sizeof(gp.item))
600  #define NV_SYNC_PERSISTENT(item) NV_WRITE_PERSISTENT(item, gp.item)
```

At the start of command processing, the index of the command is determined. This index value is used to access the various data tables that contain per-command information. There are multiple options for how the per-command tables can be implemented. This is resolved in GetClosestCommandIndex().

```
601  typedef UINT16      COMMAND_INDEX;
602  #define UNIMPLEMENTED_COMMAND_INDEX     ((COMMAND_INDEX)(~0))
603  typedef struct _COMMAND_FLAGS_
604  {
605      unsigned    trialPolicy : 1;    //1) If SET, one of the handles references a
606                                      //   trial policy and authorization may be
607                                      //   skipped. This is only allowed for a policy
608                                      //   command.
609  } COMMAND_FLAGS;
```

This structure is used to avoid having to manage a large number of parameters being passed through various levels of the command input processing.

```
610  typedef struct _COMMAND_
611  {
612      TPM_ST           tag;               // the parsed command tag
613      TPM_CC           code;              // the parsed command code
614      COMMAND_INDEX    index;             // the computed command index
615      UINT32           handleNum;         // the number of entity handles in the
616                                          //   handle area of the command
617      TPM_HANDLE       handles[MAX_HANDLE_NUM]; // the parsed handle values
618      UINT32           sessionNum;        // the number of sessions found
619      INT32            parameterSize;     // starts out with the parsed command size
620                                          // and is reduced and values are
621                                          // unmarshaled. Just before calling the
622                                          // command actions, this should be zero.
623                                          // After the command actions, this number
624                                          // should grow as values are marshaled
625                                          // in to the response buffer.
626      INT32            authSize;          // this is initialized with the parsed size
627                                          // of authorizationSize field and should
628                                          // be zero when the authorizations are
629                                          // parsed.
630      BYTE            *parameterBuffer;   // input to ExecuteCommand
631      BYTE            *responseBuffer;    // input to ExecuteCommand
632  #if ALG_SHA1
633      TPM2B_SHA1_DIGEST   sha1CpHash;
634      TPM2B_SHA1_DIGEST   sha1RpHash;
635  #endif
636  #if ALG_SHA256
637      TPM2B_SHA256_DIGEST sha256CpHash;
638      TPM2B_SHA256_DIGEST sha256RpHash;
639  #endif
640  #if ALG_SHA384
641      TPM2B_SHA384_DIGEST sha384CpHash;
642      TPM2B_SHA384_DIGEST sha384RpHash;
643  #endif
644  #if ALG_SHA512
645      TPM2B_SHA512_DIGEST sha512CpHash;
646      TPM2B_SHA512_DIGEST sha512RpHash;
647  #endif
648  #if ALG_SM3_256
649      TPM2B_SM3_256_DIGEST sm3_256CpHash;
650      TPM2B_SM3_256_DIGEST sm3_256RpHash;
```

```
651     #endif
652     } COMMAND;
```

Global sting constants for consistency in KDF function calls.

```
653     extern const TPM2B      *PRIMARY_OBJECT_CREATION;
654     extern const TPM2B      *SECRET_KEY;
655     extern const TPM2B      *SESSION_KEY;
656     extern const TPM2B      *STORAGE_KEY;
657     extern const TPM2B      *INTEGRITY_KEY;
658     extern const TPM2B      *CONTEXT_KEY;
659     extern const TPM2B      *CFB_KEY;
660     extern const TPM2B      *XOR_KEY;
661     extern const TPM2B      *DUPLICATE_STRING;
662     extern const TPM2B      *OBFUSCATE_STRING;
663     extern const TPM2B      *IDENTITY_STRING;
664     extern const TPM2B      *COMMIT_STRING;
665     #ifdef SELF_TEST
666     extern const TPM2B      *OAEP_TEST_STRING;
667     #endif // SELF_TEST
```

From Manufacture.c

```
668     extern BOOL              g_manufactured;
```

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.

```
669     extern BOOL              g_initialized;
```

5.11.14 Private data

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

From SessionProcess.c

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

Array of the authorization session handles

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

Array of authorization session attributes

```
672     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

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

Array of nonces provided by the caller for the corresponding sessions

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

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

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

Array of pointers to the SESSION structures for the sessions in a command

```
676   extern SESSION          *s_usedSessions[MAX_SESSION_NUM];
```

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter

```
678   extern UINT32           s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
679   extern UINT32           s_decryptSessionIndex;
```

Index of a session used for audit

```
680   extern UINT32           s_auditSessionIndex;
```

The *cpHash* for command audit

```
681   #ifdef  TPM_CC_GetCommandAuditDigest
682   extern TPM2B_DIGEST    s_cpHashForCommandAudit;
683   #endif
```

Number of authorization sessions present in the command

extern UINT32	***s_sessionNum*; Flag indicating if NV update is pending for the *lockOutAuthEnabled* or *failedTries* DA parameter**

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

From DA.c

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

```
687   extern UINT64       s_selfHealTimer;
```

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

```
688   extern UINT64       s_lockoutTimer;
689   #endif // DA_C
690   #if defined NV_C || defined GLOBAL_C
```

From NV.c

This marks the end of the NV area. This is a run-time variable as it might not be compile-time constant.

```
691   extern NV_REF   s_evictNvEnd;
```

This space is used to hold the index data for an orderly Index. It also contains the attributes for the index.

```
692   extern BYTE      s_indexOrderlyRam[RAM_INDEX_SPACE];   // The orderly NV Index data
```

This value contains the current max counter value. It is written to the end of allocatable NV space each time an index is deleted or added. This value is initialized on Startup. The indices are searched and the maximum of all the current counter indices and this value is the initial value for this.

```
693   extern UINT64    s_maxCounter;
```

This is space used for the NV Index cache. As with a persistent object, the contents of a referenced index are copied into the cache so that the NV Index memory scanning and data copying can be reduced. Only code that operates on NV Index data should use this cache directly. When that action code runs, *s_lastNvIndex* will contain the index header information. It will have been loaded when the handles were verified.

NOTE: An NV index handle can appear in many commands that do not operate on the NV data (e.g. TPM2_StartAuthSession()). However, only one NV Index at a time is ever directly referenced by any command. If that changes, then the NV Index caching needs to be changed to accommodate that. Currently, the code will verify that only one NV Index is referenced by the handles of the command.

```
694   extern      NV_INDEX         s_cachedNvIndex;
695   extern      NV_REF           s_cachedNvRef;
696   extern      BYTE            *s_cachedNvRamRef;
```

Initial NV Index/evict object iterator value

```
697   #define     NV_REF_INIT     (NV_REF)0xFFFFFFFF
698   #endif
699   #if defined OBJECT_C || defined GLOBAL_C
```

From Object.c

This type is the container for an object.

```
700   extern OBJECT           s_objects[MAX_LOADED_OBJECTS];
701   #endif // OBJECT_C
702   #if defined PCR_C || defined GLOBAL_C
```

From PCR.c

```
703   typedef struct
704   {
705   #ifdef TPM_ALG_SHA1
706       // SHA1 PCR
707       BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
708   #endif
709   #ifdef TPM_ALG_SHA256
710       // SHA256 PCR
711       BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
712   #endif
713   #ifdef TPM_ALG_SHA384
714       // SHA384 PCR
715       BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
716   #endif
717   #ifdef TPM_ALG_SHA512
718       // SHA512 PCR
719       BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
720   #endif
721   #ifdef TPM_ALG_SM3_256
722       // SHA256 PCR
723       BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
724   #endif
725   } PCR;
726   typedef struct
727   {
728       unsigned int    stateSave : 1;              // if the PCR value should be
```

```
729                                                 // saved in state save
730     unsigned int    resetLocality : 5;          // The locality that the PCR
731                                                 // can be reset
732     unsigned int    extendLocality : 5;         // The locality that the PCR
733                                                 // can be extend
734 } PCR_Attributes;
735 extern PCR          s_pcrs[IMPLEMENTATION_PCR];
736 #endif // PCR_C
737 #if defined SESSION_C || defined GLOBAL_C
```

From Session.c

Container for HMAC or policy session tracking information

```
738 typedef struct
739 {
740     BOOL                occupied;
741     SESSION             session;        // session structure
742 } SESSION_SLOT;
743 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.

```
744 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)

```
745 extern int              s_freeSessionSlots;
746 #endif // SESSION_C
747 #if defined IO_BUFFER_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.

```
748 extern UINT32   s_actionInputBuffer[1024];         // action input buffer
749 extern UINT32   s_actionOutputBuffer[1024];        // action output buffer
750 #endif // MEMORY_LIB_C
```

From TPMFail.c

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

```
751 extern BOOL      g_inFailureMode;       // Indicates that the TPM is in failure mode
752 #ifdef SIMULATION
753 extern BOOL      g_forceFailureMode;    // flag to force failure mode during test
754 #endif
755 typedef void(FailFunction)(const char *function, int line, int code);
756 #if defined TPM_FAIL_C || defined GLOBAL_C || 1
757 extern UINT32    s_failFunction;
758 extern UINT32    s_failLine;            // the line in the file at which
759                                         // the error was signaled
760 extern UINT32    s_failCode;            // the error code used
761 extern FailFunction    *LibFailCallback;
762 #endif // TPM_FAIL_C
```

From CommandCodeAttributes.c

```
763   extern  const  TPMA_CC              s_ccAttr[];
764   extern  const  COMMAND_ATTRIBUTES   s_commandAttributes[];
765   #endif // GLOBAL_H
```


5.12 GpMacros.h

5.12.1 Introduction

This file is a collection of miscellaneous macros.

```
#ifndef GP_MACROS_H
#define GP_MACROS_H
#ifndef NULL
#define NULL 0
#endif
#include "swap.h"
#include "VendorString.h"
#ifdef SELF_TEST
```

5.12.2 For Self-test

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

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

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

```
#   define     TEST_HASH(alg)                                                  \
            if(TEST_BIT(alg, g_toTest)                                         \
               &&  (alg != ALG_NULL_VALUE))                                    \
               CryptTestAlgorithm(alg, NULL)
#else
#   define TEST(alg)
#   define TEST_HASH(alg)
#endif // SELF_TEST
```

5.12.3 For Failures

```
#if defined _POSIX_
#   define FUNCTION_NAME       0
#else
#   define FUNCTION_NAME       __FUNCTION__
#endif
#ifdef NO_FAIL_TRACE
#   define FAIL(errorCode) (TpmFail(errorCode))
#else
#   define FAIL(errorCode) (TpmFail(FUNCTION_NAME, __LINE__, errorCode))
#endif
```

If implementation is using longjmp, then the call to TpmFail() does not return and the compiler will complain about unreachable code that comes after. To allow for not having longjmp, TpmFail() will return and the subsequent code will be executed. This macro accounts for the difference.

```
#ifndef NO_LONGJMP
#   define FAIL_RETURN(returnCode)
#   define TPM_FAIL_RETURN     NORETURN void
#else
#   define FAIL_RETURN(returnCode) return (returnCode)
#   define TPM_FAIL_RETURN     void
#endif
```

This macro tests that a condition is TRUE and puts the TPM into failure mode if it is not. If longjmp is being used, then the FAIL(FATAL_ERROR_) macro makes a call from which there is no return. Otherwise, it returns and the function will exit with the appropriate return code.

```
35  #define REQUIRE(condition, errorCode, returnCode)       \
36      {                                                   \
37          if(!!(condition))                               \
38          {                                               \
39              FAIL(FATAL_ERROR_errorCode);                            \
40              FAIL_RETURN(returnCode);                    \
41          }                                               \
42      }
43  #define PARAMETER_CHECK(condition, returnCode)          \
44      REQUIRE((condition), PARAMETER, returnCode)
45  #if defined(EMPTY_ASSERT)
46  #   define pAssert(a)  ((void)0)
47  #else
```

The additional parameter following FAIL(FATAL_ERROR_) is so that the expression within parenthesis has an lvalue. FAIL has no value so the expression is not complete.

```
48  #   define pAssert(a) (!!(a) ? 1 : (FAIL(FATAL_ERROR_PARAMETER), 0))
49  #endif
```

5.12.4 Derived from Vendor-specific values

Values derived from vendor specific settings in Implementation.h

```
50  #define PCR_SELECT_MIN          ((PLATFORM_PCR+7)/8)
51  #define PCR_SELECT_MAX          ((IMPLEMENTATION_PCR+7)/8)
52  #define MAX_ORDERLY_COUNT       ((1 << ORDERLY_BITS) - 1)
53  #define PRIVATE_VENDOR_SPECIFIC_BYTES                           \
54                  ((MAX_RSA_KEY_BYTES/2) * (3 + CRT_FORMAT_RSA * 2))
```

5.12.5 Compile-time Checks

In some cases, the relationship between two values may be dependent on things that change based on various selections like the chosen crypto libraries. It is possible that these selections will result in incompatible settings. These are often detectable by the compiler but it isn't always possible to do the check in the preprocessor code. For example, when the check requires use of **sizeof** then the preprocessor can't do the comparison. For these cases, we include a special macro that, depending on the compiler will generate a warning to indicate if the check always passes or always fails because it involves fixed constants. To run these checks, define COMPILER_CHECKS in TpmBuildSwitches.h

```
55  #ifdef COMPILER_CHECKS
56  #   define  cAssert     pAssert
57  #else
58  #   define cAssert(value)
59  #endif
```

This is used commonly in the **Crypt** code as a way to keep listings from getting too long. This is not to save paper but to allow one to see more useful stuff on the screen at any given time.

```
60  #define     ERROR_RETURN(returnCode)    \
61      {                                   \
62           retVal = returnCode;           \
63           goto Exit;                     \
64      }
65  #ifndef MAX
66  #  define MAX(a, b) ((a) > (b) ? (a) : (b))
```

```
#endif
#ifndef MIN
#  define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif
#ifndef IsOdd
#  define IsOdd(a)        (((a) & 1) != 0)
#endif
#ifndef BITS_TO_BYTES
#  define BITS_TO_BYTES(bits) (((bits) + 7) >> 3)
#endif
```

These are defined for use when the size of the vector being checked is known at compile time.

```
#define TEST_BIT(bit, vector)   TestBit((bit), (BYTE *)&(vector), sizeof(vector))
#define SET_BIT(bit, vector)    SetBit((bit), (BYTE *)&(vector), sizeof(vector))
#define CLEAR_BIT(bit, vector) ClearBit((bit), (BYTE *)&(vector), sizeof(vector))
```

The following definitions are used if they have not already been defined. The defaults for these settings are compatible with ISO/IEC 9899:2011 (E)

```
#ifndef LIB_EXPORT
#   define LIB_EXPORT
#   define LIB_IMPORT
#endif
#ifndef NORETURN
#   define NORETURN _Noreturn
#endif
#ifndef NOT_REFERENCED
#   define NOT_REFERENCED(x = x)   ((void) (x))
#endif
```

Need an unambiguous definition for DEBUG. Don't change this

```
#if !defined NDEBUG && !defined DEBUG
#  define DEBUG YES
#endif
#define STD_RESPONSE_HEADER (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC))
#ifndef CONTEXT_HASH_ALGORITHM
#   if defined ALG_SHA512 && ALG_SHA512 == YES
#       define CONTEXT_HASH_ALGORITHM    SHA512
#   elif defined ALG_SHA384 && ALG_SHA384 == YES
#       define CONTEXT_HASH_ALGORITHM    SHA384
#   elif defined ALG_SHA256 && ALG_SHA256 == YES
#       define CONTEXT_HASH_ALGORITHM    SHA256
#   elif defined ALG_SM3_256 && ALG_SM3_256 == YES
#       define CONTEXT_HASH_ALGORITHM    SM3_256
#   elif defined ALG_SHA1 && ALG_SHA1 == YES
#       define CONTEXT_HASH_ALGORITHM    SHA1
#   endif
#endif
#define JOIN(x,y) x##y
#define CONCAT(x,y) JOIN(x, y)
```

If CONTEXT_INTEGRITY_HASH_ALG is defined, then the vendor is using the old style table

```
#ifndef CONTEXT_INTEGRITY_HASH_ALG
#define CONTEXT_INTEGRITY_HASH_ALG      CONCAT(TPM_ALG_, CONTEXT_HASH_ALGORITHM)
#define CONTEXT_INTEGRITY_HASH_SIZE     CONCAT(CONTEXT_HASH_ALGORITHM, _DIGEST_SIZE)
#endif
#define PROOF_SIZE                      CONTEXT_INTEGRITY_HASH_SIZE
```

If CONTEXT_ENCRYP_ALG is defined, then the vendor is using the old style table

```
114   #ifndef CONTEXT_ENCRYPT_ALG
115   #define CONTEXT_ENCRYPT_ALG             CONCAT(TPM_ALG_, CONTEXT_ENCRYPT_ALGORITHM)
116   #define CONTEXT_ENCRYPT_KEY_BITS                                \
117                   CONCAT(CONCAT(MAX_, CONTEXT_ENCRYPT_ALGORITHM), _KEY_BITS)
118   #define CONTEXT_ENCRYPT_KEY_BYTES       ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
119   #endif
120   #ifndef MAX_ECC_KEY_BYTES
121   #define MAX_ECC_KEY_BYTES 0
122   #endif
```

Handle case when no ecc is defined

```
123   #ifndef MAX_ECC_KEY_BYTES
124   #   define MAX_ECC_KEY_BYTES    MAX_DIGEST_SIZE
125   #endif
126   #define LABEL_MAX_BUFFER   MIN(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE)
127   #if LABEL_MAX_BUFFER < 32
128   #error "The size allowed for the label is not large enough for interoperability."
129   #endif
130   #endif // GP_MACROS_H
```


5.13 InternalRoutines.h

```
#ifndef     INTERNAL_ROUTINES_H
#define     INTERNAL_ROUTINES_H
#if !defined _LIB_SUPPORT_H_ && !defined _TPM_H_
#error "Should not be called"
#endif
```

DRTM functions

```
#include "_TPM_Hash_Start_fp.h"
#include "_TPM_Hash_Data_fp.h"
#include "_TPM_Hash_End_fp.h"
```

Internal subsystem functions

```
#include "Object_fp.h"
#include "Context_spt_fp.h"
#include "Object_spt_fp.h"
#include "Entity_fp.h"
#include "Session_fp.h"
#include "Hierarchy_fp.h"
#include "NvReserved_fp.h"
#include "NvDynamic_fp.h"
#include "NV_spt_fp.h"
#include "PCR_fp.h"
#include "DA_fp.h"
#include "TpmFail_fp.h"
#include "SessionProcess_fp.h"
```

Internal support functions

```
#include "CommandCodeAttributes_fp.h"
#include "Marshal_fp.h"
#include "Time_fp.h"
#include "Locality_fp.h"
#include "PP_fp.h"
#include "CommandAudit_fp.h"
#include "Manufacture_fp.h"
#include "Handle_fp.h"
#include "Power_fp.h"
#include "Response_fp.h"
#include "CommandDispatcher_fp.h"
```

Miscellaneous

```
#include "Bits_fp.h"
#include "AlgorithmCap_fp.h"
#include "PropertyCap_fp.h"
#include "IoBuffers_fp.h"
#include "Memory_fp.h"
#include "ResponseCodeProcessing_fp.h"
```

Internal crypto functions

```
#include "BnConvert_fp.h"
#include "BnMath_fp.h"
#include "BnMemory_fp.h"
#include "Ticket_fp.h"
#include "CryptUtil_fp.h"
#include "CryptHash_fp.h"
#include "CryptSym_fp.h"
```

```
46  #include "CryptDes_fp.h"
47  #include "CryptPrime_fp.h"
48  #include "CryptRand_fp.h"
49  #include "CryptSelfTest_fp.h"
50  #include "MathOnByteBuffers_fp.h"
51  #include "CryptSym_fp.h"
52  #include "AlgorithmTests_fp.h"
53  #ifdef TPM_ALG_RSA
54  #include "CryptRsa_fp.h"
55  #include "CryptPrimeSieve_fp.h"
56  #endif
57  #ifdef TPM_ALG_ECC
58  #include "CryptEccMain_fp.h"
59  #include "CryptEccSignature_fp.h"
60  #include "CryptEccKeyExchange_fp.h"
61  #endif
```

Support library

```
62  #include "SupportLibraryFunctionPrototypes_fp.h"
```

Linkage to platform functions

```
63  #include "Platform_fp.h"
64  #endif
```


5.14 LibSupport.h

This header file is used to select the library code that gets included in the TPM built

```
1   #ifndef _LIB_SUPPORT_H_
2   #define _LIB_SUPPORT_H_
```

OSSL has a full suite but yields an executable that is much larger than it needs to be.

```
3   #define     OSSL        1
```

LTC has symmetric support, RSA support, and inadequate ECC support

```
4   #define     LTC         2
```

MSBN only provides math support so should not be used as the hash or symmetric library

```
5   #define     MSBN        3
```

SYMCRYPT only provides symmetric cryptography so would need to be combined with another library that has math support

```
6   #define     SYMCRYPT    4
7   #if RADIX_BITS == 32
8   #   define RADIX_BYTES 4
9   #elif RADIX_BITS == 64
10  #   define RADIX_BYTES 8
11  #else
12  #error  "RADIX_BITS must either be 32 or 64."
13  #endif
```

Include the options for hashing If all the optional headers were always part of the distribution then it would not be necessary to do the conditional testing before the include.)-;

```
14  #if HASH_LIB == OSSL
15  #  include "ossl/TpmToOsslHash.h"
16  #elif HASH_LIB == LTC
17  #  include "ltc/TpmToLtcHash.h"
18  #elif HASH_LIB == SYMCRYPT
19  #include "symcrypt/TpmToSymcryptHash.h"
20  #else
21  #  error "No hash library selected"
22  #endif
```

Set the linkage for the selected symmetric library

```
23  #if SYM_LIB == OSSL
24  #  include "ossl/TpmToOsslSym.h"
25  #elif SYM_LIB == LTC
26  #  include "ltc/TpmToLtcSym.h"
27  #elif SYM_LIB == SYMCRYPT
28  #include "symcrypt/TpmToSymcryptSym.h"
29  #else
30  #  error "No symmetric library selected"
31  #endif
32  #undef MIN
33  #undef MIN
```

Select a big number Library. This uses a define rather than an include so that the header will not be included until the required values have been defined.


```
34  #if MATH_LIB == OSSL
35  #  define MATHLIB_H  "ossl/TpmToOsslMath.h"
36  #elif MATH_LIB == LTC
37  #  define MATHLIB_H  "ltc/TpmToLtcMath.h"
38  #elif MATH_LIB == MSBN
39  #define MATHLIB_H  "msbn/TpmToMsBnMath.h"
40  #else
41  #  error "No math library selected"
42  #endif
43  #endif // _LIB_SUPPORT_H_
```


5.15 NV.h

```
#ifndef    _NV_H_
#define    _NV_H_
```

5.15.1 Index Type Definitions

These definitions allow the same code to be used pre and post 1.21. The main action is to redefine the index type values from the bit values. Use TPM_NT_ORDINARY to indicate if the TPM_NT type is defined

```
#ifdef     TPM_NT_ORDINARY
#   define NV_ATTRIBUTES_TO_TYPE(attributes) (attributes.TPM_NT)
#else
#   define NV_ATTRIBUTES_TO_TYPE(attributes)        \
        (   attributes.TPMA_NV_COUNTER              \
        +   (attributes.TPMA_NV_BITS << 1)          \
        +   (attributes.TPMA_NV_EXTEND << 2)        \
        )
#endif
```

5.15.2 Attribute Macros

These macros are used to isolate the differences in the way that the index type changed in version 1.21 of the specification

```
#ifdef TPM_NT_ORDINARY
#   define  IsNvOrdinaryIndex(attributes) (attributes.TPM_NT == TPM_NT_ORDINARY)
#else
#   define IsNvOrdinaryIndex(attributes)                                    \
            ((attributes.TPMA_NV_COUNTER == CLEAR)                          \
            && (attributes.TPMA_NV_BITS == CLEAR)                           \
            && (attributes.TPMA_NV_EXTEND == CLEAR)
#   define TPM_NT_ORDINARY  (0)
#endif
#ifdef  TPM_NT_COUNTER
#   define  IsNvCounterIndex(attributes) (attributes.TPM_NT == TPM_NT_COUNTER)
#else
#   define IsNvCounterIndex(attributes) (attributes.TPMA_NV_COUNTER == SET)
#   define  TPM_NT_COUNTER  (1)
#endif
#ifdef  TPM_NT_BITS
#   define  IsNvBitsIndex(attributes) (attributes.TPM_NT == TPM_NT_BITS)
#else
#   define  IsNvBitsIndex(attributes) (attributes.TPMA_NV_BITS == SET)
#   define  TPM_NT_BITS (2)
#endif
#ifdef  TPM_NT_EXTEND
#   define  IsNvExtendIndex(attributes) (attributes.TPM_NT == TPM_NT_EXTEND)
#else
#   define IsNvExtendIndex(attributes) (attributes.TPMA_NV_EXTEND == SET)
#   define  TPM_NT_EXTEND   (4)
#endif
#ifdef TPM_NT_PIN_PASS
#   define  IsNvPinPassIndex(attributes) (attributes.TPM_NT == TPM_NT_PIN_PASS)
#endif
#ifdef TPM_NT_PIN_FAIL
#   define  IsNvPinFailIndex(attributes) (attributes.TPM_NT == TPM_NT_PIN_FAIL)
#endif
typedef struct {
    UINT32      size;
    TPM_HANDLE  handle;
```

```
48  } NV_ENTRY_HEADER;
49  #define NV_EVICT_OBJECT_SIZE        \
50       (sizeof(UINT32)  + sizeof(TPM_HANDLE) + sizeof(OBJECT))
51  #define NV_INDEX_COUNTER_SIZE       \
52      (sizeof(UINT32) + sizeof(NV_INDEX) + sizeof(UINT64))
53  #define NV_RAM_INDEX_COUNTER_SIZE   \
54      (sizeof(NV_RAM_HEADER) + sizeof(UINT64))
55  typedef struct {
56      UINT32          size;
57      TPM_HANDLE      handle;
58      TPMA_NV         attributes;
59  } NV_RAM_HEADER;
```

Defines the end-of-list marker for NV. The list terminator is a UINT32 of zero, followed by the current value of *s_maxCounter* which is a 64-bit value. The structure is defined as an array of 3 UINT32 values so that there is no padding between the UINT32 list end marker and the UIT64m() *maxCounter* value.

```
60  typedef UINT32 NV_LIST_TERMINATOR[3];
```

The following defines are for accessing orderly RAM values. This is the initialize for the RAM reference iterator.

```
61  #define     NV_RAM_REF_INIT         0
```

This is the starting address of the RAM space used for orderly data

```
62  #define     RAM_ORDERLY_START               \
63                  (&s_indexOrderlyRam[0])
```

This is the offset within NV that is used to save the orderly data on an orderly shutdown.

```
64  #define     NV_ORDERLY_START                \
65                  (NV_INDEX_RAM_DATA)
```

This is the end of the orderly RAM space. It is actually the first byte after the last byte of orderly RAM data

```
66  #define     RAM_ORDERLY_END                 \
67                  (RAM_ORDERLY_START + sizeof(s_indexOrderlyRam))
```

This is the end of the orderly space in NV memory. As with RAM_ORDERLY_END, it is actually the offset of the first byte after the end of the NV orderly data.

```
68  #define     NV_ORDERLY_END                  \
69                  (NV_ORDERLY_START + sizeof(s_indexOrderlyRam))
```

Macro to check that an orderly RAM address is with range.

```
70  #define ORDERLY_RAM_ADDRESS_OK(start, offset)       \
71          ((start >= RAM_ORDERLY_START) && ((start + offset - 1) < RAM_ORDERLY_END))
72  #define RETURN_IF_NV_IS_NOT_AVAILABLE               \
73  {                                                   \
74      if(g_NvStatus != TPM_RC_SUCCESS)                \
75          return g_NvStatus;                          \
76  }
```

Routinely have to clear the orderly flag and fail if the NV is not available so that it can be cleared.

```
77  #define RETURN_IF_ORDERLY                           \
78  {                                                   \
79      if(NvClearOrderly() != TPM_RC_SUCCESS)          \
80          return g_NvStatus;                          \
```

```
81    }
82    #define NV_IS_AVAILABLE     (g_NvStatus == TPM_RC_SUCCESS)
83    #define IS_ORDERLY(value)   (value < SU_DA_USED_VALUE)
84    #define NV_IS_ORDERLY       (IS_ORDERLY(gp.orderlyState))
```

Macro to set the NV UPDATE_TYPE. This deals with the fact that the update is possibly a combination of UT_NV and UT_ORDERLY.

```
85    #define SET_NV_UPDATE(type)     g_updateNV |= (type)
86    #endif  // _NV_H_
```


5.16 PRNG_TestVectors.h

```
#ifndef     _MSBN_DRBG_TEST_VECTORS_H
#define     _MSBN_DRBG_TEST_VECTORS_H
//#if DRBG_ALGORITHM == TPM_ALG_AES && DRBG_KEY_BITS == 256
#if DRBG_KEY_SIZE_BITS == 256
```

NIST test vector [AES-256 no df] [PredictionResistance() = False] [EntropyInputLen() = 384] [NonceLen() = 128] [PersonalizationStringLen() = 0] [AdditionalInputLen() = 0] COUNT = 0 EntropyInput() = 0d15aa80 b16c3a10 906cfedb 795dae0b 5b81041c 5c5bfacb 373d4440 d9120f7e 3d6cf909 86cf52d8 5d3e947d 8c061f91 Nonce = 06caef5f b538e08e 1f3b0452 03f8f4b2 PersonalizationString() = AdditionalInput() = INTERMEDIATE Key = be5df629 34cc1230 166a6773 345bbd6b 4c8869cf 8aec1c3b 1aa98bca 37cacf61 INTERMEDIATE V = 3182dd1e 7638ec70 014e93bd 813e524c INTERMEDIATE ReturnedBits() = 28e0ebb8 21016650 8c8f65f2 207bd0a3 EntropyInputReseed() = 6ee793a3 3955d72a d12fd80a 8a3fcf95 ed3b4dac 5795fe25 cf869f7c 27573bbc 56f1acae 13a65042 b340093c 464a7a22 AdditionalInputReseed() = AdditionalInput() = ReturnedBits() = 946f5182 d54510b9 461248f5 71ca06c9

Entropy is the size of a the state. The state is the size of the key plus the IV. The IV is a block. Key = 256, block = 128, state = 384

```
#define DRBG_TEST_INITIATE_ENTROPY \
        0x0d, 0x15, 0xaa, 0x80, 0xb1, 0x6c, 0x3a, 0x10, \
        0x90, 0x6c, 0xfe, 0xdb, 0x79, 0x5d, 0xae, 0x0b, \
        0x5b, 0x81, 0x04, 0x1c, 0x5c, 0x5b, 0xfa, 0xcb, \
        0x37, 0x3d, 0x44, 0x40, 0xd9, 0x12, 0x0f, 0x7e, \
        0x3d, 0x6c, 0xf9, 0x09, 0x86, 0xcf, 0x52, 0xd8, \
        0x5d, 0x3e, 0x94, 0x7d, 0x8c, 0x06, 0x1f, 0x91
#define DRBG_TEST_RESEED_ENTROPY  \
        0x6e, 0xe7, 0x93, 0xa3, 0x39, 0x55, 0xd7, 0x2a, \
        0xd1, 0x2f, 0xd8, 0x0a, 0x8a, 0x3f, 0xcf, 0x95, \
        0xed, 0x3b, 0x4d, 0xac, 0x57, 0x95, 0xfe, 0x25, \
        0xcf, 0x86, 0x9f, 0x7c, 0x27, 0x57, 0x3b, 0xbc, \
        0x56, 0xf1, 0xac, 0xae, 0x13, 0xa6, 0x50, 0x42, \
        0xb3, 0x40, 0x09, 0x3c, 0x46, 0x4a, 0x7a, 0x22
#define DRBG_TEST_GENERATED_INTERM \
        0x28, 0xe0, 0xeb, 0xb8, 0x21, 0x01, 0x66, 0x50, \
        0x8c, 0x8f, 0x65, 0xf2, 0x20, 0x7b, 0xd0, 0xa3
#define DRBG_TEST_GENERATED \
        0x94, 0x6f, 0x51, 0x82, 0xd5, 0x45, 0x10, 0xb9, \
        0x46, 0x12, 0x48, 0xf5, 0x71, 0xca, 0x06, 0xc9
//#elif DRBG_ALGORITHM == TPM_ALG_AES && DRBG_KEY_BITS == 128
#elif DRBG_KEY_SIZE_BITS == 128
```

[AES-128 no df] [PredictionResistance() = False] [EntropyInputLen() = 256] [NonceLen() = 64] [PersonalizationStringLen() = 0] [AdditionalInputLen() = 0] COUNT = 0 EntropyInput() = 8fc11bdb5aabb7e093b61428e0907303cb459f3b600dad870955f22da80a44f8 Nonce = be1f73885ddd15aa PersonalizationString() = AdditionalInput() = INTERMEDIATE Key = b134ecc836df6dbd624900af118dd7e6 INTERMEDIATE V = 01bb09e86dabd75c9f26dbf6f9531368 INTERMEDIATE ReturnedBits() = dc3cf6bf5bd341135f2c6811a1071c87 EntropyInputReseed() = 0cd53cd5eccd5a10d7ea266111259b05574fc6ddd8bed8bd72378cf82f1dba2a AdditionalInputReseed() = AdditionalInput() = ReturnedBits() = b61850decfd7106d44769a8e6e8c1ad4

```
#define DRBG_TEST_INITIATE_ENTROPY \
        0x8f, 0xc1, 0x1b, 0xdb, 0x5a, 0xab, 0xb7, 0xe0, \
        0x93, 0xb6, 0x14, 0x28, 0xe0, 0x90, 0x73, 0x03, \
        0xcb, 0x45, 0x9f, 0x3b, 0x60, 0x0d, 0xad, 0x87, \
        0x09, 0x55, 0xf2, 0x2d, 0xa8, 0x0a, 0x44, 0xf8
#define DRBG_TEST_RESEED_ENTROPY  \
        0x0c, 0xd5, 0x3c, 0xd5, 0xec, 0xcd, 0x5a, 0x10, \
        0xd7, 0xea, 0x26, 0x61, 0x11, 0x25, 0x9b, 0x05, \
        0x57, 0x4f, 0xc6, 0xdd, 0xd8, 0xbe, 0xd8, 0xbd, \
```

```
36          0x72, 0x37, 0x8c, 0xf8, 0x2f, 0x1d, 0xba, 0x2a
37  #define DRBG_TEST_GENERATED_INTERM \
38          0xdc, 0x3c, 0xf6, 0xbf, 0x5b, 0xd3, 0x41, 0x13, \
39          0x5f, 0x2c, 0x68, 0x11, 0xa1, 0x07, 0x1c, 0x87
40  #define DRBG_TEST_GENERATED \
41          0xb6, 0x18, 0x50, 0xde, 0xcf, 0xd7, 0x10, 0x6d, \
42          0x44, 0x76, 0x9a, 0x8e, 0x6e, 0x8c, 0x1a, 0xd4
43  #endif
44  #endif      //     _MSBN_DRBG_TEST_VECTORS_H
```

5.17 SelfTest.h

```
1   #ifndef                 _SELF_TEST_H_
2   #define                 _SELF_TEST_H_
```

5.17.1 Introduction

This file contains the structure definitions for the self-test. It also contains macros for use when the self-test is implemented.

5.17.2 Defines

Was typing this a lot

```
3   #define SELF_TEST_FAILURE   FAIL(FATAL_ERROR_SELF_TEST)
```

Use the definition of key sizes to set algorithm values for key size. Need to do this to avoid a lot of #ifdefs in the code. Also, define the index for each of the algorithms.

```
4   #if ALG_AES && defined  AES_KEY_SIZE_BITS_128
5   #   define  AES_128     YES
6   #   define  AES_128_INDEX   0
7   #else
8   #   define  AES_128     NO
9   #endif
10  #if ALG_AES && defined  AES_KEY_SIZE_BITS_192
11  #   define  AES_192     YES
12  #   define  AES_192_INDEX   (AES_128)
13  #else
14  #   define  AES_192     NO
15  #endif
16  #if ALG_AES && defined  AES_KEY_SIZE_BITS_256
17  #   define  AES_256     YES
18  #   define  AES_256_INDEX   (AES_128 + AES_192)
19  #else
20  #   define  AES_256     NO
21  #endif
22  #if ALG_SM4 && defined SM4_KEY_SIZE_BITS_128
23  #   define  SM4_128     YES
24  #   define  SM4_128_INDEX   (AES_128 + AES_192 + AES_256)
25  #else
26  #   define  SM4_128     NO
27  #endif
28  #define NUM_SYMS    (AES_128 + AES_192 + AES_256 + SM4_128)
29  typedef UINT32      SYM_INDEX;
```

These two defines deal with the fact that the TPM_ALG_ID table does not delimit the symmetric mode values with a TPM_SYM_MODE_FIRST and TPM_SYM_MODE_LAST

```
30  #define TPM_SYM_MODE_FIRST      ALG_CTR_VALUE
31  #define TPM_SYM_MODE_LAST       ALG_ECB_VALUE
32  #define NUM_SYM_MODES   (TPM_SYM_MODE_LAST - TPM_SYM_MODE_FIRST + 1)
```

Define a type to hold a bit vector for the modes.

```
33  #if NUM_SYM_MODES <= 0
34  #error  "No symmetric modes implemented"
35  #elif NUM_SYM_MODES <= 8
36  typedef BYTE    SYM_MODES;
37  #elif NUM_SYM_MODES <= 16
```



```
38  typedef UINT16  SYM_MODES;
39  #elif NUM_SYM_MODES <= 32
40  typedef UINT32  SYM_MODES;
41  #else
42  #error "Too many symmetric modes"
43  #endif
44  typedef struct {
45      const TPM_ALG_ID     alg;                    // the algorithm
46      const UINT16         keyBits;                // bits in the key
47      const BYTE          *key;                    // The test key
48      const UINT32         ivSize;                 // block size of the algorithm
49      const UINT32         dataInOutSize;          // size  to encrypt/decrypt
50      const BYTE          *dataIn;                 // data to encrypt
51      const BYTE          *dataOut[NUM_SYM_MODES];// data to decrypt
52  } SYMMETRIC_TEST_VECTOR;
53  #ifdef  TPM_ALG_RSA
54  extern const RSA_KEY        c_rsaTestKey; // This is a constant structure
55  #endif
56  #define SYM_TEST_VALUE_REF(value, alg, keyBits, mode)                          \
57      SIZED_REFERENCE(value##_##alg##keyBits##_##mode)
58  typedef struct {
59      TPM_ALG_ID      alg;
60      UINT16          keySizeBits;
61  } SYM_ALG;
62  #define SET_ALG(ALG, v)  MemorySetBit((v), ALG, sizeof(v) * 8)
63  #if ALG_SHA512
64  #       define  DEFAULT_TEST_HASH               ALG_SHA512_VALUE
65  #       define  DEFAULT_TEST_DIGEST_SIZE        SHA512_DIGEST_SIZE
66  #       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA512_BLOCK_SIZE
67  #elif ALG_SHA384
68  #       define  DEFAULT_TEST_HASH               ALG_SHA384_VALUE
69  #       define  DEFAULT_TEST_DIGEST_SIZE        SHA384_DIGEST_SIZE
70  #       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA384_BLOCK_SIZE
71  #elif ALG_SHA256
72  #       define  DEFAULT_TEST_HASH               ALG_SHA256_VALUE
73  #       define  DEFAULT_TEST_DIGEST_SIZE        SHA256_DIGEST_SIZE
74  #       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA256_BLOCK_SIZE
75  #elif ALG_SHA1
76  #       define  DEFAULT_TEST_HASH               ALG_SHA1_VALUE
77  #       define  DEFAULT_TEST_DIGEST_SIZE        SHA1_DIGEST_SIZE
78  #       define  DEFAULT_TEST_HASH_BLOCK_SIZE    SHA1_BLOCK_SIZE
79  #endif
80  #endif      // _SELF_TEST_H_
```


5.18 SupportLibraryFunctionPrototypes_fp.h

5.18.1 Introduction

This file contains the function prototypes for the functions that need to be present in the selected match library. For each function listed, there should be a small stub function. That stub provides the interface between the TPM code and the support library. In most cases, the stub function will only need to do a format conversion between the TPM big number and the support library big number. The TPM big number format was chosen to make this relatively simple and fast.

```
1   #ifndef SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H
2   #define SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H
3   LIB_EXPORT int SupportLibInit(void);
```

MathLibraryCompatibililtyCheck() This function is only used during development to make sure that the library that is being referenced is using the same size of data structures as the TPM.

```
4   void
5   MathLibraryCompatibilityCheck(
6       void
7       );
```

5.18.2 BnModMult()

Does multiply and divide returning the remainder of the divide.

```
8   LIB_EXPORT BOOL
9   BnModMult(bigNum result, bigConst op1, bigConst op2, bigConst modulus);
```

5.18.3 BnMult()

Multiplies two numbers

```
10  LIB_EXPORT BOOL BnMult(bigNum result, bigConst multiplicand, bigConst multiplier);
```

5.18.4 BnDiv()

This function divides two *bigNum* values. The function always returns TRUE.

```
11  LIB_EXPORT BOOL BnDiv(bigNum quotient, bigNum remainder,
12                        bigConst dividend, bigConst divisor);
```

5.18.5 BnMod()

```
13  #define BnMod(a, b)     BnDiv(NULL, (a), (a), (b))
14  #ifdef TPM_ALG_RSA
```

5.18.6 BnGcd()

Get the greatest common divisor of two numbers

```
15  LIB_EXPORT BOOL BnGcd(bigNum gcd, bigConst number1, bigConst number2);
```


5.18.7 BnModExp()

Do modular exponentiation using *bigNum* values.

```
16  LIB_EXPORT BOOL BnModExp(bigNum result, bigConst number,
17                           bigConst exponent, bigConst modulus);
```

5.18.8 BnModInverse()

Modular multiplicative inverse

```
18  LIB_EXPORT BOOL BnModInverse(bigNum result, bigConst number,
19                               bigConst modulus);
20  #endif // TPM_ALG_RSA
21  #ifdef TPM_ALG_ECC
```

5.18.9 BnEccModMult()

This function does a point multiply of the form R = [d]S

Return Value	Meaning
FALSE	failure in operation; treat as result being point at infinity

```
22  LIB_EXPORT BOOL BnEccModMult(bigPoint R, pointConst S, bigConst d, bigCurve E);
```

5.18.10 BnEccModMult2()

This function does a point multiply of the form R = [d]S + [u]Q

Return Value	Meaning
FALSE	failure in operation; treat as result being point at infinity

```
23  LIB_EXPORT BOOL BnEccModMult2(bigPoint R, pointConst S, bigConst d,
24                                pointConst Q, bigConst u, bigCurve E);
```

5.18.11 BnEccAdd()

This function does a point add R = S + Q

Return Value	Meaning
FALSE	failure in operation; treat as result being point at infinity

```
25  LIB_EXPORT BOOL BnEccAdd(bigPoint R, pointConst S, pointConst Q, bigCurve E);
```

5.18.12 BnCurveInitialize()

This function is used to initialize the pointers of a *bnCurve_t* structure. The structure is a set of pointers to *bigNum* values. The curve-dependent values are set by a different function.

```
26  LIB_EXPORT bigCurve BnCurveInitialize(bigCurve E, TPM_ECC_CURVE curveId);
27  #endif // TPM_ALG_ECC
28  #endif
```


5.19 TPMB.h

This file contains extra TPM2B structures

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

TPM2B Types

```
typedef struct {
    UINT16          size;
    BYTE            buffer[1];
} TPM2B, *P2B;
typedef const TPM2B     *PC2B;
```

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
```

This macro defines a TPM2B with a constant character value. This macro sets the size of the string to the size minus the terminating zero byte. This lets the user of the label add their terminating 0. This method is chosen so that existing code that provides a label will continue to work correctly.

```
#define TPM2B_STRING(name, value)           \
    static const union {                    \
        struct  {                           \
            UINT16  size;                   \
            BYTE    buffer[sizeof(value)];  \
        } t;                                \
        TPM2B   b;                          \
    } name##_ = {{sizeof(value), {value}}}; \
    const TPM2B       *name = &name##_.b
```

Macro to instance and initialize a TPM2B value

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


5.20 Tpm.h

Root header file for building any TPM.lib code

```
1   #ifndef     _TPM_H_
2   #define     _TPM_H_
3   #include "Implementation.h"
4   #include "LibSupport.h"         // Types from the library. These need to come before
5                                   // Global.h because some of the structures in
6                                   // that file depend on the structures used by the
7                                   // crypto libraries.
8   #include "GpMacros.h"           // Define additional macros
9   #include "Global.h"             // Define other TPM types
10  #include "InternalRoutines.h"   // Function prototypes
11  #endif // _TPM_H_
```


5.21 TpmBuildSwitches.h

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

The switches are guarded so that they can either be set on the command line or set here.

```
1   #ifndef _TPM_BUILD_SWITCHES_H_
2   #define _TPM_BUILD_SWITCHES_H_
```

Many of the #defines are guarded so that they can be set on the command line without causing consternation in the compiler.

```
3   #ifndef INLINE_FUNCTIONS
4   //#  define INLINE_FUNCTIONS
5   #endif
```

Don't move this include ahead of the INLINE_FUNCTIONS definition.

```
6   #include "CompilerDependencies.h"
```

This definition is required for the re-factored code

```
7   #define USE_BN_ECC_DATA
```

Comment these out as needed

```
8   #ifndef SIMULATION
9   #  define SIMULATION
10  #endif
```

Define this to run the function that checks the format compatibility for the chosen big number math library. Not all ports use this.

```
11  #if !defined LIBRARY_COMPATIBILITY_CHECK && defined SIMULATION
12  #   define LIBRARY_COMPATABILITY_CHECK
13  #endif
14  #ifndef FIPS_COMPLIANT
15  //#  define FIPS_COMPLIANT
16  #endif
```

Definition to allow alternate behavior for non-orderly startup. If there is a chance that the TPM could not update *failedTries*

```
17  #ifndef USE_DA_USED
18  #   define USE_DA_USED
19  #endif
```

Define TABLE_DRIVEN_DISPATCH to use tables rather than case statements for command dispatch and handle unmarshaling

```
20  #ifndef TABLE_DRIVEN_DISPATCH
21  #  define TABLE_DRIVEN_DISPATCH
22  #endif
```

This switch is used to enable the self-test capability in AlgorithmTests.c

```
23  #ifndef SELF_TEST
24  #define SELF_TEST
25  #endif
```

Enable the generation of RSA primes using a sieve.

```
26  #ifndef RSA_KEY_SIEVE
27  #  define RSA_KEY_SIEVE
28  #endif
```

Enable the instrumentation of the sieve process. This is used to tune the sieve variables.

```
29  #if !defined RSA_INSTRUMENT && defined RSA_KEY_SIEVE && defined SIMULATION
30  //#define RSA_INSTRUMENT
31  #endif
32  #if defined RSA_KEY_SIEVE && !defined NDEBUG && !defined RSA_INSTRUMENT
33  //# define RSA_INSTRUMENT
34  #endif
```

This switch enables the RNG state save and restore

```
35  #ifndef _DRBG_STATE_SAVE
36  #  define _DRBG_STATE_SAVE        // Comment this out if no state save is wanted
37  #endif
```

Switch added to support packed lists that leave out space associated with unimplemented commands. Comment this out to use linear lists.

NOTE: if vendor specific commands are present, the associated list is always in compressed form.

```
38  #ifndef COMPRESSED_LISTS
39  #   define COMPRESSED_LISTS
40  #endif
```

This switch indicates where clock epoch value should be stored. If this value defined, then it is assumed that the timer will change at any time so the nonce should be a random number kept in RAM. When it is not defined, then the timer only stops during power outages.

```
41  #ifndef CLOCK_STOPS
42  //#   define CLOCK_STOPS
43  #endif
```

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

```
44  #ifdef SIMULATION
```

Enables use of the key cache

```
45  #   ifndef USE_RSA_KEY_CACHE
46  //#       define USE_RSA_KEY_CACHE
47  #   endif
48  #   if defined USE_RSA_KEY_CACHE && !defined USE_KEY_CACHE_FILE
49  #       define USE_KEY_CACHE_FILE
50  #   endif
51  #   if !defined NDEBUG && !defined USE_DEBUG_RNG
```

This provides fixed seeding of the RNG when doing debug on a simulator. This should allow consistent results on test runs as long as the input parameters to the functions remains the same.

```
52  #       define USE_DEBUG_RNG
53  #   endif
54  #else
55  #   undef USE_RSA_KEY_CACHE
56  #   undef USE_KEY_CACHE_FILE
57  #   undef USE_DEBUG_RNG
```

```
58  #   undef RSA_INSTRUMENT
59  #endif  // SIMULATION
60  #ifndef NDEBUG
```

In some cases, the relationship between two values may be dependent on things that change based on various selections like the chosen crypto libraries. It is possible that these selections will result in incompatible settings. These are often detectable by the compiler but it isn't always possible to do the check in the preprocessor code. For example, when the check requires use of 'sizeof()' then the preprocessor can't do the comparison. For these cases, we include a special macro that, depending on the compiler will generate a warning to indicate if the check always passes or always fails because it involves fixed constants. To run these checks, define COMPILER_CHECKS.

```
61  #ifndef COMPILER_CHECKS
62  //#   define COMPILER_CHECKS
63  #endif
```

Some of the values (such as sizes) are the result of different options set in Implementation.h. The combination might not be consistent. A function is defined *(TpmSizeChecks*()) that is used to verify the sizes at run time. To enable the function, define this parameter.

```
64  #ifndef RUNTIME_SIZE_CHECKS
65  #define RUNTIME_SIZE_CHECKS
66  #endif
```

If doing debug, can set the DRBG to print out the intermediate test values. Before enabling this, make sure that the dbgDumpMemBlock() function has been added someplace (preferably, somewhere in CryptRand.c)

```
67  #ifndef DRBG_DEBUG_PRINT
68  //#  define DRBG_DEBUG_PRINT
69  #endif
70  #endif // NDEBUG
71  #endif // _TPM_BUILD_SWITCHES_H_
```


5.22 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_SELF_TEST               (6)
#define     FATAL_ERROR_CRYPTO                  (7)
#define     FATAL_ERROR_NV_UNRECOVERABLE        (8)
#define     FATAL_ERROR_REMANUFACTURED          (9) // indicates that the TPM has
                                                    // been re-manufactured after an
                                                    // unrecoverable NV error
#define     FATAL_ERROR_DRBG                    (10)
#define     FATAL_ERROR_MOVE_SIZE               (11)
#define     FATAL_ERROR_COUNTER_OVERFLOW        (12)
#define     FATAL_ERROR_SUBTRACT                (13)
#define     FATAL_ERROR_FORCED                  (666)
#endif // _TPM_ERROR_H
```


5.23 TpmTypes.h

```
#ifndef _TPM_TYPES_H_
#define _TPM_TYPES_H_
```

Table 2:5 - Definition of Types for Documentation Clarity *(TypedefTable())*

```
typedef  UINT32             TPM_ALGORITHM_ID;
typedef  UINT32             TPM_MODIFIER_INDICATOR;
typedef  UINT32             TPM_AUTHORIZATION_SIZE;
typedef  UINT32             TPM_PARAMETER_SIZE;
typedef  UINT16             TPM_KEY_SIZE;
typedef  UINT16             TPM_KEY_BITS;
```

Table 2:6 - Definition of TPM_SPEC Constants *(EnumTable())*

```
typedef  UINT32             TPM_SPEC;
#define  SPEC_FAMILY             0x322E3000
#define  TPM_SPEC_FAMILY         (TPM_SPEC)(SPEC_FAMILY)
#define  SPEC_LEVEL              00
#define  TPM_SPEC_LEVEL          (TPM_SPEC)(SPEC_LEVEL)
#define  SPEC_VERSION            138
#define  TPM_SPEC_VERSION        (TPM_SPEC)(SPEC_VERSION)
#define  SPEC_YEAR               2016
#define  TPM_SPEC_YEAR           (TPM_SPEC)(SPEC_YEAR)
#define  SPEC_DAY_OF_YEAR        273
#define  TPM_SPEC_DAY_OF_YEAR    (TPM_SPEC)(SPEC_DAY_OF_YEAR)
```

Table 2:7 - Definition of TPM_GENERATED Constants *(EnumTable())*

```
typedef  UINT32             TPM_GENERATED;
#define  TPM_GENERATED_VALUE    (TPM_GENERATED)(0xFF544347)
```

Table 2:16 - Definition of TPM_RC Constants *(EnumTable())*

```
typedef  UINT32             TPM_RC;
#define  TPM_RC_SUCCESS              (TPM_RC)(0x000)
#define  TPM_RC_BAD_TAG              (TPM_RC)(0x01E)
#define  RC_VER1                     (TPM_RC)(0x100)
#define  TPM_RC_INITIALIZE           (TPM_RC)(RC_VER1+0x000)
#define  TPM_RC_FAILURE              (TPM_RC)(RC_VER1+0x001)
#define  TPM_RC_SEQUENCE             (TPM_RC)(RC_VER1+0x003)
#define  TPM_RC_PRIVATE              (TPM_RC)(RC_VER1+0x00B)
#define  TPM_RC_HMAC                 (TPM_RC)(RC_VER1+0x019)
#define  TPM_RC_DISABLED             (TPM_RC)(RC_VER1+0x020)
#define  TPM_RC_EXCLUSIVE            (TPM_RC)(RC_VER1+0x021)
#define  TPM_RC_AUTH_TYPE            (TPM_RC)(RC_VER1+0x024)
#define  TPM_RC_AUTH_MISSING         (TPM_RC)(RC_VER1+0x025)
#define  TPM_RC_POLICY               (TPM_RC)(RC_VER1+0x026)
#define  TPM_RC_PCR                  (TPM_RC)(RC_VER1+0x027)
#define  TPM_RC_PCR_CHANGED          (TPM_RC)(RC_VER1+0x028)
#define  TPM_RC_UPGRADE              (TPM_RC)(RC_VER1+0x02D)
#define  TPM_RC_TOO_MANY_CONTEXTS    (TPM_RC)(RC_VER1+0x02E)
#define  TPM_RC_AUTH_UNAVAILABLE     (TPM_RC)(RC_VER1+0x02F)
#define  TPM_RC_REBOOT               (TPM_RC)(RC_VER1+0x030)
#define  TPM_RC_UNBALANCED           (TPM_RC)(RC_VER1+0x031)
#define  TPM_RC_COMMAND_SIZE         (TPM_RC)(RC_VER1+0x042)
#define  TPM_RC_COMMAND_CODE         (TPM_RC)(RC_VER1+0x043)
#define  TPM_RC_AUTHSIZE             (TPM_RC)(RC_VER1+0x044)
#define  TPM_RC_AUTH_CONTEXT         (TPM_RC)(RC_VER1+0x045)
#define  TPM_RC_NV_RANGE             (TPM_RC)(RC_VER1+0x046)
#define  TPM_RC_NV_SIZE              (TPM_RC)(RC_VER1+0x047)
#define  TPM_RC_NV_LOCKED            (TPM_RC)(RC_VER1+0x048)
```

```
50   #define  TPM_RC_NV_AUTHORIZATION     (TPM_RC)(RC_VER1+0x049)
51   #define  TPM_RC_NV_UNINITIALIZED     (TPM_RC)(RC_VER1+0x04A)
52   #define  TPM_RC_NV_SPACE             (TPM_RC)(RC_VER1+0x04B)
53   #define  TPM_RC_NV_DEFINED           (TPM_RC)(RC_VER1+0x04C)
54   #define  TPM_RC_BAD_CONTEXT          (TPM_RC)(RC_VER1+0x050)
55   #define  TPM_RC_CPHASH               (TPM_RC)(RC_VER1+0x051)
56   #define  TPM_RC_PARENT               (TPM_RC)(RC_VER1+0x052)
57   #define  TPM_RC_NEEDS_TEST           (TPM_RC)(RC_VER1+0x053)
58   #define  TPM_RC_NO_RESULT            (TPM_RC)(RC_VER1+0x054)
59   #define  TPM_RC_SENSITIVE            (TPM_RC)(RC_VER1+0x055)
60   #define  RC_MAX_FM0                  (TPM_RC)(RC_VER1+0x07F)
61   #define  RC_FMT1                     (TPM_RC)(0x080)
62   #define  TPM_RC_ASYMMETRIC           (TPM_RC)(RC_FMT1+0x001)
63   #define  TPM_RCS_ASYMMETRIC          (TPM_RC)(RC_FMT1+0x001)
64   #define  TPM_RC_ATTRIBUTES           (TPM_RC)(RC_FMT1+0x002)
65   #define  TPM_RCS_ATTRIBUTES          (TPM_RC)(RC_FMT1+0x002)
66   #define  TPM_RC_HASH                 (TPM_RC)(RC_FMT1+0x003)
67   #define  TPM_RCS_HASH                (TPM_RC)(RC_FMT1+0x003)
68   #define  TPM_RC_VALUE                (TPM_RC)(RC_FMT1+0x004)
69   #define  TPM_RCS_VALUE               (TPM_RC)(RC_FMT1+0x004)
70   #define  TPM_RC_HIERARCHY            (TPM_RC)(RC_FMT1+0x005)
71   #define  TPM_RCS_HIERARCHY           (TPM_RC)(RC_FMT1+0x005)
72   #define  TPM_RC_KEY_SIZE             (TPM_RC)(RC_FMT1+0x007)
73   #define  TPM_RCS_KEY_SIZE            (TPM_RC)(RC_FMT1+0x007)
74   #define  TPM_RC_MGF                  (TPM_RC)(RC_FMT1+0x008)
75   #define  TPM_RCS_MGF                 (TPM_RC)(RC_FMT1+0x008)
76   #define  TPM_RC_MODE                 (TPM_RC)(RC_FMT1+0x009)
77   #define  TPM_RCS_MODE                (TPM_RC)(RC_FMT1+0x009)
78   #define  TPM_RC_TYPE                 (TPM_RC)(RC_FMT1+0x00A)
79   #define  TPM_RCS_TYPE                (TPM_RC)(RC_FMT1+0x00A)
80   #define  TPM_RC_HANDLE               (TPM_RC)(RC_FMT1+0x00B)
81   #define  TPM_RCS_HANDLE              (TPM_RC)(RC_FMT1+0x00B)
82   #define  TPM_RC_KDF                  (TPM_RC)(RC_FMT1+0x00C)
83   #define  TPM_RCS_KDF                 (TPM_RC)(RC_FMT1+0x00C)
84   #define  TPM_RC_RANGE                (TPM_RC)(RC_FMT1+0x00D)
85   #define  TPM_RCS_RANGE               (TPM_RC)(RC_FMT1+0x00D)
86   #define  TPM_RC_AUTH_FAIL            (TPM_RC)(RC_FMT1+0x00E)
87   #define  TPM_RCS_AUTH_FAIL           (TPM_RC)(RC_FMT1+0x00E)
88   #define  TPM_RC_NONCE                (TPM_RC)(RC_FMT1+0x00F)
89   #define  TPM_RCS_NONCE               (TPM_RC)(RC_FMT1+0x00F)
90   #define  TPM_RC_PP                   (TPM_RC)(RC_FMT1+0x010)
91   #define  TPM_RCS_PP                  (TPM_RC)(RC_FMT1+0x010)
92   #define  TPM_RC_SCHEME               (TPM_RC)(RC_FMT1+0x012)
93   #define  TPM_RCS_SCHEME              (TPM_RC)(RC_FMT1+0x012)
94   #define  TPM_RC_SIZE                 (TPM_RC)(RC_FMT1+0x015)
95   #define  TPM_RCS_SIZE                (TPM_RC)(RC_FMT1+0x015)
96   #define  TPM_RC_SYMMETRIC            (TPM_RC)(RC_FMT1+0x016)
97   #define  TPM_RCS_SYMMETRIC           (TPM_RC)(RC_FMT1+0x016)
98   #define  TPM_RC_TAG                  (TPM_RC)(RC_FMT1+0x017)
99   #define  TPM_RCS_TAG                 (TPM_RC)(RC_FMT1+0x017)
100  #define  TPM_RC_SELECTOR             (TPM_RC)(RC_FMT1+0x018)
101  #define  TPM_RCS_SELECTOR            (TPM_RC)(RC_FMT1+0x018)
102  #define  TPM_RC_INSUFFICIENT         (TPM_RC)(RC_FMT1+0x01A)
103  #define  TPM_RCS_INSUFFICIENT        (TPM_RC)(RC_FMT1+0x01A)
104  #define  TPM_RC_SIGNATURE            (TPM_RC)(RC_FMT1+0x01B)
105  #define  TPM_RCS_SIGNATURE           (TPM_RC)(RC_FMT1+0x01B)
106  #define  TPM_RC_KEY                  (TPM_RC)(RC_FMT1+0x01C)
107  #define  TPM_RCS_KEY                 (TPM_RC)(RC_FMT1+0x01C)
108  #define  TPM_RC_POLICY_FAIL          (TPM_RC)(RC_FMT1+0x01D)
109  #define  TPM_RCS_POLICY_FAIL         (TPM_RC)(RC_FMT1+0x01D)
110  #define  TPM_RC_INTEGRITY            (TPM_RC)(RC_FMT1+0x01F)
111  #define  TPM_RCS_INTEGRITY           (TPM_RC)(RC_FMT1+0x01F)
112  #define  TPM_RC_TICKET               (TPM_RC)(RC_FMT1+0x020)
113  #define  TPM_RCS_TICKET              (TPM_RC)(RC_FMT1+0x020)
114  #define  TPM_RC_RESERVED_BITS        (TPM_RC)(RC_FMT1+0x021)
115  #define  TPM_RCS_RESERVED_BITS       (TPM_RC)(RC_FMT1+0x021)
```

```
116   #define  TPM_RC_BAD_AUTH            (TPM_RC)(RC_FMT1+0x022)
117   #define  TPM_RCS_BAD_AUTH           (TPM_RC)(RC_FMT1+0x022)
118   #define  TPM_RC_EXPIRED             (TPM_RC)(RC_FMT1+0x023)
119   #define  TPM_RCS_EXPIRED            (TPM_RC)(RC_FMT1+0x023)
120   #define  TPM_RC_POLICY_CC           (TPM_RC)(RC_FMT1+0x024)
121   #define  TPM_RCS_POLICY_CC          (TPM_RC)(RC_FMT1+0x024)
122   #define  TPM_RC_BINDING             (TPM_RC)(RC_FMT1+0x025)
123   #define  TPM_RCS_BINDING            (TPM_RC)(RC_FMT1+0x025)
124   #define  TPM_RC_CURVE               (TPM_RC)(RC_FMT1+0x026)
125   #define  TPM_RCS_CURVE              (TPM_RC)(RC_FMT1+0x026)
126   #define  TPM_RC_ECC_POINT           (TPM_RC)(RC_FMT1+0x027)
127   #define  TPM_RCS_ECC_POINT          (TPM_RC)(RC_FMT1+0x027)
128   #define  RC_WARN                    (TPM_RC)(0x900)
129   #define  TPM_RC_CONTEXT_GAP         (TPM_RC)(RC_WARN+0x001)
130   #define  TPM_RC_OBJECT_MEMORY       (TPM_RC)(RC_WARN+0x002)
131   #define  TPM_RC_SESSION_MEMORY      (TPM_RC)(RC_WARN+0x003)
132   #define  TPM_RC_MEMORY              (TPM_RC)(RC_WARN+0x004)
133   #define  TPM_RC_SESSION_HANDLES     (TPM_RC)(RC_WARN+0x005)
134   #define  TPM_RC_OBJECT_HANDLES      (TPM_RC)(RC_WARN+0x006)
135   #define  TPM_RC_LOCALITY            (TPM_RC)(RC_WARN+0x007)
136   #define  TPM_RC_YIELDED             (TPM_RC)(RC_WARN+0x008)
137   #define  TPM_RC_CANCELED            (TPM_RC)(RC_WARN+0x009)
138   #define  TPM_RC_TESTING             (TPM_RC)(RC_WARN+0x00A)
139   #define  TPM_RC_REFERENCE_H0        (TPM_RC)(RC_WARN+0x010)
140   #define  TPM_RC_REFERENCE_H1        (TPM_RC)(RC_WARN+0x011)
141   #define  TPM_RC_REFERENCE_H2        (TPM_RC)(RC_WARN+0x012)
142   #define  TPM_RC_REFERENCE_H3        (TPM_RC)(RC_WARN+0x013)
143   #define  TPM_RC_REFERENCE_H4        (TPM_RC)(RC_WARN+0x014)
144   #define  TPM_RC_REFERENCE_H5        (TPM_RC)(RC_WARN+0x015)
145   #define  TPM_RC_REFERENCE_H6        (TPM_RC)(RC_WARN+0x016)
146   #define  TPM_RC_REFERENCE_S0        (TPM_RC)(RC_WARN+0x018)
147   #define  TPM_RC_REFERENCE_S1        (TPM_RC)(RC_WARN+0x019)
148   #define  TPM_RC_REFERENCE_S2        (TPM_RC)(RC_WARN+0x01A)
149   #define  TPM_RC_REFERENCE_S3        (TPM_RC)(RC_WARN+0x01B)
150   #define  TPM_RC_REFERENCE_S4        (TPM_RC)(RC_WARN+0x01C)
151   #define  TPM_RC_REFERENCE_S5        (TPM_RC)(RC_WARN+0x01D)
152   #define  TPM_RC_REFERENCE_S6        (TPM_RC)(RC_WARN+0x01E)
153   #define  TPM_RC_NV_RATE             (TPM_RC)(RC_WARN+0x020)
154   #define  TPM_RC_LOCKOUT             (TPM_RC)(RC_WARN+0x021)
155   #define  TPM_RC_RETRY               (TPM_RC)(RC_WARN+0x022)
156   #define  TPM_RC_NV_UNAVAILABLE      (TPM_RC)(RC_WARN+0x023)
157   #define  TPM_RC_NOT_USED            (TPM_RC)(RC_WARN+0x7F)
158   #define  TPM_RC_H                   (TPM_RC)(0x000)
159   #define  TPM_RC_P                   (TPM_RC)(0x040)
160   #define  TPM_RC_S                   (TPM_RC)(0x800)
161   #define  TPM_RC_1                   (TPM_RC)(0x100)
162   #define  TPM_RC_2                   (TPM_RC)(0x200)
163   #define  TPM_RC_3                   (TPM_RC)(0x300)
164   #define  TPM_RC_4                   (TPM_RC)(0x400)
165   #define  TPM_RC_5                   (TPM_RC)(0x500)
166   #define  TPM_RC_6                   (TPM_RC)(0x600)
167   #define  TPM_RC_7                   (TPM_RC)(0x700)
168   #define  TPM_RC_8                   (TPM_RC)(0x800)
169   #define  TPM_RC_9                   (TPM_RC)(0x900)
170   #define  TPM_RC_A                   (TPM_RC)(0xA00)
171   #define  TPM_RC_B                   (TPM_RC)(0xB00)
172   #define  TPM_RC_C                   (TPM_RC)(0xC00)
173   #define  TPM_RC_D                   (TPM_RC)(0xD00)
174   #define  TPM_RC_E                   (TPM_RC)(0xE00)
175   #define  TPM_RC_F                   (TPM_RC)(0xF00)
176   #define  TPM_RC_N_MASK              (TPM_RC)(0xF00)
```

Table 2:17 - Definition of TPM_CLOCK_ADJUST Constants *(EnumTable())*

```
177   typedef  INT8              TPM_CLOCK_ADJUST;
```

```
178   #define  TPM_CLOCK_COARSE_SLOWER    (TPM_CLOCK_ADJUST)(-3)
179   #define  TPM_CLOCK_MEDIUM_SLOWER    (TPM_CLOCK_ADJUST)(-2)
180   #define  TPM_CLOCK_FINE_SLOWER      (TPM_CLOCK_ADJUST)(-1)
181   #define  TPM_CLOCK_NO_CHANGE        (TPM_CLOCK_ADJUST)(0)
182   #define  TPM_CLOCK_FINE_FASTER      (TPM_CLOCK_ADJUST)(1)
183   #define  TPM_CLOCK_MEDIUM_FASTER    (TPM_CLOCK_ADJUST)(2)
184   #define  TPM_CLOCK_COARSE_FASTER    (TPM_CLOCK_ADJUST)(3)
```

Table 2:18 - Definition of TPM_EO Constants *(EnumTable())*

```
185   typedef  UINT16             TPM_EO;
186   #define  TPM_EO_EQ             (TPM_EO)(0x0000)
187   #define  TPM_EO_NEQ            (TPM_EO)(0x0001)
188   #define  TPM_EO_SIGNED_GT      (TPM_EO)(0x0002)
189   #define  TPM_EO_UNSIGNED_GT    (TPM_EO)(0x0003)
190   #define  TPM_EO_SIGNED_LT      (TPM_EO)(0x0004)
191   #define  TPM_EO_UNSIGNED_LT    (TPM_EO)(0x0005)
192   #define  TPM_EO_SIGNED_GE      (TPM_EO)(0x0006)
193   #define  TPM_EO_UNSIGNED_GE    (TPM_EO)(0x0007)
194   #define  TPM_EO_SIGNED_LE      (TPM_EO)(0x0008)
195   #define  TPM_EO_UNSIGNED_LE    (TPM_EO)(0x0009)
196   #define  TPM_EO_BITSET         (TPM_EO)(0x000A)
197   #define  TPM_EO_BITCLEAR       (TPM_EO)(0x000B)
```

Table 2:19 - Definition of TPM_ST Constants *(EnumTable())*

```
198   typedef  UINT16             TPM_ST;
199   #define  TPM_ST_RSP_COMMAND             (TPM_ST)(0x00C4)
200   #define  TPM_ST_NULL                    (TPM_ST)(0X8000)
201   #define  TPM_ST_NO_SESSIONS             (TPM_ST)(0x8001)
202   #define  TPM_ST_SESSIONS                (TPM_ST)(0x8002)
203   #define  TPM_ST_ATTEST_NV               (TPM_ST)(0x8014)
204   #define  TPM_ST_ATTEST_COMMAND_AUDIT    (TPM_ST)(0x8015)
205   #define  TPM_ST_ATTEST_SESSION_AUDIT    (TPM_ST)(0x8016)
206   #define  TPM_ST_ATTEST_CERTIFY          (TPM_ST)(0x8017)
207   #define  TPM_ST_ATTEST_QUOTE            (TPM_ST)(0x8018)
208   #define  TPM_ST_ATTEST_TIME             (TPM_ST)(0x8019)
209   #define  TPM_ST_ATTEST_CREATION         (TPM_ST)(0x801A)
210   #define  TPM_ST_CREATION                (TPM_ST)(0x8021)
211   #define  TPM_ST_VERIFIED                (TPM_ST)(0x8022)
212   #define  TPM_ST_AUTH_SECRET             (TPM_ST)(0x8023)
213   #define  TPM_ST_HASHCHECK               (TPM_ST)(0x8024)
214   #define  TPM_ST_AUTH_SIGNED             (TPM_ST)(0x8025)
215   #define  TPM_ST_FU_MANIFEST             (TPM_ST)(0x8029)
```

Table 2:20 - Definition of TPM_SU Constants *(EnumTable())*

```
216   typedef  UINT16             TPM_SU;
217   #define  TPM_SU_CLEAR    (TPM_SU)(0x0000)
218   #define  TPM_SU_STATE    (TPM_SU)(0x0001)
```

Table 2:21 - Definition of TPM_SE Constants *(EnumTable())*

```
219   typedef  UINT8              TPM_SE;
220   #define  TPM_SE_HMAC      (TPM_SE)(0x00)
221   #define  TPM_SE_POLICY    (TPM_SE)(0x01)
222   #define  TPM_SE_TRIAL     (TPM_SE)(0x03)
```

Table 2:22 - Definition of TPM_CAP Constants *(EnumTable())*

```
223   typedef  UINT32             TPM_CAP;
224   #define  TPM_CAP_FIRST              (TPM_CAP)(0x00000000)
225   #define  TPM_CAP_ALGS               (TPM_CAP)(0x00000000)
```

```
226   #define  TPM_CAP_HANDLES           (TPM_CAP)(0x00000001)
227   #define  TPM_CAP_COMMANDS          (TPM_CAP)(0x00000002)
228   #define  TPM_CAP_PP_COMMANDS       (TPM_CAP)(0x00000003)
229   #define  TPM_CAP_AUDIT_COMMANDS    (TPM_CAP)(0x00000004)
230   #define  TPM_CAP_PCRS              (TPM_CAP)(0x00000005)
231   #define  TPM_CAP_TPM_PROPERTIES    (TPM_CAP)(0x00000006)
232   #define  TPM_CAP_PCR_PROPERTIES    (TPM_CAP)(0x00000007)
233   #define  TPM_CAP_ECC_CURVES        (TPM_CAP)(0x00000008)
234   #define  TPM_CAP_AUTH_POLICIES     (TPM_CAP)(0x00000009)
235   #define  TPM_CAP_LAST              (TPM_CAP)(0x00000009)
236   #define  TPM_CAP_VENDOR_PROPERTY   (TPM_CAP)(0x00000100)
```

Table 2:23 - Definition of TPM_PT Constants *(EnumTable())*

```
237   typedef  UINT32            TPM_PT;
238   #define  TPM_PT_NONE                   (TPM_PT)(0x00000000)
239   #define  PT_GROUP                      (TPM_PT)(0x00000100)
240   #define  PT_FIXED                      (TPM_PT)(PT_GROUP*1)
241   #define  TPM_PT_FAMILY_INDICATOR       (TPM_PT)(PT_FIXED+0)
242   #define  TPM_PT_LEVEL                  (TPM_PT)(PT_FIXED+1)
243   #define  TPM_PT_REVISION               (TPM_PT)(PT_FIXED+2)
244   #define  TPM_PT_DAY_OF_YEAR            (TPM_PT)(PT_FIXED+3)
245   #define  TPM_PT_YEAR                   (TPM_PT)(PT_FIXED+4)
246   #define  TPM_PT_MANUFACTURER           (TPM_PT)(PT_FIXED+5)
247   #define  TPM_PT_VENDOR_STRING_1        (TPM_PT)(PT_FIXED+6)
248   #define  TPM_PT_VENDOR_STRING_2        (TPM_PT)(PT_FIXED+7)
249   #define  TPM_PT_VENDOR_STRING_3        (TPM_PT)(PT_FIXED+8)
250   #define  TPM_PT_VENDOR_STRING_4        (TPM_PT)(PT_FIXED+9)
251   #define  TPM_PT_VENDOR_TPM_TYPE        (TPM_PT)(PT_FIXED+10)
252   #define  TPM_PT_FIRMWARE_VERSION_1     (TPM_PT)(PT_FIXED+11)
253   #define  TPM_PT_FIRMWARE_VERSION_2     (TPM_PT)(PT_FIXED+12)
254   #define  TPM_PT_INPUT_BUFFER           (TPM_PT)(PT_FIXED+13)
255   #define  TPM_PT_HR_TRANSIENT_MIN       (TPM_PT)(PT_FIXED+14)
256   #define  TPM_PT_HR_PERSISTENT_MIN      (TPM_PT)(PT_FIXED+15)
257   #define  TPM_PT_HR_LOADED_MIN          (TPM_PT)(PT_FIXED+16)
258   #define  TPM_PT_ACTIVE_SESSIONS_MAX    (TPM_PT)(PT_FIXED+17)
259   #define  TPM_PT_PCR_COUNT              (TPM_PT)(PT_FIXED+18)
260   #define  TPM_PT_PCR_SELECT_MIN         (TPM_PT)(PT_FIXED+19)
261   #define  TPM_PT_CONTEXT_GAP_MAX        (TPM_PT)(PT_FIXED+20)
262   #define  TPM_PT_NV_COUNTERS_MAX        (TPM_PT)(PT_FIXED+22)
263   #define  TPM_PT_NV_INDEX_MAX           (TPM_PT)(PT_FIXED+23)
264   #define  TPM_PT_MEMORY                 (TPM_PT)(PT_FIXED+24)
265   #define  TPM_PT_CLOCK_UPDATE           (TPM_PT)(PT_FIXED+25)
266   #define  TPM_PT_CONTEXT_HASH           (TPM_PT)(PT_FIXED+26)
267   #define  TPM_PT_CONTEXT_SYM            (TPM_PT)(PT_FIXED+27)
268   #define  TPM_PT_CONTEXT_SYM_SIZE       (TPM_PT)(PT_FIXED+28)
269   #define  TPM_PT_ORDERLY_COUNT          (TPM_PT)(PT_FIXED+29)
270   #define  TPM_PT_MAX_COMMAND_SIZE       (TPM_PT)(PT_FIXED+30)
271   #define  TPM_PT_MAX_RESPONSE_SIZE      (TPM_PT)(PT_FIXED+31)
272   #define  TPM_PT_MAX_DIGEST             (TPM_PT)(PT_FIXED+32)
273   #define  TPM_PT_MAX_OBJECT_CONTEXT     (TPM_PT)(PT_FIXED+33)
274   #define  TPM_PT_MAX_SESSION_CONTEXT    (TPM_PT)(PT_FIXED+34)
275   #define  TPM_PT_PS_FAMILY_INDICATOR    (TPM_PT)(PT_FIXED+35)
276   #define  TPM_PT_PS_LEVEL               (TPM_PT)(PT_FIXED+36)
277   #define  TPM_PT_PS_REVISION            (TPM_PT)(PT_FIXED+37)
278   #define  TPM_PT_PS_DAY_OF_YEAR         (TPM_PT)(PT_FIXED+38)
279   #define  TPM_PT_PS_YEAR                (TPM_PT)(PT_FIXED+39)
280   #define  TPM_PT_SPLIT_MAX              (TPM_PT)(PT_FIXED+40)
281   #define  TPM_PT_TOTAL_COMMANDS         (TPM_PT)(PT_FIXED+41)
282   #define  TPM_PT_LIBRARY_COMMANDS       (TPM_PT)(PT_FIXED+42)
283   #define  TPM_PT_VENDOR_COMMANDS        (TPM_PT)(PT_FIXED+43)
284   #define  TPM_PT_NV_BUFFER_MAX          (TPM_PT)(PT_FIXED+44)
285   #define  TPM_PT_MODES                  (TPM_PT)(PT_FIXED+45)
286   #define  TPM_PT_MAX_CAP_BUFFER         (TPM_PT)(PT_FIXED+46)
287   #define  PT_VAR                        (TPM_PT)(PT_GROUP*2)
```

```
288   #define  TPM_PT_PERMANENT             (TPM_PT)(PT_VAR+0)
289   #define  TPM_PT_STARTUP_CLEAR         (TPM_PT)(PT_VAR+1)
290   #define  TPM_PT_HR_NV_INDEX           (TPM_PT)(PT_VAR+2)
291   #define  TPM_PT_HR_LOADED             (TPM_PT)(PT_VAR+3)
292   #define  TPM_PT_HR_LOADED_AVAIL       (TPM_PT)(PT_VAR+4)
293   #define  TPM_PT_HR_ACTIVE             (TPM_PT)(PT_VAR+5)
294   #define  TPM_PT_HR_ACTIVE_AVAIL       (TPM_PT)(PT_VAR+6)
295   #define  TPM_PT_HR_TRANSIENT_AVAIL    (TPM_PT)(PT_VAR+7)
296   #define  TPM_PT_HR_PERSISTENT         (TPM_PT)(PT_VAR+8)
297   #define  TPM_PT_HR_PERSISTENT_AVAIL   (TPM_PT)(PT_VAR+9)
298   #define  TPM_PT_NV_COUNTERS           (TPM_PT)(PT_VAR+10)
299   #define  TPM_PT_NV_COUNTERS_AVAIL     (TPM_PT)(PT_VAR+11)
300   #define  TPM_PT_ALGORITHM_SET         (TPM_PT)(PT_VAR+12)
301   #define  TPM_PT_LOADED_CURVES         (TPM_PT)(PT_VAR+13)
302   #define  TPM_PT_LOCKOUT_COUNTER       (TPM_PT)(PT_VAR+14)
303   #define  TPM_PT_MAX_AUTH_FAIL         (TPM_PT)(PT_VAR+15)
304   #define  TPM_PT_LOCKOUT_INTERVAL      (TPM_PT)(PT_VAR+16)
305   #define  TPM_PT_LOCKOUT_RECOVERY      (TPM_PT)(PT_VAR+17)
306   #define  TPM_PT_NV_WRITE_RECOVERY     (TPM_PT)(PT_VAR+18)
307   #define  TPM_PT_AUDIT_COUNTER_0       (TPM_PT)(PT_VAR+19)
308   #define  TPM_PT_AUDIT_COUNTER_1       (TPM_PT)(PT_VAR+20)
```

Table 2:24 - Definition of TPM_PT_PCR Constants *(EnumTable())*

```
309   typedef  UINT32            TPM_PT_PCR;
310   #define  TPM_PT_PCR_FIRST          (TPM_PT_PCR)(0x00000000)
311   #define  TPM_PT_PCR_SAVE           (TPM_PT_PCR)(0x00000000)
312   #define  TPM_PT_PCR_EXTEND_L0      (TPM_PT_PCR)(0x00000001)
313   #define  TPM_PT_PCR_RESET_L0       (TPM_PT_PCR)(0x00000002)
314   #define  TPM_PT_PCR_EXTEND_L1      (TPM_PT_PCR)(0x00000003)
315   #define  TPM_PT_PCR_RESET_L1       (TPM_PT_PCR)(0x00000004)
316   #define  TPM_PT_PCR_EXTEND_L2      (TPM_PT_PCR)(0x00000005)
317   #define  TPM_PT_PCR_RESET_L2       (TPM_PT_PCR)(0x00000006)
318   #define  TPM_PT_PCR_EXTEND_L3      (TPM_PT_PCR)(0x00000007)
319   #define  TPM_PT_PCR_RESET_L3       (TPM_PT_PCR)(0x00000008)
320   #define  TPM_PT_PCR_EXTEND_L4      (TPM_PT_PCR)(0x00000009)
321   #define  TPM_PT_PCR_RESET_L4       (TPM_PT_PCR)(0x0000000A)
322   #define  TPM_PT_PCR_NO_INCREMENT   (TPM_PT_PCR)(0x00000011)
323   #define  TPM_PT_PCR_DRTM_RESET     (TPM_PT_PCR)(0x00000012)
324   #define  TPM_PT_PCR_POLICY         (TPM_PT_PCR)(0x00000013)
325   #define  TPM_PT_PCR_AUTH           (TPM_PT_PCR)(0x00000014)
326   #define  TPM_PT_PCR_LAST           (TPM_PT_PCR)(0x00000014)
```

Table 2:25 - Definition of TPM_PS Constants *(EnumTable())*

```
327   typedef  UINT32            TPM_PS;
328   #define  TPM_PS_MAIN               (TPM_PS)(0x00000000)
329   #define  TPM_PS_PC                 (TPM_PS)(0x00000001)
330   #define  TPM_PS_PDA                (TPM_PS)(0x00000002)
331   #define  TPM_PS_CELL_PHONE         (TPM_PS)(0x00000003)
332   #define  TPM_PS_SERVER             (TPM_PS)(0x00000004)
333   #define  TPM_PS_PERIPHERAL         (TPM_PS)(0x00000005)
334   #define  TPM_PS_TSS                (TPM_PS)(0x00000006)
335   #define  TPM_PS_STORAGE            (TPM_PS)(0x00000007)
336   #define  TPM_PS_AUTHENTICATION     (TPM_PS)(0x00000008)
337   #define  TPM_PS_EMBEDDED           (TPM_PS)(0x00000009)
338   #define  TPM_PS_HARDCOPY           (TPM_PS)(0x0000000A)
339   #define  TPM_PS_INFRASTRUCTURE     (TPM_PS)(0x0000000B)
340   #define  TPM_PS_VIRTUALIZATION     (TPM_PS)(0x0000000C)
341   #define  TPM_PS_TNC                (TPM_PS)(0x0000000D)
342   #define  TPM_PS_MULTI_TENANT       (TPM_PS)(0x0000000E)
343   #define  TPM_PS_TC                 (TPM_PS)(0x0000000F)
```

Table 2:26 - Definition of Types for Handles *(TypedefTable())*

```
344   typedef  UINT32            TPM_HANDLE;
```

Table 2:27 - Definition of TPM_HT Constants *(EnumTable())*

```
345   typedef  UINT8             TPM_HT;
346   #define  TPM_HT_PCR              (TPM_HT)(0x00)
347   #define  TPM_HT_NV_INDEX         (TPM_HT)(0x01)
348   #define  TPM_HT_HMAC_SESSION     (TPM_HT)(0x02)
349   #define  TPM_HT_LOADED_SESSION   (TPM_HT)(0x02)
350   #define  TPM_HT_POLICY_SESSION   (TPM_HT)(0x03)
351   #define  TPM_HT_SAVED_SESSION    (TPM_HT)(0x03)
352   #define  TPM_HT_PERMANENT        (TPM_HT)(0x40)
353   #define  TPM_HT_TRANSIENT        (TPM_HT)(0x80)
354   #define  TPM_HT_PERSISTENT       (TPM_HT)(0x81)
```

Table 2:28 - Definition of TPM_RH Constants *(EnumTable())*

```
355   typedef  TPM_HANDLE        TPM_RH;
356   #define  TPM_RH_FIRST          (TPM_RH)(0x40000000)
357   #define  TPM_RH_SRK            (TPM_RH)(0x40000000)
358   #define  TPM_RH_OWNER          (TPM_RH)(0x40000001)
359   #define  TPM_RH_REVOKE         (TPM_RH)(0x40000002)
360   #define  TPM_RH_TRANSPORT      (TPM_RH)(0x40000003)
361   #define  TPM_RH_OPERATOR       (TPM_RH)(0x40000004)
362   #define  TPM_RH_ADMIN          (TPM_RH)(0x40000005)
363   #define  TPM_RH_EK             (TPM_RH)(0x40000006)
364   #define  TPM_RH_NULL           (TPM_RH)(0x40000007)
365   #define  TPM_RH_UNASSIGNED     (TPM_RH)(0x40000008)
366   #define  TPM_RS_PW             (TPM_RH)(0x40000009)
367   #define  TPM_RH_LOCKOUT        (TPM_RH)(0x4000000A)
368   #define  TPM_RH_ENDORSEMENT    (TPM_RH)(0x4000000B)
369   #define  TPM_RH_PLATFORM       (TPM_RH)(0x4000000C)
370   #define  TPM_RH_PLATFORM_NV    (TPM_RH)(0x4000000D)
371   #define  TPM_RH_AUTH_00        (TPM_RH)(0x40000010)
372   #define  TPM_RH_AUTH_FF        (TPM_RH)(0x4000010F)
373   #define  TPM_RH_LAST           (TPM_RH)(0x4000010F)
```

Table 2:29 - Definition of TPM_HC Constants *(EnumTable())*

```
374   typedef  TPM_HANDLE        TPM_HC;
375   #define  HR_HANDLE_MASK          (TPM_HC)(0x00FFFFFF)
376   #define  HR_RANGE_MASK           (TPM_HC)(0xFF000000)
377   #define  HR_SHIFT                (TPM_HC)(24)
378   #define  HR_PCR                  (TPM_HC)((TPM_HT_PCR<<HR_SHIFT))
379   #define  HR_HMAC_SESSION         (TPM_HC)((TPM_HT_HMAC_SESSION<<HR_SHIFT))
380   #define  HR_POLICY_SESSION       (TPM_HC)((TPM_HT_POLICY_SESSION<<HR_SHIFT))
381   #define  HR_TRANSIENT            (TPM_HC)((TPM_HT_TRANSIENT<<HR_SHIFT))
382   #define  HR_PERSISTENT           (TPM_HC)((TPM_HT_PERSISTENT<<HR_SHIFT))
383   #define  HR_NV_INDEX             (TPM_HC)((TPM_HT_NV_INDEX<<HR_SHIFT))
384   #define  HR_PERMANENT            (TPM_HC)((TPM_HT_PERMANENT<<HR_SHIFT))
385   #define  PCR_FIRST               (TPM_HC)((HR_PCR+0))
386   #define  PCR_LAST                (TPM_HC)((PCR_FIRST+IMPLEMENTATION_PCR-1))
387   #define  HMAC_SESSION_FIRST      (TPM_HC)((HR_HMAC_SESSION+0))
388   #define  HMAC_SESSION_LAST       \
389       (TPM_HC)((HMAC_SESSION_FIRST + MAX_ACTIVE_SESSIONS-1))
390   #define  LOADED_SESSION_FIRST    (TPM_HC)(HMAC_SESSION_FIRST)
391   #define  LOADED_SESSION_LAST     (TPM_HC)(HMAC_SESSION_LAST)
392   #define  POLICY_SESSION_FIRST    (TPM_HC)((HR_POLICY_SESSION+0))
393   #define  POLICY_SESSION_LAST     \
394       (TPM_HC)((POLICY_SESSION_FIRST + MAX_ACTIVE_SESSIONS-1))
395   #define  TRANSIENT_FIRST         (TPM_HC)((HR_TRANSIENT+0))
396   #define  ACTIVE_SESSION_FIRST    (TPM_HC)(POLICY_SESSION_FIRST)
397   #define  ACTIVE_SESSION_LAST     (TPM_HC)(POLICY_SESSION_LAST)
398   #define  TRANSIENT_LAST          (TPM_HC)((TRANSIENT_FIRST+MAX_LOADED_OBJECTS-1))
```

```
399   #define  PERSISTENT_FIRST        (TPM_HC)((HR_PERSISTENT+0))
400   #define  PERSISTENT_LAST         (TPM_HC)((PERSISTENT_FIRST+0x00FFFFFF))
401   #define  PLATFORM_PERSISTENT     (TPM_HC)((PERSISTENT_FIRST+0x00800000))
402   #define  NV_INDEX_FIRST          (TPM_HC)((HR_NV_INDEX+0))
403   #define  NV_INDEX_LAST           (TPM_HC)((NV_INDEX_FIRST+0x00FFFFFF))
404   #define  PERMANENT_FIRST         (TPM_HC)(TPM_RH_FIRST)
405   #define  PERMANENT_LAST          (TPM_HC)(TPM_RH_LAST)
```

Table 2:30 - Definition of TPMA_ALGORITHM Bits *(BitsTable())*

```
406   typedef struct {
407       unsigned    asymmetric         : 1 ;
408       unsigned    symmetric          : 1 ;
409       unsigned    hash               : 1 ;
410       unsigned    object             : 1 ;
411       unsigned    Reserved_at_bit_4  : 4 ;
412       unsigned    signing            : 1 ;
413       unsigned    encrypting         : 1 ;
414       unsigned    method             : 1 ;
415       unsigned    Reserved_at_bit_11 : 21;
416   } TPMA_ALGORITHM;
417   #define IsAlgorithm_asymmetric(attribute)                \
418           ((attribute.asymmetric) != 0)
419   #define IsAlgorithm_symmetric(attribute)                 \
420           ((attribute.symmetric) != 0)
421   #define IsAlgorithm_hash(attribute)                      \
422           ((attribute.hash) != 0)
423   #define IsAlgorithm_object(attribute)                    \
424           ((attribute.object) != 0)
425   #define IsAlgorithm_signing(attribute)                   \
426           ((attribute.signing) != 0)
427   #define IsAlgorithm_encrypting(attribute)                \
428           ((attribute.encrypting) != 0)
429   #define IsAlgorithm_method(attribute)                    \
430           ((attribute.method) != 0)
```

Table 2:31 - Definition of TPMA_OBJECT Bits *(BitsTable())*

```
431   typedef struct {
432       unsigned    Reserved_at_bit_0      : 1 ;
433       unsigned    fixedTPM               : 1 ;
434       unsigned    stClear                : 1 ;
435       unsigned    Reserved_at_bit_3      : 1 ;
436       unsigned    fixedParent            : 1 ;
437       unsigned    sensitiveDataOrigin    : 1 ;
438       unsigned    userWithAuth           : 1 ;
439       unsigned    adminWithPolicy        : 1 ;
440       unsigned    Reserved_at_bit_8      : 2 ;
441       unsigned    noDA                   : 1 ;
442       unsigned    encryptedDuplication   : 1 ;
443       unsigned    Reserved_at_bit_12     : 4 ;
444       unsigned    restricted             : 1 ;
445       unsigned    decrypt                : 1 ;
446       unsigned    sign                   : 1 ;
447       unsigned    Reserved_at_bit_19     : 13;
448   } TPMA_OBJECT;
449   #define IsObject_fixedTPM(attribute)                     \
450           ((attribute.fixedTPM) != 0)
451   #define IsObject_stClear(attribute)                      \
452           ((attribute.stClear) != 0)
453   #define IsObject_fixedParent(attribute)                  \
454           ((attribute.fixedParent) != 0)
455   #define IsObject_sensitiveDataOrigin(attribute)          \
456           ((attribute.sensitiveDataOrigin) != 0)
457   #define IsObject_userWithAuth(attribute)                 \
```

```
458             ((attribute.userWithAuth) != 0)
459     #define IsObject_adminWithPolicy(attribute)                 \
460             ((attribute.adminWithPolicy) != 0)
461     #define IsObject_noDA(attribute)                            \
462             ((attribute.noDA) != 0)
463     #define IsObject_encryptedDuplication(attribute)            \
464             ((attribute.encryptedDuplication) != 0)
465     #define IsObject_restricted(attribute)                      \
466             ((attribute.restricted) != 0)
467     #define IsObject_decrypt(attribute)                         \
468             ((attribute.decrypt) != 0)
469     #define IsObject_sign(attribute)                            \
470             ((attribute.sign) != 0)
471     #define IsObject_sign(attribute)                            \
472             ((attribute.sign) != 0)
```

Table 2:32 - Definition of TPMA_SESSION Bits *(BitsTable())*

```
473     typedef struct {
474         unsigned    continueSession   : 1 ;
475         unsigned    auditExclusive    : 1 ;
476         unsigned    auditReset        : 1 ;
477         unsigned    Reserved_at_bit_3 : 2 ;
478         unsigned    decrypt           : 1 ;
479         unsigned    encrypt           : 1 ;
480         unsigned    audit             : 1 ;
481     } TPMA_SESSION;
482     #define IsSession_continueSession(attribute)                \
483             ((attribute.continueSession) != 0)
484     #define IsSession_auditExclusive(attribute)                 \
485             ((attribute.auditExclusive) != 0)
486     #define IsSession_auditReset(attribute)                     \
487             ((attribute.auditReset) != 0)
488     #define IsSession_decrypt(attribute)                        \
489             ((attribute.decrypt) != 0)
490     #define IsSession_encrypt(attribute)                        \
491             ((attribute.encrypt) != 0)
492     #define IsSession_audit(attribute)                          \
493             ((attribute.audit) != 0)
```

Table 2:33 - Definition of TPMA_LOCALITY Bits *(BitsTable())*

```
494     typedef struct {
495         unsigned    TPM_LOC_ZERO    : 1 ;
496         unsigned    TPM_LOC_ONE     : 1 ;
497         unsigned    TPM_LOC_TWO     : 1 ;
498         unsigned    TPM_LOC_THREE   : 1 ;
499         unsigned    TPM_LOC_FOUR    : 1 ;
500         unsigned    Extended        : 3 ;
501     } TPMA_LOCALITY;
502     #define IsLocality_TPM_LOC_ZERO(attribute)                  \
503             ((attribute.TPM_LOC_ZERO) != 0)
504     #define IsLocality_TPM_LOC_ONE(attribute)                   \
505             ((attribute.TPM_LOC_ONE) != 0)
506     #define IsLocality_TPM_LOC_TWO(attribute)                   \
507             ((attribute.TPM_LOC_TWO) != 0)
508     #define IsLocality_TPM_LOC_THREE(attribute)                 \
509             ((attribute.TPM_LOC_THREE) != 0)
510     #define IsLocality_TPM_LOC_FOUR(attribute)                  \
511             ((attribute.TPM_LOC_FOUR) != 0)
```

Table 2:34 - Definition of TPMA_PERMANENT Bits *(BitsTable())*

```
512     typedef struct {
```

```
513         unsigned    ownerAuthSet         : 1 ;
514         unsigned    endorsementAuthSet   : 1 ;
515         unsigned    lockoutAuthSet       : 1 ;
516         unsigned    Reserved_at_bit_3    : 5 ;
517         unsigned    disableClear         : 1 ;
518         unsigned    inLockout            : 1 ;
519         unsigned    tpmGeneratedEPS      : 1 ;
520         unsigned    Reserved_at_bit_11   : 21;
521     } TPMA_PERMANENT;
522     #define IsPermanent_ownerAuthSet(attribute)                \
523             ((attribute.ownerAuthSet) != 0)
524     #define IsPermanent_endorsementAuthSet(attribute)          \
525             ((attribute.endorsementAuthSet) != 0)
526     #define IsPermanent_lockoutAuthSet(attribute)              \
527             ((attribute.lockoutAuthSet) != 0)
528     #define IsPermanent_disableClear(attribute)                \
529             ((attribute.disableClear) != 0)
530     #define IsPermanent_inLockout(attribute)                   \
531             ((attribute.inLockout) != 0)
532     #define IsPermanent_tpmGeneratedEPS(attribute)             \
533             ((attribute.tpmGeneratedEPS) != 0)
```

Table 2:35 - Definition of TPMA_STARTUP_CLEAR Bits *(BitsTable())*

```
534     typedef struct {
535         unsigned    phEnable          : 1 ;
536         unsigned    shEnable          : 1 ;
537         unsigned    ehEnable          : 1 ;
538         unsigned    phEnableNV        : 1 ;
539         unsigned    Reserved_at_bit_4 : 27;
540         unsigned    orderly           : 1 ;
541     } TPMA_STARTUP_CLEAR;
542     #define IsStartupClear_phEnable(attribute)                 \
543             ((attribute.phEnable) != 0)
544     #define IsStartupClear_shEnable(attribute)                 \
545             ((attribute.shEnable) != 0)
546     #define IsStartupClear_ehEnable(attribute)                 \
547             ((attribute.ehEnable) != 0)
548     #define IsStartupClear_phEnableNV(attribute)               \
549             ((attribute.phEnableNV) != 0)
550     #define IsStartupClear_orderly(attribute)                  \
551             ((attribute.orderly) != 0)
```

Table 2:36 - Definition of TPMA_MEMORY Bits *(BitsTable())*

```
552     typedef struct {
553         unsigned    sharedRAM           : 1 ;
554         unsigned    sharedNV            : 1 ;
555         unsigned    objectCopiedToRam   : 1 ;
556         unsigned    Reserved_at_bit_3   : 29;
557     } TPMA_MEMORY;
558     #define IsMemory_sharedRAM(attribute)                      \
559             ((attribute.sharedRAM) != 0)
560     #define IsMemory_sharedNV(attribute)                       \
561             ((attribute.sharedNV) != 0)
562     #define IsMemory_objectCopiedToRam(attribute)              \
563             ((attribute.objectCopiedToRam) != 0)
```

Table 2:37 - Definition of TPMA_CC Bits *(BitsTable())*

```
564     typedef struct {
565         unsigned    commandIndex       : 16;
566         unsigned    Reserved_at_bit_16 : 6 ;
567         unsigned    nv                 : 1 ;
```

```
568         unsigned    extensive          : 1 ;
569         unsigned    flushed            : 1 ;
570         unsigned    cHandles           : 3 ;
571         unsigned    rHandle            : 1 ;
572         unsigned    V                  : 1 ;
573         unsigned    Res                : 2 ;
574     } TPMA_CC;
575     #define IsCc_nv(attribute)                                  \
576             ((attribute.nv) != 0)
577     #define IsCc_extensive(attribute)                           \
578             ((attribute.extensive) != 0)
579     #define IsCc_flushed(attribute)                             \
580             ((attribute.flushed) != 0)
581     #define IsCc_rHandle(attribute)                             \
582             ((attribute.rHandle) != 0)
583     #define IsCc_V(attribute)                                   \
584             ((attribute.V) != 0)
```

Table 2:38 - Definition of TPMA_MODES Bits *(BitsTable())*

```
585     typedef struct {
586         unsigned    FIPS_140_2        : 1 ;
587         unsigned    Reserved_at_bit_1 : 31;
588     } TPMA_MODES;
589     #define IsModes_FIPS_140_2(attribute)                       \
590             ((attribute.FIPS_140_2) != 0)
```

Table 2:39 - Definition of TPMI_YES_NO Type *(InterfaceTable())*

```
591     typedef  BYTE               TPMI_YES_NO;
```

Table 2:40 - Definition of TPMI_DH_OBJECT Type *(InterfaceTable())*

```
592     typedef  TPM_HANDLE         TPMI_DH_OBJECT;
```

Table 2:41 - Definition of TPMI_DH_PARENT Type *(InterfaceTable())*

```
593     typedef  TPM_HANDLE         TPMI_DH_PARENT;
```

Table 2:42 - Definition of TPMI_DH_PERSISTENT Type *(InterfaceTable())*

```
594     typedef  TPM_HANDLE         TPMI_DH_PERSISTENT;
```

Table 2:43 - Definition of TPMI_DH_ENTITY Type *(InterfaceTable())*

```
595     typedef  TPM_HANDLE         TPMI_DH_ENTITY;
```

Table 2:44 - Definition of TPMI_DH_PCR Type *(InterfaceTable())*

```
596     typedef  TPM_HANDLE         TPMI_DH_PCR;
```

Table 2:45 - Definition of TPMI_SH_AUTH_SESSION Type *(InterfaceTable())*

```
597     typedef  TPM_HANDLE         TPMI_SH_AUTH_SESSION;
```

Table 2:46 - Definition of TPMI_SH_HMAC Type *(InterfaceTable())*

```
598     typedef  TPM_HANDLE         TPMI_SH_HMAC;
```

Table 2:47 - Definition of TPMI_SH_POLICY Type *(InterfaceTable())*

```
599   typedef   TPM_HANDLE          TPMI_SH_POLICY;
```

Table 2:48 - Definition of TPMI_DH_CONTEXT Type *(InterfaceTable())*

```
600   typedef   TPM_HANDLE          TPMI_DH_CONTEXT;
```

Table 2:49 - Definition of TPMI_RH_HIERARCHY Type *(InterfaceTable())*

```
601   typedef   TPM_HANDLE          TPMI_RH_HIERARCHY;
```

Table 2:50 - Definition of TPMI_RH_ENABLES Type *(InterfaceTable())*

```
602   typedef   TPM_HANDLE          TPMI_RH_ENABLES;
```

Table 2:51 - Definition of TPMI_RH_HIERARCHY_AUTH Type *(InterfaceTable())*

```
603   typedef   TPM_HANDLE          TPMI_RH_HIERARCHY_AUTH;
```

Table 2:52 - Definition of TPMI_RH_PLATFORM Type *(InterfaceTable())*

```
604   typedef   TPM_HANDLE          TPMI_RH_PLATFORM;
```

Table 2:53 - Definition of TPMI_RH_OWNER Type *(InterfaceTable())*

```
605   typedef   TPM_HANDLE          TPMI_RH_OWNER;
```

Table 2:54 - Definition of TPMI_RH_ENDORSEMENT Type *(InterfaceTable())*

```
606   typedef   TPM_HANDLE          TPMI_RH_ENDORSEMENT;
```

Table 2:55 - Definition of TPMI_RH_PROVISION Type *(InterfaceTable())*

```
607   typedef   TPM_HANDLE          TPMI_RH_PROVISION;
```

Table 2:56 - Definition of TPMI_RH_CLEAR Type *(InterfaceTable())*

```
608   typedef   TPM_HANDLE          TPMI_RH_CLEAR;
```

Table 2:57 - Definition of TPMI_RH_NV_AUTH Type *(InterfaceTable())*

```
609   typedef   TPM_HANDLE          TPMI_RH_NV_AUTH;
```

Table 2:58 - Definition of TPMI_RH_LOCKOUT Type *(InterfaceTable())*

```
610   typedef   TPM_HANDLE          TPMI_RH_LOCKOUT;
```

Table 2:59 - Definition of TPMI_RH_NV_INDEX Type *(InterfaceTable())*

```
611   typedef   TPM_HANDLE          TPMI_RH_NV_INDEX;
```

Table 2:60 - Definition of TPMI_ALG_HASH Type *(InterfaceTable())*

```
612   typedef   TPM_ALG_ID          TPMI_ALG_HASH;
```

Table 2:61 - Definition of TPMI_ALG_ASYM Type *(InterfaceTable())*

```
613   typedef   TPM_ALG_ID          TPMI_ALG_ASYM;
```

Table 2:62 - Definition of TPMI_ALG_SYM Type *(InterfaceTable())*

```
614    typedef  TPM_ALG_ID          TPMI_ALG_SYM;
```

Table 2:63 - Definition of TPMI_ALG_SYM_OBJECT Type *(InterfaceTable())*

```
615    typedef  TPM_ALG_ID          TPMI_ALG_SYM_OBJECT;
```

Table 2:64 - Definition of TPMI_ALG_SYM_MODE Type *(InterfaceTable())*

```
616    typedef  TPM_ALG_ID          TPMI_ALG_SYM_MODE;
```

Table 2:65 - Definition of TPMI_ALG_KDF Type *(InterfaceTable())*

```
617    typedef  TPM_ALG_ID          TPMI_ALG_KDF;
```

Table 2:66 - Definition of TPMI_ALG_SIG_SCHEME Type *(InterfaceTable())*

```
618    typedef  TPM_ALG_ID          TPMI_ALG_SIG_SCHEME;
```

Table 2:67 - Definition of TPMI_ECC_KEY_EXCHANGE Type *(InterfaceTable())*

```
619    typedef  TPM_ALG_ID          TPMI_ECC_KEY_EXCHANGE;
```

Table 2:68 - Definition of TPMI_ST_COMMAND_TAG Type *(InterfaceTable())*

```
620    typedef  TPM_ST              TPMI_ST_COMMAND_TAG;
```

Table 2:69 - Definition of TPMS_EMPTY Structure *(StructuresTable())*

```
621    typedef BYTE TPMS_EMPTY;
```

Table 2:70 - Definition of TPMS_ALGORITHM_DESCRIPTION Structure *(StructuresTable())*

```
622    typedef struct {
623        TPM_ALG_ID              alg;
624        TPMA_ALGORITHM          attributes;
625    } TPMS_ALGORITHM_DESCRIPTION;
```

Table 2:71 - Definition of TPMU_HA Union *(StructuresTable())*

```
626    typedef union {
627    #ifdef      TPM_ALG_SHA1
628        BYTE                    sha1[SHA1_DIGEST_SIZE];
629    #endif   // TPM_ALG_SHA1
630    #ifdef      TPM_ALG_SHA256
631        BYTE                    sha256[SHA256_DIGEST_SIZE];
632    #endif   // TPM_ALG_SHA256
633    #ifdef      TPM_ALG_SHA384
634        BYTE                    sha384[SHA384_DIGEST_SIZE];
635    #endif   // TPM_ALG_SHA384
636    #ifdef      TPM_ALG_SHA512
637        BYTE                    sha512[SHA512_DIGEST_SIZE];
638    #endif   // TPM_ALG_SHA512
639    #ifdef      TPM_ALG_SM3_256
640        BYTE                    sm3_256[SM3_256_DIGEST_SIZE];
641    #endif   // TPM_ALG_SM3_256
642    } TPMU_HA;
```

Table 2:72 - Definition of TPMT_HA Structure *(StructuresTable())*

```
643   typedef struct {
644       TPMI_ALG_HASH           hashAlg;
645       TPMU_HA                 digest;
646   } TPMT_HA;
```

Table 2:73 - Definition of TPM2B_DIGEST Structure *(StructuresTable())*

```
647   typedef union {
648       struct {
649           UINT16                  size;
650           BYTE                    buffer[sizeof(TPMU_HA)];
651       }           t;
652       TPM2B       b;
653   } TPM2B_DIGEST;
```

Table 2:74 - Definition of TPM2B_DATA Structure *(StructuresTable())*

```
654   typedef union {
655       struct {
656           UINT16                  size;
657           BYTE                    buffer[sizeof(TPMT_HA)];
658       }           t;
659       TPM2B       b;
660   } TPM2B_DATA;
```

Table 2:75 - Definition of Types for TPM2B_NONCE *(TypedefTable())*

```
661   typedef  TPM2B_DIGEST       TPM2B_NONCE;
```

Table 2:76 - Definition of Types for TPM2B_AUTH *(TypedefTable())*

```
662   typedef  TPM2B_DIGEST       TPM2B_AUTH;
```

Table 2:77 - Definition of Types for TPM2B_OPERAND *(TypedefTable())*

```
663   typedef  TPM2B_DIGEST       TPM2B_OPERAND;
```

Table 2:78 - Definition of TPM2B_EVENT Structure *(StructuresTable())*

```
664   typedef union {
665       struct {
666           UINT16                  size;
667           BYTE                    buffer[1024];
668       }           t;
669       TPM2B       b;
670   } TPM2B_EVENT;
```

Table 2:79 - Definition of TPM2B_MAX_BUFFER Structure *(StructuresTable())*

```
671   typedef union {
672       struct {
673           UINT16                  size;
674           BYTE                    buffer[MAX_DIGEST_BUFFER];
675       }           t;
676       TPM2B       b;
677   } TPM2B_MAX_BUFFER;
```

Table 2:80 - Definition of TPM2B_MAX_NV_BUFFER Structure *(StructuresTable())*

```
678   typedef union {
679       struct {
680           UINT16                  size;
```

```
681             BYTE                    buffer[MAX_NV_BUFFER_SIZE];
682         }           t;
683         TPM2B       b;
684     } TPM2B_MAX_NV_BUFFER;
```

Table 2:81 - Definition of Types for TPM2B_TIMEOUT *(TypedefTable())*

```
685     typedef  TPM2B_DIGEST       TPM2B_TIMEOUT;
```

Table 2:82 - Definition of TPM2B_IV Structure *(StructuresTable())*

```
686     typedef union {
687         struct {
688             UINT16                  size;
689             BYTE                    buffer[MAX_SYM_BLOCK_SIZE];
690         }           t;
691         TPM2B       b;
692     } TPM2B_IV;
```

Table 2:83 - Definition of TPMU_NAME Union *(StructuresTable())*

```
693     typedef union {
694         TPMT_HA                 digest;
695         TPM_HANDLE              handle;
696     } TPMU_NAME;
```

Table 2:84 - Definition of TPM2B_NAME Structure *(StructuresTable())*

```
697     typedef union {
698         struct {
699             UINT16                  size;
700             BYTE                    name[sizeof(TPMU_NAME)];
701         }           t;
702         TPM2B       b;
703     } TPM2B_NAME;
```

Table 2:85 - Definition of TPMS_PCR_SELECT Structure *(StructuresTable())*

```
704     typedef struct {
705         UINT8                   sizeofSelect;
706         BYTE                    pcrSelect[PCR_SELECT_MAX];
707     } TPMS_PCR_SELECT;
```

Table 2:86 - Definition of TPMS_PCR_SELECTION Structure *(StructuresTable())*

```
708     typedef struct {
709         TPMI_ALG_HASH           hash;
710         UINT8                   sizeofSelect;
711         BYTE                    pcrSelect[PCR_SELECT_MAX];
712     } TPMS_PCR_SELECTION;
```

Table 2:89 - Definition of TPMT_TK_CREATION Structure *(StructuresTable())*

```
713     typedef struct {
714         TPM_ST                  tag;
715         TPMI_RH_HIERARCHY       hierarchy;
716         TPM2B_DIGEST            digest;
717     } TPMT_TK_CREATION;
```

Table 2:90 - Definition of TPMT_TK_VERIFIED Structure *(StructuresTable())*

```
718     typedef struct {
```

```
719         TPM_ST                  tag;
720         TPMI_RH_HIERARCHY       hierarchy;
721         TPM2B_DIGEST            digest;
722     } TPMT_TK_VERIFIED;
```

Table 2:91 - Definition of TPMT_TK_AUTH Structure *(StructuresTable())*

```
723     typedef struct {
724         TPM_ST                  tag;
725         TPMI_RH_HIERARCHY       hierarchy;
726         TPM2B_DIGEST            digest;
727     } TPMT_TK_AUTH;
```

Table 2:92 - Definition of TPMT_TK_HASHCHECK Structure *(StructuresTable())*

```
728     typedef struct {
729         TPM_ST                  tag;
730         TPMI_RH_HIERARCHY       hierarchy;
731         TPM2B_DIGEST            digest;
732     } TPMT_TK_HASHCHECK;
```

Table 2:93 - Definition of TPMS_ALG_PROPERTY Structure *(StructuresTable())*

```
733     typedef struct {
734         TPM_ALG_ID              alg;
735         TPMA_ALGORITHM          algProperties;
736     } TPMS_ALG_PROPERTY;
```

Table 2:94 - Definition of TPMS_TAGGED_PROPERTY Structure *(StructuresTable())*

```
737     typedef struct {
738         TPM_PT                  property;
739         UINT32                  value;
740     } TPMS_TAGGED_PROPERTY;
```

Table 2:95 - Definition of TPMS_TAGGED_PCR_SELECT Structure *(StructuresTable())*

```
741     typedef struct {
742         TPM_PT_PCR              tag;
743         UINT8                   sizeofSelect;
744         BYTE                    pcrSelect[PCR_SELECT_MAX];
745     } TPMS_TAGGED_PCR_SELECT;
```

Table 2:96 - Definition of TPMS_TAGGED_POLICY Structure *(StructuresTable())*

```
746     typedef struct {
747         TPM_HANDLE              handle;
748         TPMT_HA                 policyHash;
749     } TPMS_TAGGED_POLICY;
```

Table 2:97 - Definition of TPML_CC Structure *(StructuresTable())*

```
750     typedef struct {
751         UINT32                  count;
752         TPM_CC                  commandCodes[MAX_CAP_CC];
753     } TPML_CC;
```

Table 2:98 - Definition of TPML_CCA Structure *(StructuresTable())*

```
754     typedef struct {
755         UINT32                  count;
756         TPMA_CC                 commandAttributes[MAX_CAP_CC];
```

```
} TPML_CCA;
```

Table 2:99 - Definition of TPML_ALG Structure *(StructuresTable())*

```
typedef struct {
    UINT32                  count;
    TPM_ALG_ID              algorithms[MAX_ALG_LIST_SIZE];
} TPML_ALG;
```

Table 2:100 - Definition of TPML_HANDLE Structure *(StructuresTable())*

```
typedef struct {
    UINT32                  count;
    TPM_HANDLE              handle[MAX_CAP_HANDLES];
} TPML_HANDLE;
```

Table 2:101 - Definition of TPML_DIGEST Structure *(StructuresTable())*

```
typedef struct {
    UINT32                  count;
    TPM2B_DIGEST            digests[8];
} TPML_DIGEST;
```

Table 2:102 - Definition of TPML_DIGEST_VALUES Structure *(StructuresTable())*

```
typedef struct {
    UINT32                  count;
    TPMT_HA                 digests[HASH_COUNT];
} TPML_DIGEST_VALUES;
```

Table 2:103 - Definition of TPM2B_DIGEST_VALUES Structure *(StructuresTable())*

```
typedef union {
    struct {
        UINT16                  size;
        BYTE                    buffer[sizeof(TPML_DIGEST_VALUES)];
    }           t;
    TPM2B       b;
} TPM2B_DIGEST_VALUES;
```

Table 2:104 - Definition of TPML_PCR_SELECTION Structure *(StructuresTable())*

```
typedef struct {
    UINT32                  count;
    TPMS_PCR_SELECTION      pcrSelections[HASH_COUNT];
} TPML_PCR_SELECTION;
```

Table 2:105 - Definition of TPML_ALG_PROPERTY Structure *(StructuresTable())*

```
typedef struct {
    UINT32                  count;
    TPMS_ALG_PROPERTY       algProperties[MAX_CAP_ALGS];
} TPML_ALG_PROPERTY;
```

Table 2:106 - Definition of TPML_TAGGED_TPM_PROPERTY Structure *(StructuresTable())*

```
typedef struct {
    UINT32                  count;
    TPMS_TAGGED_PROPERTY    tpmProperty[MAX_TPM_PROPERTIES];
} TPML_TAGGED_TPM_PROPERTY;
```


Table 2:107 - Definition of TPML_TAGGED_PCR_PROPERTY Structure *(StructuresTable())*

```
793   typedef struct {
794       UINT32                  count;
795       TPMS_TAGGED_PCR_SELECT  pcrProperty[MAX_PCR_PROPERTIES];
796   } TPML_TAGGED_PCR_PROPERTY;
```

Table 2:108 - Definition of TPML_ECC_CURVE Structure *(StructuresTable())*

```
797   typedef struct {
798       UINT32                  count;
799       TPM_ECC_CURVE           eccCurves[MAX_ECC_CURVES];
800   } TPML_ECC_CURVE;
```

Table 2:109 - Definition of TPML_TAGGED_POLICY Structure *(StructuresTable())*

```
801   typedef struct {
802       UINT32                  count;
803       TPMS_TAGGED_POLICY      policies[MAX_TAGGED_POLICIES];
804   } TPML_TAGGED_POLICY;
```

Table 2:110 - Definition of TPMU_CAPABILITIES Union *(StructuresTable())*

```
805   typedef union {
806       TPML_ALG_PROPERTY           algorithms;
807       TPML_HANDLE                 handles;
808       TPML_CCA                    command;
809       TPML_CC                     ppCommands;
810       TPML_CC                     auditCommands;
811       TPML_PCR_SELECTION          assignedPCR;
812       TPML_TAGGED_TPM_PROPERTY    tpmProperties;
813       TPML_TAGGED_PCR_PROPERTY    pcrProperties;
814   #ifdef      TPM_ALG_ECC
815       TPML_ECC_CURVE              eccCurves;
816   #endif   // TPM_ALG_ECC
817       TPML_TAGGED_POLICY          authPolicies;
818   } TPMU_CAPABILITIES;
```

Table 2:111 - Definition of TPMS_CAPABILITY_DATA Structure *(StructuresTable())*

```
819   typedef struct {
820       TPM_CAP                 capability;
821       TPMU_CAPABILITIES       data;
822   } TPMS_CAPABILITY_DATA;
```

Table 2:112 - Definition of TPMS_CLOCK_INFO Structure *(StructuresTable())*

```
823   typedef struct {
824       UINT64                  clock;
825       UINT32                  resetCount;
826       UINT32                  restartCount;
827       TPMI_YES_NO             safe;
828   } TPMS_CLOCK_INFO;
```

Table 2:113 - Definition of TPMS_TIME_INFO Structure *(StructuresTable())*

```
829   typedef struct {
830       UINT64                  time;
831       TPMS_CLOCK_INFO         clockInfo;
832   } TPMS_TIME_INFO;
```

Table 2:114 - Definition of TPMS_TIME_ATTEST_INFO Structure *(StructuresTable())*

```
typedef struct {
    TPMS_TIME_INFO          time;
    UINT64                  firmwareVersion;
} TPMS_TIME_ATTEST_INFO;
```

Table 2:115 - Definition of TPMS_CERTIFY_INFO Structure *(StructuresTable())*

```
typedef struct {
    TPM2B_NAME              name;
    TPM2B_NAME              qualifiedName;
} TPMS_CERTIFY_INFO;
```

Table 2:116 - Definition of TPMS_QUOTE_INFO Structure *(StructuresTable())*

```
typedef struct {
    TPML_PCR_SELECTION      pcrSelect;
    TPM2B_DIGEST            pcrDigest;
} TPMS_QUOTE_INFO;
```

Table 2:117 - Definition of TPMS_COMMAND_AUDIT_INFO Structure *(StructuresTable())*

```
typedef struct {
    UINT64                  auditCounter;
    TPM_ALG_ID              digestAlg;
    TPM2B_DIGEST            auditDigest;
    TPM2B_DIGEST            commandDigest;
} TPMS_COMMAND_AUDIT_INFO;
```

Table 2:118 - Definition of TPMS_SESSION_AUDIT_INFO Structure *(StructuresTable())*

```
typedef struct {
    TPMI_YES_NO             exclusiveSession;
    TPM2B_DIGEST            sessionDigest;
} TPMS_SESSION_AUDIT_INFO;
```

Table 2:119 - Definition of TPMS_CREATION_INFO Structure *(StructuresTable())*

```
typedef struct {
    TPM2B_NAME              objectName;
    TPM2B_DIGEST            creationHash;
} TPMS_CREATION_INFO;
```

Table 2:120 - Definition of TPMS_NV_CERTIFY_INFO Structure *(StructuresTable())*

```
typedef struct {
    TPM2B_NAME              indexName;
    UINT16                  offset;
    TPM2B_MAX_NV_BUFFER     nvContents;
} TPMS_NV_CERTIFY_INFO;
```

Table 2:121 - Definition of TPMI_ST_ATTEST Type *(InterfaceTable())*

```
typedef  TPM_ST             TPMI_ST_ATTEST;
```

Table 2:122 - Definition of TPMU_ATTEST Union *(StructuresTable())*

```
typedef union {
    TPMS_CERTIFY_INFO           certify;
    TPMS_CREATION_INFO          creation;
    TPMS_QUOTE_INFO             quote;
    TPMS_COMMAND_AUDIT_INFO     commandAudit;
    TPMS_SESSION_AUDIT_INFO     sessionAudit;
```

```
871         TPMS_TIME_ATTEST_INFO       time;
872         TPMS_NV_CERTIFY_INFO        nv;
873     } TPMU_ATTEST;
```

Table 2:123 - Definition of TPMS_ATTEST Structure *(StructuresTable())*

```
874     typedef struct {
875         TPM_GENERATED           magic;
876         TPMI_ST_ATTEST          type;
877         TPM2B_NAME              qualifiedSigner;
878         TPM2B_DATA              extraData;
879         TPMS_CLOCK_INFO         clockInfo;
880         UINT64                  firmwareVersion;
881         TPMU_ATTEST             attested;
882     } TPMS_ATTEST;
```

Table 2:124 - Definition of TPM2B_ATTEST Structure *(StructuresTable())*

```
883     typedef union {
884         struct {
885             UINT16                  size;
886             BYTE                    attestationData[sizeof(TPMS_ATTEST)];
887         }           t;
888         TPM2B       b;
889     } TPM2B_ATTEST;
```

Table 2:125 - Definition of TPMS_AUTH_COMMAND Structure *(StructuresTable())*

```
890     typedef struct {
891         TPMI_SH_AUTH_SESSION     sessionHandle;
892         TPM2B_NONCE              nonce;
893         TPMA_SESSION             sessionAttributes;
894         TPM2B_AUTH               hmac;
895     } TPMS_AUTH_COMMAND;
```

Table 2:126 - Definition of TPMS_AUTH_RESPONSE Structure *(StructuresTable())*

```
896     typedef struct {
897         TPM2B_NONCE             nonce;
898         TPMA_SESSION            sessionAttributes;
899         TPM2B_AUTH              hmac;
900     } TPMS_AUTH_RESPONSE;
```

Table 2:127 - Definition of TPMI_TDES_KEY_BITS Type *(InterfaceTable())*

```
901     typedef  TPM_KEY_BITS       TPMI_TDES_KEY_BITS;
```

Table 2:127 - Definition of TPMI_AES_KEY_BITS Type *(InterfaceTable())*

```
902     typedef  TPM_KEY_BITS       TPMI_AES_KEY_BITS;
```

Table 2:127 - Definition of TPMI_SM4_KEY_BITS Type *(InterfaceTable())*

```
903     typedef  TPM_KEY_BITS       TPMI_SM4_KEY_BITS;
```

Table 2:127 - Definition of TPMI_CAMELLIA_KEY_BITS Type *(InterfaceTable())*

```
904     typedef  TPM_KEY_BITS       TPMI_CAMELLIA_KEY_BITS;
```

Table 2:128 - Definition of TPMU_SYM_KEY_BITS Union *(StructuresTable())*

```
905   typedef union {
906   #ifdef      TPM_ALG_TDES
907       TPMI_TDES_KEY_BITS        tdes;
908   #endif   // TPM_ALG_TDES
909   #ifdef      TPM_ALG_AES
910       TPMI_AES_KEY_BITS         aes;
911   #endif   // TPM_ALG_AES
912   #ifdef      TPM_ALG_SM4
913       TPMI_SM4_KEY_BITS         sm4;
914   #endif   // TPM_ALG_SM4
915   #ifdef      TPM_ALG_CAMELLIA
916       TPMI_CAMELLIA_KEY_BITS    camellia;
917   #endif   // TPM_ALG_CAMELLIA
918       TPM_KEY_BITS              sym;
919   #ifdef      TPM_ALG_XOR
920       TPMI_ALG_HASH             xor;
921   #endif   // TPM_ALG_XOR
922   } TPMU_SYM_KEY_BITS;
```

Table 2:129 - Definition of TPMU_SYM_MODE Union *(StructuresTable())*

```
923   typedef union {
924   #ifdef      TPM_ALG_TDES
925       TPMI_ALG_SYM_MODE       tdes;
926   #endif   // TPM_ALG_TDES
927   #ifdef      TPM_ALG_AES
928       TPMI_ALG_SYM_MODE       aes;
929   #endif   // TPM_ALG_AES
930   #ifdef      TPM_ALG_SM4
931       TPMI_ALG_SYM_MODE       sm4;
932   #endif   // TPM_ALG_SM4
933   #ifdef      TPM_ALG_CAMELLIA
934       TPMI_ALG_SYM_MODE       camellia;
935   #endif   // TPM_ALG_CAMELLIA
936       TPMI_ALG_SYM_MODE       sym;
937   } TPMU_SYM_MODE;
```

Table 2:131 - Definition of TPMT_SYM_DEF Structure *(StructuresTable())*

```
938   typedef struct {
939       TPMI_ALG_SYM            algorithm;
940       TPMU_SYM_KEY_BITS       keyBits;
941       TPMU_SYM_MODE           mode;
942   } TPMT_SYM_DEF;
```

Table 2:132 - Definition of TPMT_SYM_DEF_OBJECT Structure *(StructuresTable())*

```
943   typedef struct {
944       TPMI_ALG_SYM_OBJECT     algorithm;
945       TPMU_SYM_KEY_BITS       keyBits;
946       TPMU_SYM_MODE           mode;
947   } TPMT_SYM_DEF_OBJECT;
```

Table 2:133 - Definition of TPM2B_SYM_KEY Structure *(StructuresTable())*

```
948   typedef union {
949       struct {
950           UINT16                  size;
951           BYTE                    buffer[MAX_SYM_KEY_BYTES];
952       }           t;
953       TPM2B       b;
954   } TPM2B_SYM_KEY;
```

Table 2:134 - Definition of TPMS_SYMCIPHER_PARMS Structure *(StructuresTable())*

```
typedef struct {
    TPMT_SYM_DEF_OBJECT     sym;
} TPMS_SYMCIPHER_PARMS;
```

Table 2:135 - Definition of TPM2B_LABEL Structure *(StructuresTable())*

```
typedef union {
    struct {
        UINT16                  size;
        BYTE                    buffer[LABEL_MAX_BUFFER];
    }           t;
    TPM2B       b;
} TPM2B_LABEL;
```

Table 2:136 - Definition of TPMS_DERIVE Structure *(StructuresTable())*

```
typedef struct {
    TPM2B_LABEL             label;
    TPM2B_LABEL             context;
} TPMS_DERIVE;
```

Table 2:137 - Definition of TPM2B_DERIVE Structure *(StructuresTable())*

```
typedef union {
    struct {
        UINT16                  size;
        BYTE                    buffer[sizeof(TPMS_DERIVE)];
    }           t;
    TPM2B       b;
} TPM2B_DERIVE;
```

Table 2:138 - Definition of TPMU_SENSITIVE_CREATE Union *(StructuresTable())*

```
typedef union {
    BYTE                    create[MAX_SYM_DATA];
    TPMS_DERIVE             derive;
} TPMU_SENSITIVE_CREATE;
```

Table 2:139 - Definition of TPM2B_SENSITIVE_DATA Structure *(StructuresTable())*

```
typedef union {
    struct {
        UINT16                  size;
        BYTE                    buffer[sizeof(TPMU_SENSITIVE_CREATE)];
    }           t;
    TPM2B       b;
} TPM2B_SENSITIVE_DATA;
```

Table 2:140 - Definition of TPMS_SENSITIVE_CREATE Structure *(StructuresTable())*

```
typedef struct {
    TPM2B_AUTH              userAuth;
    TPM2B_SENSITIVE_DATA    data;
} TPMS_SENSITIVE_CREATE;
```

Table 2:141 - Definition of TPM2B_SENSITIVE_CREATE Structure *(StructuresTable())*

```
typedef struct {
    UINT16                   size;
    TPMS_SENSITIVE_CREATE    sensitive;
```

```
} TPM2B_SENSITIVE_CREATE;
```

Table 2:142 - Definition of TPMS_SCHEME_HASH Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_HASH           hashAlg;
} TPMS_SCHEME_HASH;
```

Table 2:143 - Definition of TPMS_SCHEME_ECDAA Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_HASH           hashAlg;
    UINT16                  count;
} TPMS_SCHEME_ECDAA;
```

Table 2:144 - Definition of TPMI_ALG_KEYEDHASH_SCHEME Type *(InterfaceTable())*

```
typedef  TPM_ALG_ID         TPMI_ALG_KEYEDHASH_SCHEME;
```

Table 2:145 - Definition of Types for HMAC_SIG_SCHEME *(TypedefTable())*

```
typedef  TPMS_SCHEME_HASH    TPMS_SCHEME_HMAC;
```

Table 2:146 - Definition of TPMS_SCHEME_XOR Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_HASH           hashAlg;
    TPMI_ALG_KDF            kdf;
} TPMS_SCHEME_XOR;
```

Table 2:147 - Definition of TPMU_SCHEME_KEYEDHASH Union *(StructuresTable())*

```
typedef union {
#ifdef      TPM_ALG_HMAC
    TPMS_SCHEME_HMAC        hmac;
#endif   // TPM_ALG_HMAC
#ifdef      TPM_ALG_XOR
    TPMS_SCHEME_XOR         xor;
#endif   // TPM_ALG_XOR
} TPMU_SCHEME_KEYEDHASH;
```

Table 2:148 - Definition of TPMT_KEYEDHASH_SCHEME Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_KEYEDHASH_SCHEME     scheme;
    TPMU_SCHEME_KEYEDHASH         details;
} TPMT_KEYEDHASH_SCHEME;
```

Table 2:149 - Definition of Types for RSA Signature Schemes *(TypedefTable())*

```
typedef  TPMS_SCHEME_HASH    TPMS_SIG_SCHEME_RSASSA;
typedef  TPMS_SCHEME_HASH    TPMS_SIG_SCHEME_RSAPSS;
```

Table 2:150 - Definition of Types for ECC Signature Schemes *(TypedefTable())*

```
typedef  TPMS_SCHEME_HASH     TPMS_SIG_SCHEME_ECDSA;
typedef  TPMS_SCHEME_HASH     TPMS_SIG_SCHEME_SM2;
typedef  TPMS_SCHEME_HASH     TPMS_SIG_SCHEME_ECSCHNORR;
typedef  TPMS_SCHEME_ECDAA    TPMS_SIG_SCHEME_ECDAA;
```

Table 2:151 - Definition of TPMU_SIG_SCHEME Union *(StructuresTable())*

```
1026    typedef union {
1027    #ifdef      TPM_ALG_ECC
1028        TPMS_SIG_SCHEME_ECDAA        ecdaa;
1029    #endif   // TPM_ALG_ECC
1030    #ifdef      TPM_ALG_RSASSA
1031        TPMS_SIG_SCHEME_RSASSA       rsassa;
1032    #endif   // TPM_ALG_RSASSA
1033    #ifdef      TPM_ALG_RSAPSS
1034        TPMS_SIG_SCHEME_RSAPSS       rsapss;
1035    #endif   // TPM_ALG_RSAPSS
1036    #ifdef      TPM_ALG_ECDSA
1037        TPMS_SIG_SCHEME_ECDSA        ecdsa;
1038    #endif   // TPM_ALG_ECDSA
1039    #ifdef      TPM_ALG_SM2
1040        TPMS_SIG_SCHEME_SM2          sm2;
1041    #endif   // TPM_ALG_SM2
1042    #ifdef      TPM_ALG_ECSCHNORR
1043        TPMS_SIG_SCHEME_ECSCHNORR    ecschnorr;
1044    #endif   // TPM_ALG_ECSCHNORR
1045    #ifdef      TPM_ALG_HMAC
1046        TPMS_SCHEME_HMAC             hmac;
1047    #endif   // TPM_ALG_HMAC
1048        TPMS_SCHEME_HASH             any;
1049    } TPMU_SIG_SCHEME;
```

Table 2:152 - Definition of TPMT_SIG_SCHEME Structure *(StructuresTable())*

```
1050    typedef struct {
1051        TPMI_ALG_SIG_SCHEME     scheme;
1052        TPMU_SIG_SCHEME         details;
1053    } TPMT_SIG_SCHEME;
```

Table 2:153 - Definition of Types for Encryption Schemes *(TypedefTable())*

```
1054    typedef  TPMS_SCHEME_HASH    TPMS_ENC_SCHEME_OAEP;
1055    typedef  TPMS_EMPTY          TPMS_ENC_SCHEME_RSAES;
```

Table 2:154 - Definition of Types for ECC Key Exchange *(TypedefTable())*

```
1056    typedef  TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECDH;
1057    typedef  TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECMQV;
```

Table 2:155 - Definition of Types for KDF Schemes *(TypedefTable())*

```
1058    typedef  TPMS_SCHEME_HASH    TPMS_SCHEME_MGF1;
1059    typedef  TPMS_SCHEME_HASH    TPMS_SCHEME_KDF1_SP800_56A;
1060    typedef  TPMS_SCHEME_HASH    TPMS_SCHEME_KDF2;
1061    typedef  TPMS_SCHEME_HASH    TPMS_SCHEME_KDF1_SP800_108;
```

Table 2:156 - Definition of TPMU_KDF_SCHEME Union *(StructuresTable())*

```
1062    typedef union {
1063    #ifdef      TPM_ALG_MGF1
1064        TPMS_SCHEME_MGF1              mgf1;
1065    #endif   // TPM_ALG_MGF1
1066    #ifdef      TPM_ALG_KDF1_SP800_56A
1067        TPMS_SCHEME_KDF1_SP800_56A    kdf1_sp800_56a;
1068    #endif   // TPM_ALG_KDF1_SP800_56A
1069    #ifdef      TPM_ALG_KDF2
1070        TPMS_SCHEME_KDF2              kdf2;
1071    #endif   // TPM_ALG_KDF2
1072    #ifdef      TPM_ALG_KDF1_SP800_108
1073        TPMS_SCHEME_KDF1_SP800_108    kdf1_sp800_108;
```

```
#endif   // TPM_ALG_KDF1_SP800_108
} TPMU_KDF_SCHEME;
```

Table 2:157 - Definition of TPMT_KDF_SCHEME Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_KDF            scheme;
    TPMU_KDF_SCHEME         details;
} TPMT_KDF_SCHEME;
```

Table 2:158 - Definition of TPMI_ALG_ASYM_SCHEME Type *(InterfaceTable())*

```
typedef  TPM_ALG_ID         TPMI_ALG_ASYM_SCHEME;
```

Table 2:159 - Definition of TPMU_ASYM_SCHEME Union *(StructuresTable())*

```
typedef union {
#ifdef      TPM_ALG_ECDH
    TPMS_KEY_SCHEME_ECDH         ecdh;
#endif   // TPM_ALG_ECDH
#ifdef      TPM_ALG_ECMQV
    TPMS_KEY_SCHEME_ECMQV        ecmqv;
#endif   // TPM_ALG_ECMQV
#ifdef      TPM_ALG_ECC
    TPMS_SIG_SCHEME_ECDAA        ecdaa;
#endif   // TPM_ALG_ECC
#ifdef      TPM_ALG_RSASSA
    TPMS_SIG_SCHEME_RSASSA       rsassa;
#endif   // TPM_ALG_RSASSA
#ifdef      TPM_ALG_RSAPSS
    TPMS_SIG_SCHEME_RSAPSS       rsapss;
#endif   // TPM_ALG_RSAPSS
#ifdef      TPM_ALG_ECDSA
    TPMS_SIG_SCHEME_ECDSA        ecdsa;
#endif   // TPM_ALG_ECDSA
#ifdef      TPM_ALG_SM2
    TPMS_SIG_SCHEME_SM2          sm2;
#endif   // TPM_ALG_SM2
#ifdef      TPM_ALG_ECSCHNORR
    TPMS_SIG_SCHEME_ECSCHNORR    ecschnorr;
#endif   // TPM_ALG_ECSCHNORR
#ifdef      TPM_ALG_RSAES
    TPMS_ENC_SCHEME_RSAES        rsaes;
#endif   // TPM_ALG_RSAES
#ifdef      TPM_ALG_OAEP
    TPMS_ENC_SCHEME_OAEP         oaep;
#endif   // TPM_ALG_OAEP
    TPMS_SCHEME_HASH             anySig;
} TPMU_ASYM_SCHEME;
```

Table 2:160 - Definition of TPMT_ASYM_SCHEME Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_ASYM_SCHEME     scheme;
    TPMU_ASYM_SCHEME         details;
} TPMT_ASYM_SCHEME;
```

Table 2:161 - Definition of TPMI_ALG_RSA_SCHEME Type *(InterfaceTable())*

```
typedef  TPM_ALG_ID         TPMI_ALG_RSA_SCHEME;
```

Table 2:162 - Definition of TPMT_RSA_SCHEME Structure *(StructuresTable())*

```
1119    typedef struct {
1120        TPMI_ALG_RSA_SCHEME     scheme;
1121        TPMU_ASYM_SCHEME        details;
1122    } TPMT_RSA_SCHEME;
```

Table 2:163 - Definition of TPMI_ALG_RSA_DECRYPT Type *(InterfaceTable())*

```
1123    typedef  TPM_ALG_ID         TPMI_ALG_RSA_DECRYPT;
```

Table 2:164 - Definition of TPMT_RSA_DECRYPT Structure *(StructuresTable())*

```
1124    typedef struct {
1125        TPMI_ALG_RSA_DECRYPT     scheme;
1126        TPMU_ASYM_SCHEME         details;
1127    } TPMT_RSA_DECRYPT;
```

Table 2:165 - Definition of TPM2B_PUBLIC_KEY_RSA Structure *(StructuresTable())*

```
1128    typedef union {
1129        struct {
1130            UINT16                  size;
1131            BYTE                    buffer[MAX_RSA_KEY_BYTES];
1132        }           t;
1133        TPM2B       b;
1134    } TPM2B_PUBLIC_KEY_RSA;
```

Table 2:166 - Definition of TPMI_RSA_KEY_BITS Type *(InterfaceTable())*

```
1135    typedef  TPM_KEY_BITS       TPMI_RSA_KEY_BITS;
```

Table 2:167 - Definition of TPM2B_PRIVATE_KEY_RSA Structure *(StructuresTable())*

```
1136    typedef union {
1137        struct {
1138            UINT16                  size;
1139            BYTE                    buffer[MAX_RSA_KEY_BYTES/2];
1140        }           t;
1141        TPM2B       b;
1142    } TPM2B_PRIVATE_KEY_RSA;
```

Table 2:168 - Definition of TPM2B_ECC_PARAMETER Structure *(StructuresTable())*

```
1143    typedef union {
1144        struct {
1145            UINT16                  size;
1146            BYTE                    buffer[MAX_ECC_KEY_BYTES];
1147        }           t;
1148        TPM2B       b;
1149    } TPM2B_ECC_PARAMETER;
```

Table 2:169 - Definition of TPMS_ECC_POINT Structure *(StructuresTable())*

```
1150    typedef struct {
1151        TPM2B_ECC_PARAMETER     x;
1152        TPM2B_ECC_PARAMETER     y;
1153    } TPMS_ECC_POINT;
```

Table 2:170 - Definition of TPM2B_ECC_POINT Structure *(StructuresTable())*

```
1154    typedef struct {
1155        UINT16                  size;
1156        TPMS_ECC_POINT          point;
```

```
} TPM2B_ECC_POINT;
```

Table 2:171 - Definition of TPMI_ALG_ECC_SCHEME Type *(InterfaceTable())*

```
typedef  TPM_ALG_ID          TPMI_ALG_ECC_SCHEME;
```

Table 2:172 - Definition of TPMI_ECC_CURVE Type *(InterfaceTable())*

```
typedef  TPM_ECC_CURVE       TPMI_ECC_CURVE;
```

Table 2:173 - Definition of TPMT_ECC_SCHEME Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_ECC_SCHEME     scheme;
    TPMU_ASYM_SCHEME        details;
} TPMT_ECC_SCHEME;
```

Table 2:174 - Definition of TPMS_ALGORITHM_DETAIL_ECC Structure *(StructuresTable())*

```
typedef struct {
    TPM_ECC_CURVE           curveID;
    UINT16                  keySize;
    TPMT_KDF_SCHEME         kdf;
    TPMT_ECC_SCHEME         sign;
    TPM2B_ECC_PARAMETER     p;
    TPM2B_ECC_PARAMETER     a;
    TPM2B_ECC_PARAMETER     b;
    TPM2B_ECC_PARAMETER     gX;
    TPM2B_ECC_PARAMETER     gY;
    TPM2B_ECC_PARAMETER     n;
    TPM2B_ECC_PARAMETER     h;
} TPMS_ALGORITHM_DETAIL_ECC;
```

Table 2:175 - Definition of TPMS_SIGNATURE_RSA Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_HASH           hash;
    TPM2B_PUBLIC_KEY_RSA    sig;
} TPMS_SIGNATURE_RSA;
```

Table 2:176 - Definition of Types for Signature *(TypedefTable())*

```
typedef  TPMS_SIGNATURE_RSA    TPMS_SIGNATURE_RSASSA;
typedef  TPMS_SIGNATURE_RSA    TPMS_SIGNATURE_RSAPSS;
```

Table 2:177 - Definition of TPMS_SIGNATURE_ECC Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_HASH           hash;
    TPM2B_ECC_PARAMETER     signatureR;
    TPM2B_ECC_PARAMETER     signatureS;
} TPMS_SIGNATURE_ECC;
```

Table 2:178 - Definition of Types for TPMS_SIGNATURE_ECC *(TypedefTable())*

```
typedef  TPMS_SIGNATURE_ECC    TPMS_SIGNATURE_ECDAA;
typedef  TPMS_SIGNATURE_ECC    TPMS_SIGNATURE_ECDSA;
typedef  TPMS_SIGNATURE_ECC    TPMS_SIGNATURE_SM2;
typedef  TPMS_SIGNATURE_ECC    TPMS_SIGNATURE_ECSCHNORR;
```

Table 2:179 - Definition of TPMU_SIGNATURE Union *(StructuresTable())*

```
1192    typedef union {
1193    #ifdef      TPM_ALG_ECC
1194        TPMS_SIGNATURE_ECDAA        ecdaa;
1195    #endif   // TPM_ALG_ECC
1196    #ifdef      TPM_ALG_RSA
1197        TPMS_SIGNATURE_RSASSA       rsassa;
1198    #endif   // TPM_ALG_RSA
1199    #ifdef      TPM_ALG_RSA
1200        TPMS_SIGNATURE_RSAPSS       rsapss;
1201    #endif   // TPM_ALG_RSA
1202    #ifdef      TPM_ALG_ECC
1203        TPMS_SIGNATURE_ECDSA        ecdsa;
1204    #endif   // TPM_ALG_ECC
1205    #ifdef      TPM_ALG_ECC
1206        TPMS_SIGNATURE_SM2          sm2;
1207    #endif   // TPM_ALG_ECC
1208    #ifdef      TPM_ALG_ECC
1209        TPMS_SIGNATURE_ECSCHNORR    ecschnorr;
1210    #endif   // TPM_ALG_ECC
1211    #ifdef      TPM_ALG_HMAC
1212        TPMT_HA                     hmac;
1213    #endif   // TPM_ALG_HMAC
1214        TPMS_SCHEME_HASH            any;
1215    } TPMU_SIGNATURE;
```

Table 2:180 - Definition of TPMT_SIGNATURE Structure *(StructuresTable())*

```
1216    typedef struct {
1217        TPMI_ALG_SIG_SCHEME     sigAlg;
1218        TPMU_SIGNATURE          signature;
1219    } TPMT_SIGNATURE;
```

Table 2:181 - Definition of TPMU_ENCRYPTED_SECRET Union *(StructuresTable())*

```
1220    typedef union {
1221    #ifdef      TPM_ALG_ECC
1222        BYTE                    ecc[sizeof(TPMS_ECC_POINT)];
1223    #endif   // TPM_ALG_ECC
1224    #ifdef      TPM_ALG_RSA
1225        BYTE                    rsa[MAX_RSA_KEY_BYTES];
1226    #endif   // TPM_ALG_RSA
1227    #ifdef      TPM_ALG_SYMCIPHER
1228        BYTE                    symmetric[sizeof(TPM2B_DIGEST)];
1229    #endif   // TPM_ALG_SYMCIPHER
1230    #ifdef      TPM_ALG_KEYEDHASH
1231        BYTE                    keyedHash[sizeof(TPM2B_DIGEST)];
1232    #endif   // TPM_ALG_KEYEDHASH
1233    } TPMU_ENCRYPTED_SECRET;
```

Table 2:182 - Definition of TPM2B_ENCRYPTED_SECRET Structure *(StructuresTable())*

```
1234    typedef union {
1235        struct {
1236            UINT16                  size;
1237            BYTE                    secret[sizeof(TPMU_ENCRYPTED_SECRET)];
1238        }           t;
1239        TPM2B       b;
1240    } TPM2B_ENCRYPTED_SECRET;
```

Table 2:183 - Definition of TPMI_ALG_PUBLIC Type *(InterfaceTable())*

```
1241    typedef  TPM_ALG_ID         TPMI_ALG_PUBLIC;
```

Table 2:184 - Definition of TPMU_PUBLIC_ID Union *(StructuresTable())*

```
typedef union {
#ifdef      TPM_ALG_KEYEDHASH
    TPM2B_DIGEST            keyedHash;
#endif   // TPM_ALG_KEYEDHASH
#ifdef      TPM_ALG_SYMCIPHER
    TPM2B_DIGEST            sym;
#endif   // TPM_ALG_SYMCIPHER
#ifdef      TPM_ALG_RSA
    TPM2B_PUBLIC_KEY_RSA    rsa;
#endif   // TPM_ALG_RSA
#ifdef      TPM_ALG_ECC
    TPMS_ECC_POINT          ecc;
#endif   // TPM_ALG_ECC
    TPMS_DERIVE             derive;
} TPMU_PUBLIC_ID;
```

Table 2:185 - Definition of TPMS_KEYEDHASH_PARMS Structure *(StructuresTable())*

```
typedef struct {
    TPMT_KEYEDHASH_SCHEME     scheme;
} TPMS_KEYEDHASH_PARMS;
```

Table 2:186 - Definition of TPMS_ASYM_PARMS Structure *(StructuresTable())*

```
typedef struct {
    TPMT_SYM_DEF_OBJECT     symmetric;
    TPMT_ASYM_SCHEME        scheme;
} TPMS_ASYM_PARMS;
```

Table 2:187 - Definition of TPMS_RSA_PARMS Structure *(StructuresTable())*

```
typedef struct {
    TPMT_SYM_DEF_OBJECT     symmetric;
    TPMT_RSA_SCHEME         scheme;
    TPMI_RSA_KEY_BITS       keyBits;
    UINT32                  exponent;
} TPMS_RSA_PARMS;
```

Table 2:188 - Definition of TPMS_ECC_PARMS Structure *(StructuresTable())*

```
typedef struct {
    TPMT_SYM_DEF_OBJECT     symmetric;
    TPMT_ECC_SCHEME         scheme;
    TPMI_ECC_CURVE          curveID;
    TPMT_KDF_SCHEME         kdf;
} TPMS_ECC_PARMS;
```

Table 2:189 - Definition of TPMU_PUBLIC_PARMS Union *(StructuresTable())*

```
typedef union {
#ifdef      TPM_ALG_KEYEDHASH
    TPMS_KEYEDHASH_PARMS    keyedHashDetail;
#endif   // TPM_ALG_KEYEDHASH
#ifdef      TPM_ALG_SYMCIPHER
    TPMS_SYMCIPHER_PARMS    symDetail;
#endif   // TPM_ALG_SYMCIPHER
#ifdef      TPM_ALG_RSA
    TPMS_RSA_PARMS          rsaDetail;
#endif   // TPM_ALG_RSA
#ifdef      TPM_ALG_ECC
    TPMS_ECC_PARMS          eccDetail;
```

```
#endif   // TPM_ALG_ECC
    TPMS_ASYM_PARMS         asymDetail;
} TPMU_PUBLIC_PARMS;
```

Table 2:190 - Definition of TPMT_PUBLIC_PARMS Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_PUBLIC         type;
    TPMU_PUBLIC_PARMS       parameters;
} TPMT_PUBLIC_PARMS;
```

Table 2:191 - Definition of TPMT_PUBLIC Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_PUBLIC         type;
    TPMI_ALG_HASH           nameAlg;
    TPMA_OBJECT             objectAttributes;
    TPM2B_DIGEST            authPolicy;
    TPMU_PUBLIC_PARMS       parameters;
    TPMU_PUBLIC_ID          unique;
} TPMT_PUBLIC;
```

Table 2:192 - Definition of TPM2B_PUBLIC Structure *(StructuresTable())*

```
typedef struct {
    UINT16                  size;
    TPMT_PUBLIC             publicArea;
} TPM2B_PUBLIC;
```

Table 2:193 - Definition of TPM2B_TEMPLATE Structure *(StructuresTable())*

```
typedef union {
    struct {
        UINT16                  size;
        BYTE                    buffer[sizeof(TPMT_PUBLIC)];
    }           t;
    TPM2B       b;
} TPM2B_TEMPLATE;
```

Table 2:194 - Definition of TPM2B_PRIVATE_VENDOR_SPECIFIC Structure *(StructuresTable())*

```
typedef union {
    struct {
        UINT16                  size;
        BYTE                    buffer[PRIVATE_VENDOR_SPECIFIC_BYTES];
    }           t;
    TPM2B       b;
} TPM2B_PRIVATE_VENDOR_SPECIFIC;
```

Table 2:195 - Definition of TPMU_SENSITIVE_COMPOSITE Union *(StructuresTable())*

```
typedef union {
#ifdef      TPM_ALG_RSA
    TPM2B_PRIVATE_KEY_RSA           rsa;
#endif   // TPM_ALG_RSA
#ifdef      TPM_ALG_ECC
    TPM2B_ECC_PARAMETER             ecc;
#endif   // TPM_ALG_ECC
#ifdef      TPM_ALG_KEYEDHASH
    TPM2B_SENSITIVE_DATA            bits;
#endif   // TPM_ALG_KEYEDHASH
#ifdef      TPM_ALG_SYMCIPHER
    TPM2B_SYM_KEY                   sym;
```

```
#endif   // TPM_ALG_SYMCIPHER
    TPM2B_PRIVATE_VENDOR_SPECIFIC    any;
} TPMU_SENSITIVE_COMPOSITE;
```

Table 2:196 - Definition of TPMT_SENSITIVE Structure *(StructuresTable())*

```
typedef struct {
    TPMI_ALG_PUBLIC             sensitiveType;
    TPM2B_AUTH                  authValue;
    TPM2B_DIGEST                seedValue;
    TPMU_SENSITIVE_COMPOSITE    sensitive;
} TPMT_SENSITIVE;
```

Table 2:197 - Definition of TPM2B_SENSITIVE Structure *(StructuresTable())*

```
typedef struct {
    UINT16                  size;
    TPMT_SENSITIVE          sensitiveArea;
} TPM2B_SENSITIVE;
```

Table 2:198 - Definition of _PRIVATE Structure *(StructuresTable())*

```
typedef struct {
    TPM2B_DIGEST            integrityOuter;
    TPM2B_DIGEST            integrityInner;
    TPM2B_SENSITIVE         sensitive;
} _PRIVATE;
```

Table 2:199 - Definition of TPM2B_PRIVATE Structure *(StructuresTable())*

```
typedef union {
    struct {
        UINT16                  size;
        BYTE                    buffer[sizeof(_PRIVATE)];
    }           t;
    TPM2B       b;
} TPM2B_PRIVATE;
```

Table 2:200 - Definition of TPMS_ID_OBJECT Structure *(StructuresTable())*

```
typedef struct {
    TPM2B_DIGEST            integrityHMAC;
    TPM2B_DIGEST            encIdentity;
} TPMS_ID_OBJECT;
```

Table 2:201 - Definition of TPM2B_ID_OBJECT Structure *(StructuresTable())*

```
typedef union {
    struct {
        UINT16                  size;
        BYTE                    credential[sizeof(TPMS_ID_OBJECT)];
    }           t;
    TPM2B       b;
} TPM2B_ID_OBJECT;
```

Table 2:202 - Definition of TPM_NV_INDEX Bits *(BitsTable())*

```
typedef struct {
    unsigned    index   : 24;
    unsigned    RH_NV   : 8 ;
} TPM_NV_INDEX;
```


Table 2:203 - Definition of TPM_NT Constants *(EnumTable())*

```
typedef  UINT32            TPM_NT;
#define  TPM_NT_ORDINARY   (TPM_NT)(0x0)
#define  TPM_NT_COUNTER    (TPM_NT)(0x1)
#define  TPM_NT_BITS       (TPM_NT)(0x2)
#define  TPM_NT_EXTEND     (TPM_NT)(0x4)
#define  TPM_NT_PIN_FAIL   (TPM_NT)(0x8)
#define  TPM_NT_PIN_PASS   (TPM_NT)(0x9)
```

Table 2:204 - Definition of TPMS_NV_PIN_COUNTER_PARAMETERS Structure *(StructuresTable())*

```
typedef struct {
    UINT32                 pinCount;
    UINT32                 pinLimit;
} TPMS_NV_PIN_COUNTER_PARAMETERS;
```

Table 2:205 - Definition of TPMA_NV Bits *(BitsTable())*

```
typedef struct {
    unsigned    TPMA_NV_PPWRITE          : 1 ;
    unsigned    TPMA_NV_OWNERWRITE       : 1 ;
    unsigned    TPMA_NV_AUTHWRITE        : 1 ;
    unsigned    TPMA_NV_POLICYWRITE      : 1 ;
    unsigned    TPM_NT                   : 4 ;
    unsigned    Reserved_at_bit_8        : 2 ;
    unsigned    TPMA_NV_POLICY_DELETE    : 1 ;
    unsigned    TPMA_NV_WRITELOCKED      : 1 ;
    unsigned    TPMA_NV_WRITEALL         : 1 ;
    unsigned    TPMA_NV_WRITEDEFINE      : 1 ;
    unsigned    TPMA_NV_WRITE_STCLEAR    : 1 ;
    unsigned    TPMA_NV_GLOBALLOCK       : 1 ;
    unsigned    TPMA_NV_PPREAD           : 1 ;
    unsigned    TPMA_NV_OWNERREAD        : 1 ;
    unsigned    TPMA_NV_AUTHREAD         : 1 ;
    unsigned    TPMA_NV_POLICYREAD       : 1 ;
    unsigned    Reserved_at_bit_20       : 5 ;
    unsigned    TPMA_NV_NO_DA            : 1 ;
    unsigned    TPMA_NV_ORDERLY          : 1 ;
    unsigned    TPMA_NV_CLEAR_STCLEAR    : 1 ;
    unsigned    TPMA_NV_READLOCKED       : 1 ;
    unsigned    TPMA_NV_WRITTEN          : 1 ;
    unsigned    TPMA_NV_PLATFORMCREATE   : 1 ;
    unsigned    TPMA_NV_READ_STCLEAR     : 1 ;
} TPMA_NV;
#define IsNv_TPMA_NV_PPWRITE(attribute)                    \
        ((attribute.TPMA_NV_PPWRITE) != 0)
#define IsNv_TPMA_NV_OWNERWRITE(attribute)                 \
        ((attribute.TPMA_NV_OWNERWRITE) != 0)
#define IsNv_TPMA_NV_AUTHWRITE(attribute)                  \
        ((attribute.TPMA_NV_AUTHWRITE) != 0)
#define IsNv_TPMA_NV_POLICYWRITE(attribute)                \
        ((attribute.TPMA_NV_POLICYWRITE) != 0)
#define IsNv_TPMA_NV_POLICY_DELETE(attribute)              \
        ((attribute.TPMA_NV_POLICY_DELETE) != 0)
#define IsNv_TPMA_NV_WRITELOCKED(attribute)                \
        ((attribute.TPMA_NV_WRITELOCKED) != 0)
#define IsNv_TPMA_NV_WRITEALL(attribute)                   \
        ((attribute.TPMA_NV_WRITEALL) != 0)
#define IsNv_TPMA_NV_WRITEDEFINE(attribute)                \
        ((attribute.TPMA_NV_WRITEDEFINE) != 0)
#define IsNv_TPMA_NV_WRITE_STCLEAR(attribute)              \
        ((attribute.TPMA_NV_WRITE_STCLEAR) != 0)
#define IsNv_TPMA_NV_GLOBALLOCK(attribute)                 \
```

```
1429            ((attribute.TPMA_NV_GLOBALLOCK) != 0)
1430    #define IsNv_TPMA_NV_PPREAD(attribute)                    \
1431            ((attribute.TPMA_NV_PPREAD) != 0)
1432    #define IsNv_TPMA_NV_OWNERREAD(attribute)                 \
1433            ((attribute.TPMA_NV_OWNERREAD) != 0)
1434    #define IsNv_TPMA_NV_AUTHREAD(attribute)                  \
1435            ((attribute.TPMA_NV_AUTHREAD) != 0)
1436    #define IsNv_TPMA_NV_POLICYREAD(attribute)                \
1437            ((attribute.TPMA_NV_POLICYREAD) != 0)
1438    #define IsNv_TPMA_NV_NO_DA(attribute)                     \
1439            ((attribute.TPMA_NV_NO_DA) != 0)
1440    #define IsNv_TPMA_NV_ORDERLY(attribute)                   \
1441            ((attribute.TPMA_NV_ORDERLY) != 0)
1442    #define IsNv_TPMA_NV_CLEAR_STCLEAR(attribute)             \
1443            ((attribute.TPMA_NV_CLEAR_STCLEAR) != 0)
1444    #define IsNv_TPMA_NV_READLOCKED(attribute)                \
1445            ((attribute.TPMA_NV_READLOCKED) != 0)
1446    #define IsNv_TPMA_NV_WRITTEN(attribute)                   \
1447            ((attribute.TPMA_NV_WRITTEN) != 0)
1448    #define IsNv_TPMA_NV_PLATFORMCREATE(attribute)            \
1449            ((attribute.TPMA_NV_PLATFORMCREATE) != 0)
1450    #define IsNv_TPMA_NV_READ_STCLEAR(attribute)              \
1451            ((attribute.TPMA_NV_READ_STCLEAR) != 0)
```

Table 2:206 - Definition of TPMS_NV_PUBLIC Structure *(StructuresTable())*

```
1452    typedef struct {
1453        TPMI_RH_NV_INDEX        nvIndex;
1454        TPMI_ALG_HASH           nameAlg;
1455        TPMA_NV                 attributes;
1456        TPM2B_DIGEST            authPolicy;
1457        UINT16                  dataSize;
1458    } TPMS_NV_PUBLIC;
```

Table 2:207 - Definition of TPM2B_NV_PUBLIC Structure *(StructuresTable())*

```
1459    typedef struct {
1460        UINT16                  size;
1461        TPMS_NV_PUBLIC          nvPublic;
1462    } TPM2B_NV_PUBLIC;
```

Table 2:208 - Definition of TPM2B_CONTEXT_SENSITIVE Structure *(StructuresTable())*

```
1463    typedef union {
1464        struct {
1465            UINT16                  size;
1466            BYTE                    buffer[MAX_CONTEXT_SIZE];
1467        }           t;
1468        TPM2B       b;
1469    } TPM2B_CONTEXT_SENSITIVE;
```

Table 2:209 - Definition of TPMS_CONTEXT_DATA Structure *(StructuresTable())*

```
1470    typedef struct {
1471        TPM2B_DIGEST                integrity;
1472        TPM2B_CONTEXT_SENSITIVE     encrypted;
1473    } TPMS_CONTEXT_DATA;
```

Table 2:210 - Definition of TPM2B_CONTEXT_DATA Structure *(StructuresTable())*

```
1474    typedef union {
1475        struct {
1476            UINT16                  size;
```

```
1477            BYTE                    buffer[sizeof(TPMS_CONTEXT_DATA)];
1478        }           t;
1479        TPM2B       b;
1480    } TPM2B_CONTEXT_DATA;
```

Table 2:211 - Definition of TPMS_CONTEXT Structure *(StructuresTable())*

```
1481    typedef struct {
1482        UINT64                  sequence;
1483        TPMI_DH_CONTEXT         savedHandle;
1484        TPMI_RH_HIERARCHY       hierarchy;
1485        TPM2B_CONTEXT_DATA      contextBlob;
1486    } TPMS_CONTEXT;
```

Table 2:213 - Definition of TPMS_CREATION_DATA Structure *(StructuresTable())*

```
1487    typedef struct {
1488        TPML_PCR_SELECTION      pcrSelect;
1489        TPM2B_DIGEST            pcrDigest;
1490        TPMA_LOCALITY           locality;
1491        TPM_ALG_ID              parentNameAlg;
1492        TPM2B_NAME              parentName;
1493        TPM2B_NAME              parentQualifiedName;
1494        TPM2B_DATA              outsideInfo;
1495    } TPMS_CREATION_DATA;
```

Table 2:214 - Definition of TPM2B_CREATION_DATA Structure *(StructuresTable())*

```
1496    typedef struct {
1497        UINT16                  size;
1498        TPMS_CREATION_DATA      creationData;
1499    } TPM2B_CREATION_DATA;
1500    #endif    //  _TPM_TYPES_H_
```

5.24 VendorStrng.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-octet, vendor-specific values. The values must each be 4 octet 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

A vendor-specific string should be provided here.

```
#define       VENDOR_STRING_1       "DPA "
#define       VENDOR_STRING_2       "fTPM"
```

The vendor strings 2-4 may also be defined as needed.

```
//#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 vendor specific \
string.
#endif
```

A vendor-specific FIRMWARE_V1 is required here. It is the more significant 32-bits of a vendor-specific value indicating the version of the firmware

```
#define   FIRMWARE_V1         (0x20160511)
```

A vendor-specific FIRMWARE_V2 may be provided here. If present, it is the less significant 32-bits of the version of the firmware.

```
#define   FIRMWARE_V2         (0x00162800)
```

The following macro is just to insure that a FIRMWARE_V1 value 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
```


5.25 swap.h

```
#ifndef _SWAP_H
#define _SWAP_H
#if LITTLE_ENDIAN_TPM == YES
#define TO_BIG_ENDIAN_UINT16(i)     REVERSE_ENDIAN_16(i)
#define FROM_BIG_ENDIAN_UINT16(i)   REVERSE_ENDIAN_16(i)
#define TO_BIG_ENDIAN_UINT32(i)     REVERSE_ENDIAN_32(i)
#define FROM_BIG_ENDIAN_UINT32(i)   REVERSE_ENDIAN_32(i)
#define TO_BIG_ENDIAN_UINT64(i)     REVERSE_ENDIAN_64(i)
#define FROM_BIG_ENDIAN_UINT64(i)   REVERSE_ENDIAN_64(i)
#else
#define TO_BIG_ENDIAN_UINT16(i)     (i)
#define FROM_BIG_ENDIAN_UINT16(i)   (i)
#define TO_BIG_ENDIAN_UINT32(i)     (i)
#define FROM_BIG_ENDIAN_UINT32(i)   (i)
#define TO_BIG_ENDIAN_UINT64(i)     (i)
#define FROM_BIG_ENDIAN_UINT64(i)   (i)
#endif
#if   AUTO_ALIGN == NO
```

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

```
#define BYTE_ARRAY_TO_UINT8(b)  (uint8_t)((b)[0])
#define BYTE_ARRAY_TO_UINT16(b) ByteArrayToUint16((BYTE *)(b))
#define BYTE_ARRAY_TO_UINT32(b) ByteArrayToUint32((BYTE *)(b))
#define BYTE_ARRAY_TO_UINT64(b) ByteArrayToUint64((BYTE *)(b))
#define UINT8_TO_BYTE_ARRAY(i, b) ((b)[0] = (uint8_t)(i))
#define UINT16_TO_BYTE_ARRAY(i, b)  Uint16ToByteArray((i), (BYTE *)(b))
#define UINT32_TO_BYTE_ARRAY(i, b)  Uint32ToByteArray((i), (BYTE *)(b))
#define UINT64_TO_BYTE_ARRAY(i, b)  Uint64ToByteArray((i), (BYTE *)(b))
#else // AUTO_ALIGN
#if BIG_ENDIAN_TPM
```

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

```
#define BYTE_ARRAY_TO_UINT8(b)        *((uint8_t  *)(b))
#define BYTE_ARRAY_TO_UINT16(b)       *((uint16_t *)(b))
#define BYTE_ARRAY_TO_UINT32(b)       *((uint32_t *)(b))
#define BYTE_ARRAY_TO_UINT64(b)       *((uint64_t *)(b))
```

Disaggregate a UINT into a byte array

```
#define UINT8_TO_BYTE_ARRAY(i, b)   {*((uint8_t  *)(b)) = (i);}
#define UINT16_TO_BYTE_ARRAY(i, b)  {*((uint16_t *)(b)) = (i);}
#define UINT32_TO_BYTE_ARRAY(i, b)  {*((uint32_t *)(b)) = (i);}
#define UINT64_TO_BYTE_ARRAY(i, b)  {*((uint64_t *)(b)) = (i);}
#else
```

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

```
#define BYTE_ARRAY_TO_UINT8(b)        *((uint8_t  *)(b))
#define BYTE_ARRAY_TO_UINT16(b)       REVERSE_ENDIAN_16(*((uint16_t *)(b)))
#define BYTE_ARRAY_TO_UINT32(b)       REVERSE_ENDIAN_32(*((uint32_t *)(b)))
#define BYTE_ARRAY_TO_UINT64(b)       REVERSE_ENDIAN_64(*((uint64_t *)(b)))
```

Disaggregate a UINT into a byte array

```
#define UINT8_TO_BYTE_ARRAY(i, b)   {*((uint8_t  *)(b)) = (i);}
```

```
43    #define UINT16_TO_BYTE_ARRAY(i, b)  {*((uint16_t *)(b)) = REVERSE_ENDIAN_16(i);}
44    #define UINT32_TO_BYTE_ARRAY(i, b)  {*((uint32_t *)(b)) = REVERSE_ENDIAN_32(i);}
45    #define UINT64_TO_BYTE_ARRAY(i, b)  {*((uint64_t *)(b)) = REVERSE_ENDIAN_64(i);}
46    #endif   // BIG_ENDIAN_TPM
47    #endif  // AUTO_ALIGN == NO
48    #endif  // _SWAP_H
```

6 Main

6.1 Introduction

The files in this section are the main processing blocks for the TPM. ExecuteCommand.c contains the entry point into the TPM code and the parsing of the command header. SessionProcess.c handles the parsing of the session area and the authorization checks, and CommandDispatch.c does the parameter unmarshaling and command dispatch.

6.2 ExecCommand.c

6.2.1 Introduction

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

6.2.2 Includes

```
1    #include "Tpm.h"
2    #include "ExecCommand_fp.h"
```

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

```
3    // #include "CommandResponseSizes_fp.h"
```

6.2.3 ExecuteCommand()

The function performs the following steps.

a) Parses the command header from input buffer.

b) Calls ParseHandleBuffer() to parse the handle area of the command.

c) Validates that each of the handles references a loaded entity.

d) Calls ParseSessionBuffer() () to:

   1) unmarshal and parse the session area;

   2) check the authorizations; and

   3) when necessary, decrypt a parameter.

e) Calls CommandDispatcher() to:

   1) unmarshal the command parameters from the command buffer;

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

   3) marshal the responses into the response buffer.

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

g) Calls BuildResponseSessions() to:

   1) when necessary, encrypt a parameter

   2) build the response authorization sessions

   3) update the audit sessions and nonces

h) Calls BuildResponseHeader() to complete the construction of the response.

responseSize is set by the caller to the maximum number of bytes available in the output buffer. ExecuteCommand() will adjust the value and return the number of bytes placed in the buffer.

response is also set by the caller to indicate the buffer into which ExecuteCommand() is to place the response.

request and *response* may point to the same buffer

NOTE: As of February, 2016, the failure processing has been moved to the platform-specific code. When the TPM code encounters an unrecoverable failure, it will SET *g_inFailureMode* and call _plat__Fail(). That function should not return but may call ExecuteCommand().

```
LIB_EXPORT void
ExecuteCommand(
    uint32_t         requestSize,   // IN: command buffer size
    unsigned char   *request,       // IN: command buffer
    uint32_t        *responseSize,  // IN/OUT: response buffer size
    unsigned char   **response      // IN/OUT: response buffer
    )
{
    // Command local variables
    UINT32              commandSize;
    COMMAND             command;
    // Response local variables
    UINT32              maxResponse = *responseSize;
    TPM_RC              result;           // return code for the command
// This next function call is used in development to size the command and response
// buffers. The values printed are the sizes of the internal structures and
// not the sizes of the canonical forms of the command response structures. Also,
// the sizes do not include the tag, command.code, requestSize, or the authorization
// fields.
//CommandResponseSizes();
    // Set flags for NV access state. This should happen before any other
    // operation that may require a NV write. Note, that this needs to be done
    // even when in failure mode. Otherwise, g_updateNV would stay SET while in
    // Failure mode and the NV would be written on each call.
    g_updateNV = UT_NONE;
    g_clearOrderly = FALSE;
    if(g_inFailureMode)
    {
        // Do failure mode processing
        TpmFailureMode(requestSize, request, responseSize, response);
        return;
    }
    // Query platform to get the NV state.  The result state is saved internally
    // and will be reported by NvIsAvailable(). The reference code requires that
    // accessibility of NV does not change during the execution of a command.
    // Specifically, if NV is available when the command execution starts and then
    // is not available later when it is necessary to write to NV, then the TPM
    // will go into failure mode.
    NvCheckState();
    // Due to the limitations of the simulation, TPM clock must be explicitly
    // synchronized with the system clock whenever a command is received.
    // This function call is not necessary in a hardware TPM. However, taking
    // a snapshot of the hardware timer at the beginning of the command allows
    // the time value to be consistent for the duration of the command execution.
    TimeUpdateToCurrent();
    // Any command through this function will unceremoniously end the
    // _TPM_Hash_Data/_TPM_Hash_End sequence.
    if(g_DRTMHandle != TPM_RH_UNASSIGNED)
        ObjectTerminateEvent();
    // Get command buffer size and command buffer.
    command.parameterBuffer = request;
    command.parameterSize = requestSize;
    // Parse command header: tag, commandSize and command.code.
    // First parse the tag. The unmarshaling routine will validate
```

```
58          // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
59          result = TPMI_ST_COMMAND_TAG_Unmarshal(&command.tag,
60                                                 &command.parameterBuffer,
61                                                 &command.parameterSize);
62          if(result != TPM_RC_SUCCESS)
63              goto Cleanup;
64          // Unmarshal the commandSize indicator.
65          result = UINT32_Unmarshal(&commandSize,
66                                    &command.parameterBuffer,
67                                    &command.parameterSize);
68          if(result != TPM_RC_SUCCESS)
69              goto Cleanup;
70          // On a TPM that receives bytes on a port, the number of bytes that were
71          // received on that port is requestSize it must be identical to commandSize.
72          // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
73          // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
74          // as the input processing (the function that receives the command bytes and
75          // places them in the input buffer) would likely have the input truncated when
76          // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
77          if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
78          {
79              result = TPM_RC_COMMAND_SIZE;
80              goto Cleanup;
81          }
82          // Unmarshal the command code.
83          result = TPM_CC_Unmarshal(&command.code, &command.parameterBuffer,
84                                    &command.parameterSize);
85          if(result != TPM_RC_SUCCESS)
86              goto Cleanup;
87          // Check to see if the command is implemented.
88          command.index = CommandCodeToCommandIndex(command.code);
89          if(UNIMPLEMENTED_COMMAND_INDEX == command.index)
90          {
91              result = TPM_RC_COMMAND_CODE;
92              goto Cleanup;
93          }
94      #if  FIELD_UPGRADE_IMPLEMENTED  == YES
95          // If the TPM is in FUM, then the only allowed command is
96          // TPM_CC_FieldUpgradeData.
97          if(IsFieldUgradeMode() && (command.code != TPM_CC_FieldUpgradeData))
98          {
99              result = TPM_RC_UPGRADE;
100             goto Cleanup;
101         }
102         else
103     #endif
104         // Excepting FUM, the TPM only accepts TPM2_Startup() after
105         // _TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
106         // is no longer allowed.
107             if((!TPMIsStarted() && command.code != TPM_CC_Startup)
108                || (TPMIsStarted() && command.code == TPM_CC_Startup))
109             {
110                 result = TPM_RC_INITIALIZE;
111                 goto Cleanup;
112             }
113     // Start regular command process.
114         NvIndexCacheInit();
115         // Parse Handle buffer.
116         result = ParseHandleBuffer(&command);
117         if(result != TPM_RC_SUCCESS)
118             goto Cleanup;
119         // All handles in the handle area are required to reference TPM-resident
120         // entities.
121         result = EntityGetLoadStatus(&command);
122         if(result != TPM_RC_SUCCESS)
123             goto Cleanup;
```



```
124         // Authorization session handling for the command.
125         ClearCpRpHashes(&command);
126         if(command.tag == TPM_ST_SESSIONS)
127         {
128             // Find out session buffer size.
129             result = UINT32_Unmarshal((UINT32 *)&command.authSize,
130                                       &command.parameterBuffer,
131                                       &command.parameterSize);
132             if(result != TPM_RC_SUCCESS)
133                 goto Cleanup;
134             // Perform sanity check on the unmarshaled value. If it is smaller than
135             // the smallest possible session or larger than the remaining size of
136             // the command, then it is an error. NOTE: This check could pass but the
137             // session size could still be wrong. That will be determined after the
138             // sessions are unmarshaled.
139             if(command.authSize < 9
140                || command.authSize > command.parameterSize)
141             {
142                 result = TPM_RC_SIZE;
143                 goto Cleanup;
144             }
145             command.parameterSize -= command.authSize;
146             // The actions of ParseSessionBuffer() are described in the introduction.
147             // As the sessions are parsed command.parameterBuffer is advanced so, on a
148             // successful return, command.parameterBuffer should be pointing at the
149             // first byte of the parameters.
150             result = ParseSessionBuffer(&command);
151             if(result != TPM_RC_SUCCESS)
152                 goto Cleanup;
153         }
154         else
155         {
156             command.authSize = 0;
157             // The command has no authorization sessions.
158             // If the command requires authorizations, then CheckAuthNoSession() will
159             // return an error.
160             result = CheckAuthNoSession(&command);
161             if(result != TPM_RC_SUCCESS)
162                 goto Cleanup;
163         }
164         // Set up the response buffer pointers. CommandDispatch will marshal the
165         // response parameters starting at the address in command.responseBuffer.
166     //    *response = MemoryGetResponseBuffer(command.index);
167         // leave space for the command header
168         command.responseBuffer = *response + STD_RESPONSE_HEADER;
169         // leave space for the parameter size field if needed
170         if(command.tag == TPM_ST_SESSIONS)
171             command.responseBuffer += sizeof(UINT32);
172         if(IsHandleInResponse(command.index))
173             command.responseBuffer += sizeof(TPM_HANDLE);
174         // CommandDispatcher returns a response handle buffer and a response parameter
175         // buffer if it succeeds. It will also set the parameterSize field in the
176         // buffer if the tag is TPM_RC_SESSIONS.
177         result = CommandDispatcher(&command);
178         if(result != TPM_RC_SUCCESS)
179             goto Cleanup;
180         // Build the session area at the end of the parameter area.
181         BuildResponseSession(&command);
182     Cleanup:
183         if(g_clearOrderly == TRUE
184             && NV_IS_ORDERLY)
185         {
186     #ifdef USE_DA_USED
187             gp.orderlyState = g_daUsed ? SU_DA_USED_VALUE : SU_NONE_VALUE;
188     #else
189             gp.orderlyState = SU_NONE_VALUE;
```

```
190 #endif
191         NV_SYNC_PERSISTENT(orderlyState);
192     }
193     // This implementation loads an "evict" object to a transient object slot in
194     // RAM whenever an "evict" object handle is used in a command so that the
195     // access to any object is the same. These temporary objects need to be
196     // cleared from RAM whether the command succeeds or fails.
197     ObjectCleanupEvict();
198     // The parameters and sessions have been marshaled. Now tack on the header and
199     // set the sizes
200     BuildResponseHeader(&command, *response, result);
201     // Try to commit all the writes to NV if any NV write happened during this
202     // command execution. This check should be made for both succeeded and failed
203     // commands, because a failed one may trigger a NV write in DA logic as well.
204     // This is the only place in the command execution path that may call the NV
205     // commit. If the NV commit fails, the TPM should be put in failure mode.
206     if((g_updateNV != UT_NONE) && !g_inFailureMode)
207     {
208         if(g_updateNV == UT_ORDERLY)
209             NvUpdateIndexOrderlyData();
210         if(!NvCommit())
211             FAIL(FATAL_ERROR_INTERNAL);
212         g_updateNV = UT_NONE;
213     }
214     pAssert((UINT32)command.parameterSize <= maxResponse);
215     // Clear unused bits in response buffer.
216     MemorySet(*response + *responseSize, 0, maxResponse - *responseSize);
217     // as a final act, and not before, update the response size.
218     *responseSize = (UINT32)command.parameterSize;
219     return;
220 }
```

6.3 CommandDispatcher.c

6.3.1 Introduction

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;

NOTE Unlike other unmarshaling functions, *parmBufferStart* does not advance. *parmBufferSize* Is reduced.

- 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.

NOTE 1 The output buffer is the return from the MemoryGetResponseBuffer() function. It includes the header, handles, response parameters, and authorization area. *respParmSize* is the response parameter size, and does not include the header, handles, or authorization area.

NOTE 2 The reference implementation is permitted to do compare operations over a union as a byte array. Therefore, the command parameter *in* structure must be initialized (e.g., zeroed) before unmarshaling so that the compare operation is valid in cases where some bytes are unused.

6.3.2 Includes

```
#include "Tpm.h"
#ifdef  TABLE_DRIVEN_DISPATCH  //%
typedef TPM_RC(NoFlagFunction)(void *target, BYTE **buffer, INT32 *size);
typedef TPM_RC(FlagFunction)(void *target, BYTE **buffer, INT32 *size, BOOL flag);
typedef FlagFunction *UNMARSHAL_t;
typedef INT16(MarshalFunction)(void *source, BYTE **buffer, INT32 *size);
typedef MarshalFunction *MARSHAL_t;
typedef TPM_RC(COMMAND_NO_ARGS)(void);
typedef TPM_RC(COMMAND_IN_ARG)(void *in);
typedef TPM_RC(COMMAND_OUT_ARG)(void *out);
typedef TPM_RC(COMMAND_INOUT_ARG)(void *in, void *out);
typedef union
{
    COMMAND_NO_ARGS         *noArgs;
    COMMAND_IN_ARG          *inArg;
    COMMAND_OUT_ARG         *outArg;
    COMMAND_INOUT_ARG       *inOutArg;
} COMMAND_t;
```

This structure is used by ParseHandleBuffer() and CommandDispatcher(). The parameters in this structure are unique for each command. The parameters are:

command --	**holds the address of the command processing function that is called by Command Dispatcher.**
inSize --	this is the size of the command-dependent input structure. The input structure holds the unmarshaled handles and command parameters. If the command takes no arguments (handles or parameters) then *inSize* will have a value of 0.
outSize --	this is the size of the command-dependent output structure. The output structure holds the results of the command in an unmarshaled form. When command processing is completed, these values are marshaled into the output buffer. It is always the case that the unmarshaled version of an output structure is larger then the marshaled version. This is because the marshaled version contains the exact same number of significant bytes but with padding removed. *typesOffsets* -- this parameter points to the list of data types that are to be marshaled or unmarshaled. The list of types follows the *offsets* array. The offsets array is variable sized so the *typesOffset* filed is necessary for the handle and command processing to be able to find the types that are being handled. The *offsets* array may be empty. The types structure is described below.
offsets --	this is an array of offsets of each of the parameters in the command or response. When processing the command parameters (not handles) the list contains the offset of the next parameter. For example, if the first command parameter has a size of 4 and there is a second command parameter, then the offset would be 4, indicating that the second parameter starts at 4. If the second parameter has a size of 8, and there is a third parameter, then the second entry in offsets is 12 (4 for the first parameter and 8 for the second). An offset value of 0 in the list indicates the start of the response parameter list. When CommandDispatcher() hits this value, it will stop unmarshaling the parameters and call 'command.' If a command has no response parameters and only one command parameter, then offsets can be an empty list.

```
19  typedef struct
20  {
21      COMMAND_t       command;        // Address of the command
22      UINT16          inSize;         // Maximum size of the input structure
23      UINT16          outSize;        // Maximum size of the output structure
24      UINT16          typesOffset;    // address of the types field
25      UINT16          offsets[1];
26  } COMMAND_DESCRIPTOR_t;
```

The types list is an encoded byte array. The byte value has two parts. The most significant bit is used when a parameter takes a flag and indicates if the flag should be SET or not. The remaining 7 bits are an index into an array of addresses of marshaling and unmarshaling functions. The array of functions is divided into 6 sections with a value assigned to denote the start of that section (and the end of the previous section). The defined offset values for each section are:

a) 0 -- unmarshaling for handles that do not take flags

b) HANDLE_FIRST_FLAG_TYPE -- unmarshaling for handles that take flags

c) PARAMETER_FIRST_TYPE -- unmarshaling for parameters that do not take flags

d) PARAMETER_FIRST_FLAG_TYPE -- unmarshaling for parameters that take flags

e) PARAMETER_LAST_TYPE + 1 -- marshaling for handles

f) RESPONSE_PARAMETER_FIRST_TYPE -- marshaling for parameters
RESPONSE_PARAMETER_LAST_TYPE is the last value in the list of marshaling and unmarshaling functions.

The types list is constructed with a byte of 0xff at the end of the command parameters and with an 0xff at the end of the response parameters.

```
#ifdef COMPRESSED_LISTS
#   define PAD_LIST 0
#else
#   define PAD_LIST 1
#endif
#define _COMMAND_TABLE_DISPATCH_
#include "CommandDispatchData.h"
#define TEST_COMMAND    TPM_CC_Startup
#define NEW_CC
#else
#include "Commands.h"
#endif
```

6.3.3 Marshal/Unmarshal Functions

6.3.4 ParseHandleBuffer()

This is the table-driven version of the handle buffer unmarshaling code

```
TPM_RC
ParseHandleBuffer(
    COMMAND                 *command
    )
{
    TPM_RC                   result;
#if defined TABLE_DRIVEN_DISPATCH
    COMMAND_DESCRIPTOR_t    *desc;
    BYTE                    *types;
    BYTE                     type;
    BYTE                     dtype;
    // Make sure that nothing strange has happened
    pAssert(command->index
            < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
    // Get the address of the descriptor for this command
    desc = s_CommandDataArray[command->index];
    pAssert(desc != NULL);
    // Get the associated list of unmarshaling data types.
    types = &((BYTE *)desc)[desc->typesOffset];
//    if(s_ccAttr[commandIndex].commandIndex == TEST_COMMAND)
//        commandIndex = commandIndex;
    // No handles yet
    command->handleNum = 0;
    // Get the first type value
    for(type = *types++;
        // check each byte to make sure that we have not hit the start
        // of the parameters
    (dtype = (type & 0x7F)) < PARAMETER_FIRST_TYPE;
    // get the next type
        type = *types++)
    {
        // See if unmarshaling of this handle type requires a flag
        if(dtype < HANDLE_FIRST_FLAG_TYPE)
        {
            // Look up the function to do the unmarshaling
            NoFlagFunction  *f = (NoFlagFunction *)UnmarshalArray[dtype];
            // call it
            result = f(&(command->handles[command->handleNum]),
                       &command->parameterBuffer,
                       &command->parameterSize);
        }
```

```
80          else
81          {
82              //  Look up the function
83              FlagFunction    *f = UnmarshalArray[dtype];
84              // Call it setting the flag to the appropriate value
85              result = f(&(command->handles[command->handleNum]),
86                         &command->parameterBuffer,
87                         &command->parameterSize, (type & 0x80) != 0);
88          }
89          // Got a handle
90          // We do this first so that the match for the handle offset of the
91          // response code works correctly.
92          command->handleNum += 1;
93          if(result != TPM_RC_SUCCESS)
94              // if the unmarshaling failed, return the response code with the
95              // handle indication set
96              return result + TPM_RC_H + (command->handleNum * TPM_RC_1);
97      }
98  #else
99      BYTE            **handleBufferStart = &command->parameterBuffer;
100     INT32           *bufferRemainingSize = &command->parameterSize;
101     TPM_HANDLE      *handles = &command->handles[0];
102     UINT32          *handleCount = &command->handleNum;
103     switch(command->code)
104     {
105 #include "HandleProcess.h"
106 #undef handles
107         default:
108             FAIL(FATAL_ERROR_INTERNAL);
109             break;
110     }
111 #endif
112     return TPM_RC_SUCCESS;
113 }
114 TPM_RC
115 CommandDispatcher(
116     COMMAND                 *command
117     )
118 {
119 #if !defined TABLE_DRIVEN_DISPATCH
120     TPM_RC       result;
121     BYTE        *parm_buffer = command->parameterBuffer;
122     INT32       *paramBufferSize = &command->parameterSize;
123     BYTE        *responseBuffer = command->responseBuffer;
124     INT32        resHandleSize = 0;
125     INT32       *responseHandleSize = &resHandleSize;
126     INT32       *respParmSize = &command->parameterSize;
127     BYTE         rHandle[4];
128     BYTE        *responseHandle = &rHandle[0];
129     INT32        rSize = MAX_RESPONSE_SIZE; // used to make sure that the marshaling
130                                             // operation does not run away due to
131                                             // bad parameter in the function
132                                             // output.
133     TPM_HANDLE  *handles = &command->handles[0];
134     switch(GetCommandCode(command->index))
135     {
136 #include "CommandDispatcher.h"
137         default:
138             FAIL(FATAL_ERROR_INTERNAL);
139             break;
140     }
141     command->responseBuffer = responseBuffer;
142     command->handleNum = 0;
143     // The response handle was marshaled into rHandle. 'Unmarshal' it into
144     // handles.
145     if(*responseHandleSize > 0)
```

```
146         {
147             command->handles[command->handleNum++] = BYTE_ARRAY_TO_UINT32(rHandle);
148         }
149         return TPM_RC_SUCCESS;
150     #else
151         COMMAND_DESCRIPTOR_t    *desc;
152         BYTE                    *types;
153         BYTE                     type;
154         UINT16                  *offsets;
155         UINT16                   offset = 0;
156         UINT32                   maxInSize;
157         BYTE                    *commandIn;
158         INT32                    maxOutSize;
159         BYTE                    *commandOut;
160         COMMAND_t                cmd;
161         TPM_HANDLE              *handles;
162         UINT32                   hasInParameters = 0;
163         BOOL                     hasOutParameters = FALSE;
164         UINT32                   pNum = 0;
165         BYTE                     dType;     // dispatch type
166         TPM_RC                   result;
167         // Get the address of the descriptor for this command
168         pAssert(command->index
169                 < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
170         desc = s_CommandDataArray[command->index];
171         // Get the list of parameter types for this command
172         pAssert(desc != NULL);
173         types = &((BYTE *)desc)[desc->typesOffset];
174         // Get a pointer to the list of parameter offsets
175         offsets = &desc->offsets[0];
176         // pointer to handles
177         handles = command->handles;
178         // Get the size required to hold all the unmarshaled parameters for this command
179         maxInSize = desc->inSize;
180         // and the size of the output parameter structure returned by this command
181         maxOutSize = desc->outSize;
182         // Get a buffer for the input parameters
183         commandIn = MemoryGetActionInputBuffer(maxInSize);
184         // And the output parameters
185         commandOut = (BYTE *)MemoryGetActionOutputBuffer((UINT32)maxOutSize);
186         // Get the address of the action code dispatch
187         cmd = desc->command;
188         // Copy any handles into the input buffer
189         for(type = *types++; (type & 0x7F) < PARAMETER_FIRST_TYPE; type = *types++)
190         {
191             // 'offset' was initialized to zero so the first unmarshaling will always
192             // be to the start of the data structure
193             *(TPM_HANDLE *)&(commandIn[offset]) = *handles++;
194             // This check is used so that we don't have to add an additional offset
195             // value to the offsets list to correspond to the stop value in the
196             // command parameter list.
197             if(*types != 0xFF)
198                 offset = *offsets++;
199             maxInSize -= sizeof(TPM_HANDLE);
200             hasInParameters++;
201         }
202         // Exit loop with type containing the last value read from types
203         // maxInSize has the amount of space remaining in the command action input
204         // buffer. Make sure that we don't have more data to unmarshal than is going to
205         // fit.
206         // type contains the last value read from types so it is not necessary to
207         // reload it, which is good because *types now points to the next value
208         for(; (dType = (type & 0x7F)) <= PARAMETER_LAST_TYPE; type = *types++)
209         {
210             pNum++;
211             if(dType < PARAMETER_FIRST_FLAG_TYPE)
```

```
212             {
213                 NoFlagFunction      *f = (NoFlagFunction *)UnmarshalArray[dType];
214                 result = f(&commandIn[offset], &command->parameterBuffer,
215                            &command->parameterSize);
216             }
217             else
218             {
219                 FlagFunction        *f = UnmarshalArray[dType];
220                 result = f(&commandIn[offset], &command->parameterBuffer,
221                            &command->parameterSize,
222                            (type & 0x80) != 0);
223             }
224             if(result != TPM_RC_SUCCESS)
225                 return result + TPM_RC_P + (TPM_RC_1 * pNum);
226             // This check is used so that we don't have to add an additional offset
227             // value to the offsets list to correspond to the stop value in the
228             // command parameter list.
229             if(*types != 0xFF)
230                 offset = *offsets++;
231             hasInParameters++;
232         }
233         // Should have used all the bytes in the input
234         if(command->parameterSize != 0)
235             return TPM_RC_SIZE;
236         // The command parameter unmarshaling stopped when it hit a value that was out
237         // of range for unmarshaling values and left *types pointing to the first
238         // marshaling type. If that type happens to be the STOP value, then there
239         // are no response parameters. So, set the flag to indicate if there are
240         // output parameters.
241         hasOutParameters = *types != 0xFF;
242         // There are four cases for calling, with and without input parameters and with
243         // and without output parameters.
244         if(hasInParameters > 0)
245         {
246             if(hasOutParameters)
247                 result = cmd.inOutArg(commandIn, commandOut);
248             else
249                 result = cmd.inArg(commandIn);
250         }
251         else
252         {
253             if(hasOutParameters)
254                 result = cmd.outArg(commandOut);
255             else
256                 result = cmd.noArgs();
257         }
258         if(result != TPM_RC_SUCCESS)
259             return result;
260         // Offset in the marshaled output structure
261         offset = 0;
262         // Process the return handles, if any
263         command->handleNum = 0;
264         // Could make this a loop to process output handles but there is only ever
265         // one handle in the outputs (for now).
266         type = *types++;
267         if((dType = (type & 0x7F)) < RESPONSE_PARAMETER_FIRST_TYPE)
268         {
269             // The out->handle value was referenced as TPM_HANDLE in the
270             // action code so it has to be properly aligned.
271             command->handles[command->handleNum++] =
272                 *((TPM_HANDLE *)&(commandOut[offset]));
273             maxOutSize -= sizeof(UINT32);
274             type = *types++;
275             offset = *offsets++;
276         }
277         // Use the size of the command action output buffer as the maximum for the
```

```
278         // number of bytes that can get marshaled. Since the marshaling code has
279         // no pointers to data, all of the data being returned has to be in the
280         // command action output buffer. If we try to marshal more bytes than
281         // could fit into the output buffer, we need to fail.
282         for(;(dType = (type & 0x7F)) <= RESPONSE_PARAMETER_LAST_TYPE
283             && !g_inFailureMode; type = *types++)
284         {
285             const MARSHAL_t     f = MarshalArray[dType];
286             command->parameterSize += f(&commandOut[offset], &command->responseBuffer,
287                                         &maxOutSize);
288             offset = *offsets++;
289         }
290         return (maxOutSize < 0) ? TPM_RC_FAILURE : TPM_RC_SUCCESS;
291     #endif
292     }
```

6.4 SessionProcess.c

6.4.1 Introduction

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

6.4.2 Includes and Data Definitions

```
1   #define SESSION_PROCESS_C
2   #include "Tpm.h"
```

6.4.3 Authorization Support Functions

6.4.3.1 IsDAExempted()

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

a) a primary seed handle,

b) an object with *noDA* bit SET,

c) an NV Index with TPMA_NV_NO_DA bit SET, or

d) a PCR handle.

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

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

```
31              result = TRUE;
32              break;
33          default:
34              break;
35      }
36      return result;
37  }
```

6.4.3.2 IncrementLockout()

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

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

```
38  static TPM_RC
39  IncrementLockout(
40      UINT32          sessionIndex
41      )
42  {
43      TPM_HANDLE      handle = s_associatedHandles[sessionIndex];
44      TPM_HANDLE      sessionHandle = s_sessionHandles[sessionIndex];
45      SESSION         *session = NULL;
46      // Don't increment lockout unless the handle associated with the session
47      // is DA protected or the session is bound to a DA protected entity.
48      if(sessionHandle == TPM_RS_PW)
49      {
50          if(IsDAExempted(handle))
51              return TPM_RC_BAD_AUTH;
52      }
53      else
54      {
55          session = SessionGet(sessionHandle);
56          // If the session is bound to lockout, then use that as the relevant
57          // handle. This means that an authorization failure with a bound session
58          // bound to lockoutAuth will take precedence over any other
59          // lockout check
60          if(session->attributes.isLockoutBound == SET)
61              handle = TPM_RH_LOCKOUT;
62          if(session->attributes.isDaBound == CLEAR
63             && (IsDAExempted(handle) || session->attributes.includeAuth == CLEAR))
64             // If the handle was changed to TPM_RH_LOCKOUT, this will not return
65             // TPM_RC_BAD_AUTH
66              return TPM_RC_BAD_AUTH;
67      }
68      if(handle == TPM_RH_LOCKOUT)
69      {
70          pAssert(gp.lockOutAuthEnabled == TRUE);
71          // lockout is no longer enabled
72          gp.lockOutAuthEnabled = FALSE;
73          // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
74          // the lockout authorization will be reset at startup.
75          if(gp.lockoutRecovery != 0)
76          {
77              if(NV_IS_AVAILABLE)
78                  // Update NV.
79                  NV_SYNC_PERSISTENT(lockOutAuthEnabled);
80              else
81                  // No NV access for now. Put the TPM in pending mode.
```

```
                s_DAPendingOnNV = TRUE;
            }
        }
        else
        {
            if(gp.recoveryTime != 0)
            {
                gp.failedTries++;
                if(NV_IS_AVAILABLE)
                    // Record changes to NV. NvWrite will SET g_updateNV
                    NV_SYNC_PERSISTENT(failedTries);
                else
                    // No NV access for now.  Put the TPM in pending mode.
                    s_DAPendingOnNV = TRUE;
            }
        }
        // Register a DA failure and reset the timers.
        DARegisterFailure(handle);
        return TPM_RC_AUTH_FAIL;
    }
```

6.4.3.3 IsSessionBindEntity()

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

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

```
    static BOOL
    IsSessionBindEntity(
        TPM_HANDLE       associatedHandle,  // IN: handle to be authorized
        SESSION         *session            // IN: associated session
        )
    {
        TPM2B_NAME     entity;             // The bind value for the entity
        // If the session is not bound, return FALSE.
        if(session->attributes.isBound)
        {
            // Compute the bind value for the entity.
            SessionComputeBoundEntity(associatedHandle, &entity);
            // Compare to the bind value in the session.
            return MemoryEqual2B(&entity.b, &session->u1.boundEntity.b);
        }
        return FALSE;
    }
```

6.4.3.4 IsPolicySessionRequired()

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

a) the command requires the DUP role,

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

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

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

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

```
119  static BOOL
120  IsPolicySessionRequired(
121      COMMAND_INDEX    commandIndex,  // IN: command index
122      UINT32           sessionIndex   // IN: session index
123      )
124  {
125      AUTH_ROLE       role = CommandAuthRole(commandIndex, sessionIndex);
126      TPM_HT          type = HandleGetType(s_associatedHandles[sessionIndex]);
127      if(role == AUTH_DUP)
128          return TRUE;
129      if(role == AUTH_ADMIN)
130      {
131          // We allow an exception for ADMIN role in a transient object. If the object
132          // allows ADMIN role actions with authorization, then policy is not
133          // required. For all other cases, there is no way to override the command
134          // requirement that a policy be used
135          if(type == TPM_HT_TRANSIENT)
136          {
137              OBJECT      *object = HandleToObject(s_associatedHandles[sessionIndex]);
138              if(object->publicArea.objectAttributes.adminWithPolicy == CLEAR)
139                  return FALSE;
140          }
141          return TRUE;
142      }
143      if(type == TPM_HT_PCR)
144      {
145          if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
146          {
147              TPM2B_DIGEST        policy;
148              TPMI_ALG_HASH       policyAlg;
149              policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
150                                           &policy);
151              if(policyAlg != TPM_ALG_NULL)
152                  return TRUE;
153          }
154      }
155      return FALSE;
156  }
```

6.4.3.5 IsAuthValueAvailable()

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

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

```
157  static BOOL
158  IsAuthValueAvailable(
159      TPM_HANDLE       handle,        // IN: handle of entity
160      COMMAND_INDEX    commandIndex,  // IN: command index
161      UINT32           sessionIndex   // IN: session index
162      )
163  {
164      BOOL             result = FALSE;
165      switch(HandleGetType(handle))
166      {
167          case TPM_HT_PERMANENT:
168              switch(handle)
169              {
170                      // At this point hierarchy availability has already been
171                      // checked so primary seed handles are always available here
172                  case TPM_RH_OWNER:
173                  case TPM_RH_ENDORSEMENT:
174                  case TPM_RH_PLATFORM:
175  #ifdef VENDOR_PERMANENT
176                      // This vendor defined handle associated with the
177                      // manufacturer's shared secret
178                  case VENDOR_PERMANENT:
179  #endif
180                      // The DA checking has been performed on LockoutAuth but we
181                      // bypass the DA logic if we are using lockout policy. The
182                      // policy would allow execution to continue an lockoutAuth
183                      // could be used, even if direct use of lockoutAuth is disabled
184                  case TPM_RH_LOCKOUT:
185                      // NullAuth is always available.
186                  case TPM_RH_NULL:
187                      result = TRUE;
188                      break;
189                  default:
190                      // Otherwise authValue is not available.
191                      break;
192              }
193              break;
194          case TPM_HT_TRANSIENT:
195              // A persistent object has already been loaded and the internal
196              // handle changed.
197          {
198              OBJECT          *object;
199              object = HandleToObject(handle);
200              // authValue is always available for a sequence object.
201              // An alternative for this is to SET
202              // object->publicArea.objectAttributes.userWithAuth when the
203              // sequence is started.
204              if(ObjectIsSequence(object))
205              {
206                  result = TRUE;
207                  break;
208              }
209              // authValue is available for an object if it has its sensitive
210              // portion loaded and
211              //  1. userWithAuth bit is SET, or
212              //  2. ADMIN role is required
213              if(object->attributes.publicOnly == CLEAR
214                 && (object->publicArea.objectAttributes.userWithAuth == SET
215                     || (CommandAuthRole(commandIndex, sessionIndex) == AUTH_ADMIN
```

```
216                         &&  object->publicArea.objectAttributes.adminWithPolicy
217                         == CLEAR)))
218                     result = TRUE;
219             }
220             break;
221             case TPM_HT_NV_INDEX:
222                 // NV Index.
223             {
224                 NV_REF           locator;
225                 NV_INDEX        *nvIndex = NvGetIndexInfo(handle, &locator);
226                 TPMA_NV          nvAttributes;
227                 pAssert(nvIndex != 0);
228                 nvAttributes = nvIndex->publicArea.attributes;
229                 if(IsWriteOperation(commandIndex))
230                 {
231                     // AuthWrite can't be set for a PIN index
232                     if(nvAttributes.TPMA_NV_AUTHWRITE == SET)
233                         result = TRUE;
234                 }
235                 else
236                 {
237                     // A "read" operation
238                     // For a PIN Index, the authValue is available as long as the
239                     // Index has been written and the pinCount is less than pinLimit
240                     if(IsNvPinFailIndex(nvAttributes)
241                        || IsNvPinPassIndex(nvAttributes))
242                     {
243                         NV_PIN          pin;
244                         if(nvAttributes.TPMA_NV_WRITTEN != SET)
245                             break; // return false
246                         // get the index values
247                         pin.intVal = NvGetUINT64Data(nvIndex, locator);
248                         if(pin.pin.pinCount < pin.pin.pinLimit)
249                             result = TRUE;
250                     }
251                     // For non-PIN Indices, need to allow use of the authValue
252                     else if(nvAttributes.TPMA_NV_AUTHREAD == SET)
253                         result = TRUE;
254                 }
255             }
256             break;
257             case TPM_HT_PCR:
258                 // PCR handle.
259                 // authValue is always allowed for PCR
260                 result = TRUE;
261                 break;
262             default:
263                 // Otherwise, authValue is not available
264                 break;
265         }
266         return result;
267     }
```

6.4.3.6 IsAuthPolicyAvailable()

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

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

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

```
268 static BOOL
269 IsAuthPolicyAvailable(
270     TPM_HANDLE       handle,        // IN: handle of entity
271     COMMAND_INDEX    commandIndex,  // IN: command index
272     UINT32           sessionIndex   // IN: session index
273     )
274 {
275     BOOL            result = FALSE;
276     switch(HandleGetType(handle))
277     {
278         case TPM_HT_PERMANENT:
279             switch(handle)
280             {
281                 // At this point hierarchy availability has already been checked.
282                 case TPM_RH_OWNER:
283                     if(gp.ownerPolicy.t.size != 0)
284                         result = TRUE;
285                     break;
286                 case TPM_RH_ENDORSEMENT:
287                     if(gp.endorsementPolicy.t.size != 0)
288                         result = TRUE;
289                     break;
290                 case TPM_RH_PLATFORM:
291                     if(gc.platformPolicy.t.size != 0)
292                         result = TRUE;
293                     break;
294                 case TPM_RH_LOCKOUT:
295                     if(gp.lockoutPolicy.t.size != 0)
296                         result = TRUE;
297                     break;
298                 default:
299                     break;
300             }
301             break;
302         case TPM_HT_TRANSIENT:
303         {
304             // Object handle.
305             // An evict object would already have been loaded and given a
306             // transient object handle by this point.
307             OBJECT  *object = HandleToObject(handle);
308             // Policy authorization is not available for an object with only
309             // public portion loaded.
310             if(object->attributes.publicOnly == CLEAR)
311             {
312                 // Policy authorization is always available for an object but
313                 // is never available for a sequence.
314                 if(!ObjectIsSequence(object))
315                     result = TRUE;
316             }
317             break;
318         }
319         case TPM_HT_NV_INDEX:
320             // An NV Index.
321         {
322             NV_INDEX         *nvIndex = NvGetIndexInfo(handle, NULL);
323             TPMA_NV           attributes = nvIndex->publicArea.attributes;
324             // If the policy size is not zero, check if policy can be used.
325             if(nvIndex->publicArea.authPolicy.t.size != 0)
326             {
```

```
                // If policy session is required for this handle, always
                // uses policy regardless of the attributes bit setting
                if(IsPolicySessionRequired(commandIndex, sessionIndex))
                    result = TRUE;
                // Otherwise, the presence of the policy depends on the NV
                // attributes.
                else if(IsWriteOperation(commandIndex))
                {
                    if(IsNv_TPMA_NV_POLICYWRITE(attributes))
                        result = TRUE;
                }
                else
                {
                    if(IsNv_TPMA_NV_POLICYREAD(attributes))
                        result = TRUE;
                }
            }
        }
        break;
        case TPM_HT_PCR:
            // PCR handle.
            if(PCRPolicyIsAvailable(handle))
                result = TRUE;
            break;
        default:
            break;
    }
    return result;
}
```

6.4.4 Session Parsing Functions

6.4.4.1 ClearCpRpHashes()

```
void
ClearCpRpHashes(
    COMMAND         *command
    )
{
#if ALG_SHA1
    command->sha1CpHash.t.size = 0;
    command->sha1RpHash.t.size = 0;
#endif
#if ALG_SHA256
    command->sha256CpHash.t.size = 0;
    command->sha256RpHash.t.size = 0;
#endif
#if ALG_SHA384
    command->sha384CpHash.t.size = 0;
    command->sha384RpHash.t.size = 0;
#endif
#if ALG_SHA512
    command->sha512CpHash.t.size = 0;
    command->sha512RpHash.t.size = 0;
#endif
#if ALG_SM3_256
    command->sm3_256CpHash.t.size = 0;
    command->sm3_256RpHash.t.size = 0;
#endif
}
```


6.4.4.2 GetCpHashPointer()

Function to get a pointer to the *cpHash* of the command

```
382 static TPM2B_DIGEST *
383 GetCpHashPointer(
384     COMMAND         *command,
385     TPMI_ALG_HASH    hashAlg
386     )
387 {
388     switch(hashAlg)
389     {
390 #if ALG_SHA1
391         case TPM_ALG_SHA1:
392             return (TPM2B_DIGEST *)&command->sha1CpHash;
393 #endif
394 #if ALG_SHA256
395         case TPM_ALG_SHA256:
396             return (TPM2B_DIGEST *)&command->sha256CpHash;
397 #endif
398 #if ALG_SHA384
399         case TPM_ALG_SHA384:
400             return (TPM2B_DIGEST *)&command->sha384CpHash;
401 #endif
402 #if ALG_SHA512
403         case TPM_ALG_SHA512:
404             return (TPM2B_DIGEST *)&command->sha512CpHash;
405 #endif
406 #if ALG_SM3_256
407         case TPM_ALG_SM3_256:
408             return (TPM2B_DIGEST *)&command->sm3_256CpHash;
409 #endif
410         default:
411             break;
412     }
413     return NULL;
414 }
```

6.4.4.3 GetRpHashPointer()

Function to get a pointer to the RpHash() of the command

```
415 static TPM2B_DIGEST *
416 GetRpHashPointer(
417     COMMAND         *command,
418     TPMI_ALG_HASH    hashAlg
419     )
420 {
421     switch(hashAlg)
422     {
423 #if ALG_SHA1
424         case TPM_ALG_SHA1:
425             return (TPM2B_DIGEST *)&command->sha1RpHash;
426 #endif
427 #if ALG_SHA256
428         case TPM_ALG_SHA256:
429             return (TPM2B_DIGEST *)&command->sha256RpHash;
430 #endif
431 #if ALG_SHA384
432         case TPM_ALG_SHA384:
433             return (TPM2B_DIGEST *)&command->sha384RpHash;
434 #endif
435 #if ALG_SHA512
436         case TPM_ALG_SHA512:
```

```
437             return (TPM2B_DIGEST *)&command->sha512RpHash;
438     #endif
439     #if ALG_SM3_256
440             case TPM_ALG_SM3_256:
441                 return (TPM2B_DIGEST *)&command->sm3_256RpHash;
442     #endif
443             default:
444                 break;
445         }
446         return NULL;
447     }
```

6.4.4.4 ComputeCpHash()

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

```
448     static TPM2B_DIGEST *
449     ComputeCpHash(
450         COMMAND         *command,       // IN: command parsing structure
451         TPMI_ALG_HASH    hashAlg        // IN: hash algorithm
452         )
453     {
454         UINT32              i;
455         HASH_STATE          hashState;
456         TPM2B_NAME          name;
457         TPM2B_DIGEST        *cpHash;
458         // cpHash = hash(commandCode [ || authName1
459         //                            [ || authName2
460         //                            [ || authName 3 ]]]
461         //                            [ || parameters])
462         // A cpHash can contain just a commandCode only if the lone session is
463         // an audit session.
464         // Get pointer to the hash value
465         cpHash = GetCpHashPointer(command, hashAlg);
466         if(cpHash->t.size == 0)
467         {
468             cpHash->t.size = CryptHashStart(&hashState, hashAlg);
469             //  Add commandCode.
470             CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
471             //  Add authNames for each of the handles.
472             for(i = 0; i < command->handleNum; i++)
473                 CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
474                                                                &name)->b);
475             //  Add the parameters.
476             CryptDigestUpdate(&hashState, command->parameterSize,
477                               command->parameterBuffer);
478             //  Complete the hash.
479             CryptHashEnd2B(&hashState, &cpHash->b);
480         }
481         return cpHash;
482     }
```

6.4.4.5 GetCpHash()

This function is used to access a precomputed *cpHash*.

```
483     static TPM2B_DIGEST *
484     GetCpHash(
485         COMMAND         *command,
486         TPMI_ALG_HASH    hashAlg
487         )
488     {
489         TPM2B_DIGEST        *cpHash = GetCpHashPointer(command, hashAlg);
```



```
490         pAssert(cpHash->t.size != 0);
491         return cpHash;
492     }
```

6.4.4.6 CompareTemplateHash()

This function computes the template hash and compares it to the session *templateHash*. It is the hash of the second parameter assuming that the command is TPM2_Create(), TPM2_CreatePrimary(), or TPM2_Derive()

```
493     static BOOL
494     CompareTemplateHash(
495         COMMAND         *command,       // IN: parsing structure
496         SESSION         *session        // IN: session data
497         )
498     {
499         BYTE                *pBuffer = command->parameterBuffer;
500         INT32                pSize = command->parameterSize;
501         TPM2B_DIGEST         tHash;
502         UINT16               size;
503     //
504         // Only try this for the three commands for which it is intended
505         if(command->code != TPM_CC_Create
506            && command->code != TPM_CC_CreatePrimary
507     #ifdef TPM_CC_CreateLoaded
508            && command->code != TPM_CC_CreateLoaded
509     #endif
510            )
511             return FALSE;
512         // Assume that the first parameter is a TPM2B and unmarshal the size field
513         // Note: this will not affect the parameter buffer and size in the calling
514         // function.
515         if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
516             return FALSE;
517         // reduce the space in the buffer.
518         // NOTE: this could make pSize go negative if the parameters are not correct but
519         // the unmarshaling code does not try to unmarshal if the remaining size is
520         // negative.
521         pSize -= size;
522         // Advance the pointer
523         pBuffer += size;
524         // Get the size of what should be the template
525         if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
526             return FALSE;
527         // See if this is reasonable
528         if(size > pSize)
529             return FALSE;
530         // Hash the template data
531         tHash.t.size = CryptHashBlock(session->authHashAlg, size, pBuffer,
532                                       sizeof(tHash.t.buffer), tHash.t.buffer);
533         return(MemoryEqual2B(&session->u1.templateHash.b, &tHash.b));
534     }
```

6.4.4.7 ComparNameHash()

This function computes the name hash and compares it to the *nameHash* in the session data.

```
535     BOOL
536     CompareNameHash(
537         COMMAND         *command,       // IN: main parsing structure
538         SESSION         *session        // IN: session structure with nameHash
539         )
```



```
{
    HASH_STATE          hashState;
    TPM2B_DIGEST        nameHash;
    UINT32              i;
    TPM2B_NAME          name;
//
    nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
    //  Add names.
    for(i = 0; i < command->handleNum; i++)
        CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
                                                       &name)->b);
    //  Complete hash.
    CryptHashEnd2B(&hashState, &nameHash.b);
    // and compare
    return MemoryEqual(session->u1.nameHash.t.buffer, nameHash.t.buffer,
                       nameHash.t.size);
}
```

6.4.4.8 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	authorization fails and increments DA failure count
TPM_RC_BAD_AUTH	authorization fails but DA does not apply

```
static TPM_RC
CheckPWAuthSession(
    UINT32          sessionIndex   // IN: index of session to be processed
    )
{
    TPM2B_AUTH      authValue;
    TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
    // Strip trailing zeros from the password.
    MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
    // Get the authValue with trailing zeros removed
    EntityGetAuthValue(associatedHandle, &authValue);
    // Success if the values are identical.
    if(MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &authValue.b))
    {
        return TPM_RC_SUCCESS;
    }
    else                    // if the digests are not identical
    {
        // Invoke DA protection if applicable.
        return IncrementLockout(sessionIndex);
    }
}
```

6.4.4.9 ComputeCommandHMAC()

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

```
static TPM2B_DIGEST *
ComputeCommandHMAC(
    COMMAND         *command,       // IN: primary control structure
    UINT32           sessionIndex,  // IN: index of session to be processed
    TPM2B_DIGEST    *hmac           // OUT: authorization HMAC
    )
```



```
585 {
586     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
587     TPM2B_KEY        key;
588     BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
589     BYTE            *buffer;
590     UINT32           marshalSize;
591     HMAC_STATE       hmacState;
592     TPM2B_NONCE     *nonceDecrypt;
593     TPM2B_NONCE     *nonceEncrypt;
594     SESSION         *session;
595     nonceDecrypt = NULL;
596     nonceEncrypt = NULL;
597     // Determine if extra nonceTPM values are going to be required.
598     // If this is the first session (sessionIndex = 0) and it is an authorization
599     // session that uses an HMAC, then check if additional session nonces are to be
600     // included.
601     if(sessionIndex == 0
602        && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
603     {
604         // If there is a decrypt session and if this is not the decrypt session,
605         // then an extra nonce may be needed.
606         if(s_decryptSessionIndex != UNDEFINED_INDEX
607            && s_decryptSessionIndex != sessionIndex)
608         {
609             // Will add the nonce for the decrypt session.
610             SESSION *decryptSession
611                 = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
612             nonceDecrypt = &decryptSession->nonceTPM;
613         }
614         // Now repeat for the encrypt session.
615         if(s_encryptSessionIndex != UNDEFINED_INDEX
616            && s_encryptSessionIndex != sessionIndex
617            && s_encryptSessionIndex != s_decryptSessionIndex)
618         {
619             // Have to have the nonce for the encrypt session.
620             SESSION *encryptSession
621                 = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
622             nonceEncrypt = &encryptSession->nonceTPM;
623         }
624     }
625     // Continue with the HMAC processing.
626     session = SessionGet(s_sessionHandles[sessionIndex]);
627     // Generate HMAC key.
628     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
629     // Check if the session has an associated handle and if the associated entity
630     // is the one to which the session is bound. If not, add the authValue of
631     // this entity to the HMAC key.
632     // If the session is bound to the object or the session is a policy session
633     // with no authValue required, do not include the authValue in the HMAC key.
634     // Note: For a policy session, its isBound attribute is CLEARED.
635     // Include the entity authValue if it is needed
636     if(session->attributes.includeAuth == SET)
637     {
638         TPM2B_AUTH          authValue;
639         // Get the entity authValue with trailing zeros removed
640         EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
641         // add the authValue to the HMAC key
642         MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
643     }
644      // if the HMAC key size is 0, a NULL string HMAC is allowed
645     if(key.t.size == 0
646        && s_inputAuthValues[sessionIndex].t.size == 0)
647     {
648         hmac->t.size = 0;
649         return hmac;
650     }
```

```
651      // Start HMAC
652      hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
653      //  Add cpHash
654      CryptDigestUpdate2B(&hmacState.hashState,
655                          &ComputeCpHash(command, session->authHashAlg)->b);
656      //  Add nonces as required
657      CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
658      CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
659      if(nonceDecrypt != NULL)
660          CryptDigestUpdate2B(&hmacState.hashState, &nonceDecrypt->b);
661      if(nonceEncrypt != NULL)
662          CryptDigestUpdate2B(&hmacState.hashState, &nonceEncrypt->b);
663      //  Add sessionAttributes
664      buffer = marshalBuffer;
665      marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
666                                         &buffer, NULL);
667      CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
668      // Complete the HMAC computation
669      CryptHmacEnd2B(&hmacState, &hmac->b);
670      return hmac;
671  }
```

6.4.4.10 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	authorization failure caused *failureCount* increment
TPM_RC_BAD_AUTH	authorization failure did not cause *failureCount* increment

```
672  static TPM_RC
673  CheckSessionHMAC(
674      COMMAND         *command,       // IN: primary control structure
675      UINT32           sessionIndex   // IN: index of session to be processed
676      )
677  {
678      TPM2B_DIGEST        hmac;           // authHMAC for comparing
679      // Compute authHMAC
680      ComputeCommandHMAC(command, sessionIndex, &hmac);
681      // Compare the input HMAC with the authHMAC computed above.
682      if(!MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &hmac.b))
683      {
684          // If an HMAC session has a failure, invoke the anti-hammering
685          // if it applies to the authorized entity or the session.
686          // Otherwise, just indicate that the authorization is bad.
687          return IncrementLockout(sessionIndex);
688      }
689      return TPM_RC_SUCCESS;
690  }
```

6.4.4.11 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

```
691  static TPM_RC
692  CheckPolicyAuthSession(
693      COMMAND         *command,       // IN: primary parsing structure
694      UINT32           sessionIndex   // IN: index of session to be processed
695      )
696  {
697      SESSION             *session;
698      TPM2B_DIGEST         authPolicy;
699      TPMI_ALG_HASH        policyAlg;
700      UINT8                locality;
701      // Initialize pointer to the authorization session.
702      session = SessionGet(s_sessionHandles[sessionIndex]);
703      // If the command is TPM2_PolicySecret(), make sure that
704      // either password or authValue is required
705      if(command->code == TPM_CC_PolicySecret
706         &&  session->attributes.isPasswordNeeded == CLEAR
707         &&  session->attributes.isAuthValueNeeded == CLEAR)
708          return TPM_RC_MODE;
709      // See if the PCR counter for the session is still valid.
710      if(!SessionPCRValueIsCurrent(session))
711          return TPM_RC_PCR_CHANGED;
712      // Get authPolicy.
713      policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
714                                      &authPolicy);
715      // Compare authPolicy.
716      if(!MemoryEqual2B(&session->u2.policyDigest.b, &authPolicy.b))
717          return TPM_RC_POLICY_FAIL;
718      // Policy is OK so check if the other factors are correct
719      // Compare policy hash algorithm.
720      if(policyAlg != session->authHashAlg)
721          return TPM_RC_POLICY_FAIL;
722      // Compare timeout.
723      if(session->timeout != 0)
724      {
725          // Cannot compare time if clock stop advancing.  An TPM_RC_NV_UNAVAILABLE
726          // or TPM_RC_NV_RATE error may be returned here. This doesn't mean that
727          // a new nonce will be created just that, because TPM time can't advance
728          // we can't do time-based operations.
729          RETURN_IF_NV_IS_NOT_AVAILABLE;
```



```
730             if((session->timeout < g_time)
731                || (session->epoch != g_timeEpoch))
732                 return TPM_RC_EXPIRED;
733         }
734         // If command code is provided it must match
735         if(session->commandCode != 0)
736         {
737             if(session->commandCode != command->code)
738                 return TPM_RC_POLICY_CC;
739         }
740         else
741         {
742             // If command requires a DUP or ADMIN authorization, the session must have
743             // command code set.
744             AUTH_ROLE   role = CommandAuthRole(command->index, sessionIndex);
745             if(role == AUTH_ADMIN || role == AUTH_DUP)
746                 return TPM_RC_POLICY_FAIL;
747         }
748         // Check command locality.
749         {
750             BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
751             BYTE        *buffer = sessionLocality;
752             // Get existing locality setting in canonical form
753             sessionLocality[0] = 0;   // Code analysis says that this is not initialized
754             TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
755             // See if the locality has been set
756             if(sessionLocality[0] != 0)
757             {
758                 // If so, get the current locality
759                 locality = _plat__LocalityGet();
760                 if(locality < 5)
761                 {
762                     if(((sessionLocality[0] & (1 << locality)) == 0)
763                        || sessionLocality[0] > 31)
764                         return TPM_RC_LOCALITY;
765                 }
766                 else if(locality > 31)
767                 {
768                     if(sessionLocality[0] != locality)
769                         return TPM_RC_LOCALITY;
770                 }
771                 else
772                 {
773                     // Could throw an assert here but a locality error is just
774                     // as good. It just means that, whatever the locality is, it isn't
775                     // the locality requested so...
776                     return TPM_RC_LOCALITY;
777                 }
778             }
779         } // end of locality check
780         // Check physical presence.
781         if(session->attributes.isPPRequired == SET
782            && !_plat__PhysicalPresenceAsserted())
783             return TPM_RC_PP;
784         // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
785         // DUP role for this handle.
786         if(session->u1.cpHash.b.size != 0)
787         {
788             BOOL        OK;
789             if(session->attributes.isCpHashDefined)
790                 // Compare cpHash.
791                 OK = MemoryEqual2B(&session->u1.cpHash.b,
792                                    &ComputeCpHash(command, session->authHashAlg)->b);
793             else if(session->attributes.isTemplateSet)
794                 OK = CompareTemplateHash(command, session);
795             else
```

```
796             OK = CompareNameHash(command, session);
797         if(!OK)
798             return TPM_RCS_POLICY_FAIL;
799     }
800     if(session->attributes.checkNvWritten)
801     {
802         NV_REF           locator;
803         NV_INDEX        *nvIndex;
804         // If this is not an NV index, the policy makes no sense so fail it.
805         if(HandleGetType(s_associatedHandles[sessionIndex]) != TPM_HT_NV_INDEX)
806             return TPM_RC_POLICY_FAIL;
807         // Get the index data
808         nvIndex = NvGetIndexInfo(s_associatedHandles[sessionIndex], &locator);
809         // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
810         if((IsNv_TPMA_NV_WRITTEN(nvIndex->publicArea.attributes))
811            != (session->attributes.nvWrittenState == SET))
812             return TPM_RC_POLICY_FAIL;
813     }
814     return TPM_RC_SUCCESS;
815 }
```

6.4.4.12 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

```
816 static TPM_RC
817 RetrieveSessionData(
818     COMMAND         *command        // IN: main parsing structure for command
819     )
820 {
821     int              i;
822     TPM_RC           result;
823     SESSION         *session;
824     TPM_HT           sessionType;
825     INT32            sessionIndex;
826     TPM_RC           errorIndex;
827     s_decryptSessionIndex = UNDEFINED_INDEX;
828     s_encryptSessionIndex = UNDEFINED_INDEX;
829     s_auditSessionIndex = UNDEFINED_INDEX;
830     for(sessionIndex = 0; command->authSize > 0; sessionIndex++)
831     {
832         errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
833         // If maximum allowed number of sessions has been parsed, return a size
834         // error with a session number that is larger than the number of allowed
835         // sessions
836         if(sessionIndex == MAX_SESSION_NUM)
837             return TPM_RCS_SIZE + errorIndex;
838         // make sure that the associated handle for each session starts out
839         // unassigned
840         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
841         // First parameter: Session handle.
842         result = TPMI_SH_AUTH_SESSION_Unmarshal(
843             &s_sessionHandles[sessionIndex],
844             &command->parameterBuffer,
845             &command->authSize, TRUE);
846         if(result != TPM_RC_SUCCESS)
```



```
847             return result + TPM_RC_S + g_rcIndex[sessionIndex];
848         // Second parameter: Nonce.
849         result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
850                                        &command->parameterBuffer,
851                                        &command->authSize);
852         if(result != TPM_RC_SUCCESS)
853             return result + TPM_RC_S + g_rcIndex[sessionIndex];
854         // Third parameter: sessionAttributes.
855         result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
856                                         &command->parameterBuffer,
857                                         &command->authSize);
858         if(result != TPM_RC_SUCCESS)
859             return result + TPM_RC_S + g_rcIndex[sessionIndex];
860         // Fourth parameter: authValue (PW or HMAC).
861         result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
862                                       &command->parameterBuffer,
863                                       &command->authSize);
864         if(result != TPM_RC_SUCCESS)
865             return result + errorIndex;
866         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
867         {
868             // A PWAP session needs additional processing.
869             //     Can't have any attributes set other than continueSession bit
870             if(s_attributes[sessionIndex].encrypt
871                || s_attributes[sessionIndex].decrypt
872                || s_attributes[sessionIndex].audit
873                || s_attributes[sessionIndex].auditExclusive
874                || s_attributes[sessionIndex].auditReset)
875                 return TPM_RCS_ATTRIBUTES + errorIndex;
876             //     The nonce size must be zero.
877             if(s_nonceCaller[sessionIndex].t.size != 0)
878                 return TPM_RCS_NONCE + errorIndex;
879             continue;
880         }
881         // For not password sessions...
882         // Find out if the session is loaded.
883         if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
884             return TPM_RC_REFERENCE_S0 + sessionIndex;
885         sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
886         session = SessionGet(s_sessionHandles[sessionIndex]);
887         // Check if the session is an HMAC/policy session.
888         if((session->attributes.isPolicy == SET
889             && sessionType == TPM_HT_HMAC_SESSION)
890            || (session->attributes.isPolicy == CLEAR
891                && sessionType == TPM_HT_POLICY_SESSION))
892             return TPM_RCS_HANDLE + errorIndex;
893         // Check that this handle has not previously been used.
894         for(i = 0; i < sessionIndex; i++)
895         {
896             if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
897                 return TPM_RCS_HANDLE + errorIndex;
898         }
899         // If the session is used for parameter encryption or audit as well, set
900         // the corresponding indices.
901         // First process decrypt.
902         if(s_attributes[sessionIndex].decrypt)
903         {
904             // Check if the commandCode allows command parameter encryption.
905             if(DecryptSize(command->index) == 0)
906                 return TPM_RCS_ATTRIBUTES + errorIndex;
907             // Encrypt attribute can only appear in one session
908             if(s_decryptSessionIndex != UNDEFINED_INDEX)
909                 return TPM_RCS_ATTRIBUTES + errorIndex;
910             // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
911             if(session->symmetric.algorithm == TPM_ALG_NULL)
912                 return TPM_RCS_SYMMETRIC + errorIndex;
```

```
913             // All checks passed, so set the index for the session used to decrypt
914             // a command parameter.
915             s_decryptSessionIndex = sessionIndex;
916         }
917         // Now process encrypt.
918         if(s_attributes[sessionIndex].encrypt)
919         {
920             // Check if the commandCode allows response parameter encryption.
921             if(EncryptSize(command->index) == 0)
922                 return TPM_RCS_ATTRIBUTES + errorIndex;
923             // Encrypt attribute can only appear in one session.
924             if(s_encryptSessionIndex != UNDEFINED_INDEX)
925                 return TPM_RCS_ATTRIBUTES + errorIndex;
926             // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
927             if(session->symmetric.algorithm == TPM_ALG_NULL)
928                 return TPM_RCS_SYMMETRIC + errorIndex;
929             // All checks passed, so set the index for the session used to encrypt
930             // a response parameter.
931             s_encryptSessionIndex = sessionIndex;
932         }
933         // At last process audit.
934         if(s_attributes[sessionIndex].audit)
935         {
936             // Audit attribute can only appear in one session.
937             if(s_auditSessionIndex != UNDEFINED_INDEX)
938                 return TPM_RCS_ATTRIBUTES + errorIndex;
939             // An audit session can not be policy session.
940             if(HandleGetType(s_sessionHandles[sessionIndex])
941                == TPM_HT_POLICY_SESSION)
942                 return TPM_RCS_ATTRIBUTES + errorIndex;
943             // If this is a reset of the audit session, or the first use
944             // of the session as an audit session, it doesn't matter what
945             // the exclusive state is. The session will become exclusive.
946             if(s_attributes[sessionIndex].auditReset == CLEAR
947                && session->attributes.isAudit == SET)
948             {
949                 // Not first use or reset. If auditExlusive is SET, then this
950                 // session must be the current exclusive session.
951                 if(s_attributes[sessionIndex].auditExclusive == SET
952                    && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
953                     return TPM_RC_EXCLUSIVE;
954             }
955             s_auditSessionIndex = sessionIndex;
956         }
957         // Initialize associated handle as undefined. This will be changed when
958         // the handles are processed.
959         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
960     }
961     command->sessionNum = sessionIndex;
962     return TPM_RC_SUCCESS;
963 }
```

6.4.4.13 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

```
964   static TPM_RC
965   CheckLockedOut(
966       BOOL             lockoutAuthCheck   // IN: TRUE if checking is for lockoutAuth
967       )
968   {
969       // If NV is unavailable, and current cycle state recorded in NV is not
970       // SU_NONE_VALUE, refuse to check any authorization because we would
971       // not be able to handle a DA failure.
972       if(!NV_IS_AVAILABLE && NV_IS_ORDERLY)
973           return g_NvStatus;
974       // Check if DA info needs to be updated in NV.
975       if(s_DAPendingOnNV)
976       {
977           // If NV is accessible,
978           RETURN_IF_NV_IS_NOT_AVAILABLE;
979           // ... write the pending DA data and proceed.
980           NV_SYNC_PERSISTENT(lockOutAuthEnabled);
981           NV_SYNC_PERSISTENT(failedTries);
982           s_DAPendingOnNV = FALSE;
983       }
984       // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
985       // is disabled...
986       if(lockoutAuthCheck)
987       {
988           if(gp.lockOutAuthEnabled == FALSE)
989               return TPM_RC_LOCKOUT;
990       }
991       else
992       {
993           // ... or if the number of failed tries has been maxed out.
994           if(gp.failedTries >= gp.maxTries)
995               return TPM_RC_LOCKOUT;
996   #ifdef USE_DA_USED
997              // If the daUsed flag is not SET, then no DA validation until the
998           // daUsed state is written to NV
999           if(!g_daUsed)
1000          {
1001              RETURN_IF_NV_IS_NOT_AVAILABLE;
1002              g_daUsed = TRUE;
1003              gp.orderlyState = SU_DA_USED_VALUE;
1004              NV_SYNC_PERSISTENT(orderlyState);
1005              return TPM_RC_RETRY;
1006          }
1007  #endif
1008      }
1009      return TPM_RC_SUCCESS;
1010  }
```

6.4.4.14 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

```
1011    static TPM_RC
1012    CheckAuthSession(
1013        COMMAND         *command,       // IN: primary parsing structure
1014        UINT32           sessionIndex   // IN: index of session to be processed
1015        )
1016    {
1017        TPM_RC           result = TPM_RC_SUCCESS;
1018        SESSION         *session = NULL;
1019        TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
1020        TPM_HANDLE       associatedHandle = s_associatedHandles[sessionIndex];
1021        TPM_HT           sessionHandleType = HandleGetType(sessionHandle);
1022        pAssert(sessionHandle != TPM_RH_UNASSIGNED);
1023        // Take care of physical presence
1024        if(associatedHandle == TPM_RH_PLATFORM)
1025        {
1026            // If the physical presence is required for this command, check for PP
1027            // assertion. If it isn't asserted, no point going any further.
1028            if(PhysicalPresenceIsRequired(command->index)
1029               && !_plat__PhysicalPresenceAsserted())
1030                return TPM_RC_PP;
1031        }
1032        if(sessionHandle != TPM_RS_PW)
1033        {
1034            session = SessionGet(sessionHandle);
1035            // Set includeAuth to indicate if DA checking will be required and if the
1036            // authValue will be included in any HMAC.
1037            if(sessionHandleType == TPM_HT_POLICY_SESSION)
1038            {
1039                // For a policy session, will check the DA status of the entity if either
1040                // isAuthValueNeeded or isPasswordNeeded is SET.
1041                session->attributes.includeAuth =
1042                    session->attributes.isAuthValueNeeded
1043                    || session->attributes.isPasswordNeeded;
1044            }
1045            else
1046            {
1047                // For an HMAC session, need to check unless the session
1048                // is bound.
1049                session->attributes.includeAuth =
1050                    !IsSessionBindEntity(s_associatedHandles[sessionIndex], session);
1051            }
1052        }
1053        // If the authorization session is going to use an authValue, then make sure
1054        // that access to that authValue isn't locked out.
```

```
1055        // Note: session == NULL for a PW session.
1056        if(session == NULL || session->attributes.includeAuth)
1057        {
1058            // See if entity is subject to lockout.
1059            if(!IsDAExempted(associatedHandle))
1060            {
1061                // See if in lockout
1062                result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1063                if(result != TPM_RC_SUCCESS)
1064                    return result;
1065            }
1066        }
1067        // Policy or HMAC+PW?
1068        if(sessionHandleType != TPM_HT_POLICY_SESSION)
1069        {
1070            // for non-policy session make sure that a policy session is not required
1071            if(IsPolicySessionRequired(command->index, sessionIndex))
1072                return TPM_RC_AUTH_TYPE;
1073            // The authValue must be available.
1074            // Note: The authValue is going to be "used" even if it is an EmptyAuth.
1075            // and the session is bound.
1076            if(!IsAuthValueAvailable(associatedHandle, command->index, sessionIndex))
1077                return TPM_RC_AUTH_UNAVAILABLE;
1078        }
1079        else
1080        {
1081            // ... see if the entity has a policy, ...
1082            // Note: IsAutPolciyAvalable will return FALSE if the sensitive area of the
1083            // object is not loaded
1084            if(!IsAuthPolicyAvailable(associatedHandle, command->index, sessionIndex))
1085                return TPM_RC_AUTH_UNAVAILABLE;
1086            // ... and check the policy session.
1087            result = CheckPolicyAuthSession(command, sessionIndex);
1088            if(result != TPM_RC_SUCCESS)
1089                return result;
1090        }
1091        // Check authorization according to the type
1092        if(session == NULL || session->attributes.isPasswordNeeded == SET)
1093            result = CheckPWAuthSession(sessionIndex);
1094        else
1095            result = CheckSessionHMAC(command, sessionIndex);
1096        // Do processing for PIN indices are only three possibilities for 'result' at
1097        // this point.
1098        //  TPM_RC_SUCCESS
1099        //  TPM_RC_AUTH_FAIL
1100        //  TPM_RC_BAD_AUTH
1101        // For all these cases, we would have to process a PIN index if the
1102        // authValue of the index was used for authorization.
1103        // See if we need to do anything to a PIN index
1104        if(TPM_HT_NV_INDEX == HandleGetType(associatedHandle))
1105        {
1106            NV_REF           locator;
1107            NV_INDEX        *nvIndex = NvGetIndexInfo(associatedHandle, &locator);
1108            NV_PIN           pinData;
1109            TPMA_NV          nvAttributes;
1110            pAssert(nvIndex != NULL);
1111            nvAttributes = nvIndex->publicArea.attributes;
1112            // If this is a PIN FAIL index and the value has been written
1113            // then we can update the counter (increment or clear)
1114            if(IsNvPinFailIndex(nvAttributes) && nvAttributes.TPMA_NV_WRITTEN == SET)
1115            {
1116                pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1117                if(result != TPM_RC_SUCCESS)
1118                    pinData.pin.pinCount++;
1119                else
1120                    pinData.pin.pinCount = 0;
```

```
1121                NvWriteUINT64Data(nvIndex, pinData.intVal);
1122            }
1123            // If this is a PIN PASS Index, increment if we have used the
1124            // authorization value for anything other than NV_Read.
1125            // NOTE: If the counter has already hit the limit, then we
1126            // would not get here because the authorization value would not
1127            // be available and the TPM would have returned before it gets here
1128            else if(IsNvPinPassIndex(nvAttributes)
1129                    && nvAttributes.TPMA_NV_WRITTEN == SET
1130                    &&  result == TPM_RC_SUCCESS)
1131            {
1132                // If the access is valid, then increment the use counter
1133                pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1134                pinData.pin.pinCount++;
1135                NvWriteUINT64Data(nvIndex, pinData.intVal);
1136            }
1137        }
1138        return result;
1139    }
1140    #ifdef  TPM_CC_GetCommandAuditDigest
```

6.4.4.15 CheckCommandAudit()

This function is called before the command is processed if audit is enabled for the command. It will check to see if the audit can be performed and will ensure that the *cpHash* is available for the audit.

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

```
1141    static TPM_RC
1142    CheckCommandAudit(
1143        COMMAND         *command
1144        )
1145    {
1146        // If the audit digest is clear and command audit is required, NV must be
1147        // available so that TPM2_GetCommandAuditDigest() is able to increment
1148        // audit counter. If NV is not available, the function bails out to prevent
1149        // the TPM from attempting an operation that would fail anyway.
1150        if(gr.commandAuditDigest.t.size == 0
1151           || GetCommandCode(command->index) == TPM_CC_GetCommandAuditDigest)
1152        {
1153            RETURN_IF_NV_IS_NOT_AVAILABLE;
1154        }
1155        // Make sure that the cpHash is computed for the algorithm
1156        ComputeCpHash(command, gp.auditHashAlg);
1157        return TPM_RC_SUCCESS;
1158    }
1159    #endif
```

6.4.4.16 ParseSessionBuffer()

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

Error Returns	Meaning
various	parsing failure or authorization failure

```
1160  TPM_RC
1161  ParseSessionBuffer(
1162      COMMAND         *command        // IN: the structure that contains
1163      )
1164  {
1165      TPM_RC              result;
1166      UINT32              i;
1167      INT32               size = 0;
1168      TPM2B_AUTH          extraKey;
1169      UINT32              sessionIndex;
1170      TPM_RC              errorIndex;
1171      SESSION            *session = NULL;
1172      // Check if a command allows any session in its session area.
1173      if(!IsSessionAllowed(command->index))
1174          return TPM_RC_AUTH_CONTEXT;
1175      // Default-initialization.
1176      command->sessionNum = 0;
1177      result = RetrieveSessionData(command);
1178      if(result != TPM_RC_SUCCESS)
1179          return result;
1180      // There is no command in the TPM spec that has more handles than
1181      // MAX_SESSION_NUM.
1182      pAssert(command->handleNum <= MAX_SESSION_NUM);
1183      // Associate the session with an authorization handle.
1184      for(i = 0; i < command->handleNum; i++)
1185      {
1186          if(CommandAuthRole(command->index, i) != AUTH_NONE)
1187          {
1188              // If the received session number is less than the number of handles
1189              // that requires authorization, an error should be returned.
1190              // Note: for all the TPM 2.0 commands, handles requiring
1191              // authorization come first in a command input and there are only ever
1192              // two values requiring authorization
1193              if(i > (command->sessionNum - 1))
1194                  return TPM_RC_AUTH_MISSING;
1195              // Record the handle associated with the authorization session
1196              s_associatedHandles[i] = command->handles[i];
1197          }
1198      }
1199      // Consistency checks are done first to avoid authorization failure when the
1200      // command will not be executed anyway.
1201      for(sessionIndex = 0; sessionIndex < command->sessionNum; sessionIndex++)
1202      {
1203          errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
1204          // PW session must be an authorization session
1205          if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1206          {
1207              if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1208                  return TPM_RCS_HANDLE + errorIndex;
1209              // a password session can't be audit, encrypt or decrypt
1210              if(s_attributes[sessionIndex].audit == SET
1211                 || s_attributes[sessionIndex].encrypt == SET
1212                 || s_attributes[sessionIndex].decrypt == SET)
1213                  return TPM_RCS_ATTRIBUTES + errorIndex;
1214              session = NULL;
1215          }
1216          else
1217          {
1218              session = SessionGet(s_sessionHandles[sessionIndex]);
1219              // A trial session can not appear in session area, because it cannot
1220              // be used for authorization, audit or encrypt/decrypt.
```

```
1221                if(session->attributes.isTrialPolicy == SET)
1222                    return TPM_RCS_ATTRIBUTES + errorIndex;
1223                // See if the session is bound to a DA protected entity
1224                // NOTE: Since a policy session is never bound, a policy is still
1225                // usable even if the object is DA protected and the TPM is in
1226                // lockout.
1227                if(session->attributes.isDaBound == SET)
1228                {
1229                    result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1230                    if(result != TPM_RC_SUCCESS)
1231                        return result;
1232                }
1233                // If this session is for auditing, make sure the cpHash is computed.
1234                if(s_attributes[sessionIndex].audit)
1235                    ComputeCpHash(command, session->authHashAlg);
1236            }
1237            // if the session has an associated handle, check the authorization
1238            if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1239            {
1240                result = CheckAuthSession(command, sessionIndex);
1241                if(result != TPM_RC_SUCCESS)
1242                    return RcSafeAddToResult(result, errorIndex);
1243            }
1244            else
1245            {
1246                // a session that is not for authorization must either be encrypt,
1247                // decrypt, or audit
1248                if(s_attributes[sessionIndex].audit == CLEAR
1249                   &&  s_attributes[sessionIndex].encrypt == CLEAR
1250                   &&  s_attributes[sessionIndex].decrypt == CLEAR)
1251                    return TPM_RCS_ATTRIBUTES + errorIndex;
1252                // no authValue included in any of the HMAC computations
1253                pAssert(session != NULL);
1254                session->attributes.includeAuth = CLEAR;
1255                // check HMAC for encrypt/decrypt/audit only sessions
1256                result = CheckSessionHMAC(command, sessionIndex);
1257                if(result != TPM_RC_SUCCESS)
1258                    return RcSafeAddToResult(result, errorIndex);
1259            }
1260        }
1261    #ifdef  TPM_CC_GetCommandAuditDigest
1262        // Check if the command should be audited. Need to do this before any parameter
1263        // encryption so that the cpHash for the audit is correct
1264        if(CommandAuditIsRequired(command->index))
1265        {
1266            result = CheckCommandAudit(command);
1267            if(result != TPM_RC_SUCCESS)
1268                return result;              // No session number to reference
1269        }
1270    #endif
1271        // Decrypt the first parameter if applicable. This should be the last operation
1272        // in session processing.
1273        // If the encrypt session is associated with a handle and the handle's
1274        // authValue is available, then authValue is concatenated with sessionKey to
1275        // generate encryption key, no matter if the handle is the session bound entity
1276        // or not.
1277        if(s_decryptSessionIndex != UNDEFINED_INDEX)
1278        {
1279            // If this is an authorization session, include the authValue in the
1280            // generation of the decryption key
1281            if(s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED)
1282            {
1283                EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1284                                   &extraKey);
1285            }
1286            else
```

```
1287            {
1288                extraKey.b.size = 0;
1289            }
1290            size = DecryptSize(command->index);
1291            result = CryptParameterDecryption(s_sessionHandles[s_decryptSessionIndex],
1292                                              &s_nonceCaller[s_decryptSessionIndex].b,
1293                                              command->parameterSize, (UINT16)size,
1294                                              &extraKey,
1295                                              command->parameterBuffer);
1296            if(result != TPM_RC_SUCCESS)
1297                return RcSafeAddToResult(result,
1298                                         TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1299        }
1300        return TPM_RC_SUCCESS;
1301    }
```

6.4.4.17 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 authorization

```
1302    TPM_RC
1303    CheckAuthNoSession(
1304        COMMAND         *command        // IN: command parsing structure
1305        )
1306    {
1307        UINT32 i;
1308        TPM_RC           result = TPM_RC_SUCCESS;
1309        // Check if the command requires authorization
1310        for(i = 0; i < command->handleNum; i++)
1311        {
1312            if(CommandAuthRole(command->index, i) != AUTH_NONE)
1313                return TPM_RC_AUTH_MISSING;
1314        }
1315    #ifdef  TPM_CC_GetCommandAuditDigest
1316        // Check if the command should be audited.
1317        if(CommandAuditIsRequired(command->index))
1318        {
1319            result = CheckCommandAudit(command);
1320            if(result != TPM_RC_SUCCESS)
1321                return result;
1322        }
1323    #endif
1324        // Initialize number of sessions to be 0
1325        command->sessionNum = 0;
1326        return TPM_RC_SUCCESS;
1327    }
```

6.4.5 Response Session Processing

6.4.5.1 Introduction

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

6.4.5.2 ComputeRpHash()

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

```
1328    static TPM2B_DIGEST *
1329    ComputeRpHash(
1330        COMMAND         *command,       // IN: command structure
1331        TPM_ALG_ID       hashAlg        // IN: hash algorithm to compute rpHash
1332        )
1333    {
1334        TPM2B_DIGEST    *rpHash = GetRpHashPointer(command, hashAlg);
1335        HASH_STATE       hashState;
1336        if(rpHash->t.size == 0)
1337        {
1338        //   rpHash := hash(responseCode || commandCode || parameters)
1339        // Initiate hash creation.
1340            rpHash->t.size = CryptHashStart(&hashState, hashAlg);
1341            // Add hash constituents.
1342            CryptDigestUpdateInt(&hashState, sizeof(TPM_RC), TPM_RC_SUCCESS);
1343            CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
1344            CryptDigestUpdate(&hashState, command->parameterSize,
1345                              command->parameterBuffer);
1346            // Complete hash computation.
1347            CryptHashEnd2B(&hashState, &rpHash->b);
1348        }
1349        return rpHash;
1350    }
```

6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```
1351    static void
1352    InitAuditSession(
1353        SESSION         *session        // session to be initialized
1354        )
1355    {
1356        // Mark session as an audit session.
1357        session->attributes.isAudit = SET;
1358        // Audit session can not be bound.
1359        session->attributes.isBound = CLEAR;
1360        // Size of the audit log is the size of session hash algorithm digest.
1361        session->u2.auditDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
1362        // Set the original digest value to be 0.
1363        MemorySet(&session->u2.auditDigest.t.buffer,
1364                  0,
1365                  session->u2.auditDigest.t.size);
1366        return;
1367    }
```

6.4.5.4 UpdateAuditDigest

Function to update an audit digest

```
1368    static void
1369    UpdateAuditDigest(
1370        COMMAND         *command,
1371        TPMI_ALG_HASH    hashAlg,
1372        TPM2B_DIGEST    *digest
1373        )
1374    {
```

```
1375        HASH_STATE       hashState;
1376        TPM2B_DIGEST    *cpHash = GetCpHash(command, hashAlg);
1377        TPM2B_DIGEST    *rpHash = ComputeRpHash(command, hashAlg);
1378    //
1379        pAssert(cpHash != NULL);
1380        // digestNew :=  hash (digestOld || cpHash || rpHash)
1381            // Start hash computation.
1382        digest->t.size = CryptHashStart(&hashState, hashAlg);
1383            // Add old digest.
1384        CryptDigestUpdate2B(&hashState, &digest->b);
1385            // Add cpHash
1386        CryptDigestUpdate2B(&hashState, &cpHash->b);
1387            // Add rpHash
1388        CryptDigestUpdate2B(&hashState, &rpHash->b);
1389            // Finalize the hash.
1390        CryptHashEnd2B(&hashState, &digest->b);
1391    }
```

6.4.5.5 Audit()

This function updates the audit digest in an audit session.

```
1392    static void
1393    Audit(
1394        COMMAND         *command,       // IN: primary control structure
1395        SESSION         *auditSession   // IN: loaded audit session
1396        )
1397    {
1398        UpdateAuditDigest(command, auditSession->authHashAlg,
1399                          &auditSession->u2.auditDigest);
1400        return;
1401    }
1402    #ifdef  TPM_CC_GetCommandAuditDigest
```

6.4.5.6 CommandAudit()

This function updates the command audit digest.

```
1403    static void
1404    CommandAudit(
1405        COMMAND         *command        // IN:
1406        )
1407    {
1408        // If the digest.size is one, it indicates the special case of changing
1409        // the audit hash algorithm. For this case, no audit is done on exit.
1410        // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1411        // force an update to the NV on exit so that the change in digest will
1412        // be recorded. So, it is safe to exit here without setting any flags
1413        // because the digest change will be written to NV when this code exits.
1414        if(gr.commandAuditDigest.t.size == 1)
1415        {
1416            gr.commandAuditDigest.t.size = 0;
1417            return;
1418        }
1419        // If the digest size is zero, need to start a new digest and increment
1420        // the audit counter.
1421        if(gr.commandAuditDigest.t.size == 0)
1422        {
1423            gr.commandAuditDigest.t.size = CryptHashGetDigestSize(gp.auditHashAlg);
1424            MemorySet(gr.commandAuditDigest.t.buffer,
1425                      0,
1426                      gr.commandAuditDigest.t.size);
1427            // Bump the counter and save its value to NV.
```

```
1428            gp.auditCounter++;
1429            NV_SYNC_PERSISTENT(auditCounter);
1430        }
1431        UpdateAuditDigest(command, gp.auditHashAlg, &gr.commandAuditDigest);
1432        return;
1433    }
1434    #endif
```

6.4.5.7 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.

```
1435    static void
1436    UpdateAuditSessionStatus(
1437        COMMAND         *command        // IN: primary control structure
1438        )
1439    {
1440        UINT32           i;
1441        TPM_HANDLE       auditSession = TPM_RH_UNASSIGNED;
1442        // Iterate through sessions
1443        for(i = 0; i < command->sessionNum; i++)
1444        {
1445            SESSION     *session;
1446            // PW session do not have a loaded session and can not be an audit
1447            // session either.  Skip it.
1448            if(s_sessionHandles[i] == TPM_RS_PW)
1449                continue;
1450            session = SessionGet(s_sessionHandles[i]);
1451            // If a session is used for audit
1452            if(s_attributes[i].audit == SET)
1453            {
1454                // An audit session has been found
1455                auditSession = s_sessionHandles[i];
1456                // If the session has not been an audit session yet, or
1457                // the auditSetting bits indicate a reset, initialize it and set
1458                // it to be the exclusive session
1459                if(session->attributes.isAudit == CLEAR
1460                   || s_attributes[i].auditReset == SET)
1461                {
1462                    InitAuditSession(session);
1463                    g_exclusiveAuditSession = auditSession;
1464                }
1465                else
1466                {
1467                    // Check if the audit session is the current exclusive audit
1468                    // session and, if not, clear previous exclusive audit session.
1469                    if(g_exclusiveAuditSession != auditSession)
1470                        g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1471                }
1472                // Report audit session exclusivity.
1473                if(g_exclusiveAuditSession == auditSession)
1474                {
1475                    s_attributes[i].auditExclusive = SET;
1476                }
1477                else
1478                {
```

```
1479                s_attributes[i].auditExclusive = CLEAR;
1480            }
1481            // Extend audit log.
1482            Audit(command, session);
1483        }
1484    }
1485    // If no audit session is found in the command, and the command allows
1486    // a session then, clear the current exclusive
1487    // audit session.
1488    if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(command->index))
1489    {
1490        g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1491    }
1492    return;
1493 }
```

6.4.5.8 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```
1494 static void
1495 ComputeResponseHMAC(
1496     COMMAND         *command,       // IN: command structure
1497     UINT32           sessionIndex,  // IN: session index to be processed
1498     SESSION         *session,       // IN: loaded session
1499     TPM2B_DIGEST    *hmac           // OUT: authHMAC
1500     )
1501 {
1502     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1503     TPM2B_KEY        key;       // HMAC key
1504     BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
1505     BYTE            *buffer;
1506     UINT32           marshalSize;
1507     HMAC_STATE       hmacState;
1508     TPM2B_DIGEST    *rpHash = ComputeRpHash(command, session->authHashAlg);
1509     // Generate HMAC key
1510     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1511     // Add the object authValue if required
1512     if(session->attributes.includeAuth == SET)
1513     {
1514         // Note: includeAuth may be SET for a policy that is used in
1515         // UndefineSpaceSpecial(). At this point, the Index has been deleted
1516         // so the includeAuth will have no meaning. However, the
1517         // s_associatedHandles[] value for the session is now set to TPM_RH_NULL so
1518         // this will return the authValue associated with TPM_RH_NULL and that is
1519         // and empty buffer.
1520         // Get the authValue with trailing zeros removed
1521         TPM2B_AUTH          authValue;
1522         EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
1523         // Add it to the key
1524         MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
1525     }
1526     // if the HMAC key size is 0, the response HMAC is computed according to the
1527     // input HMAC
1528     if(key.t.size == 0
1529        && s_inputAuthValues[sessionIndex].t.size == 0)
1530     {
1531         hmac->t.size = 0;
1532         return;
1533     }
1534     // Start HMAC computation.
1535     hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
1536     // Add hash components.
1537     CryptDigestUpdate2B(&hmacState.hashState, &rpHash->b);
```

```
1538        CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
1539        CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
1540        // Add session attributes.
1541        buffer = marshalBuffer;
1542        marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1543        CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
1544        // Finalize HMAC.
1545        CryptHmacEnd2B(&hmacState, &hmac->b);
1546        return;
1547    }
```

6.4.5.9 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time.

b) Clears a policy session since nonce is rolling.

```
1548    static void
1549    UpdateInternalSession(
1550        SESSION         *session,       // IN: the session structure
1551        UINT32           i              // IN: session number
1552        )
1553    {
1554        // If nonce is rolling in a policy session, the policy related data
1555        // will be re-initialized.
1556        if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION
1557           && s_attributes[i].continueSession != CLEAR)
1558        {
1559            // When the nonce rolls it starts a new timing interval for the
1560            // policy session.
1561            SessionResetPolicyData(session);
1562            SessionSetStartTime(session);
1563        }
1564        return;
1565    }
```

6.4.5.10 BuildSingleResponseAuth()

Function to compute response HMAC value for a policy or HMAC session.

```
1566    static TPM2B_NONCE *
1567    BuildSingleResponseAuth(
1568        COMMAND         *command,       // IN: command structure
1569        UINT32           sessionIndex,  // IN: session index to be processed
1570        TPM2B_AUTH      *auth           // OUT: authHMAC
1571        )
1572    {
1573        // Fill in policy/HMAC based session response.
1574        SESSION     *session = SessionGet(s_sessionHandles[sessionIndex]);
1575        // If the session is a policy session with isPasswordNeeded SET, the
1576        // authorization field is empty.
1577        if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1578           && session->attributes.isPasswordNeeded == SET)
1579            auth->t.size = 0;
1580        else
1581            // Compute response HMAC.
1582            ComputeResponseHMAC(command, sessionIndex, session, auth);
1583        UpdateInternalSession(session, sessionIndex);
1584        return &session->nonceTPM;
1585    }
```

6.4.5.11 UpdateAllNonceTPM()

Updates TPM nonce for all sessions in command.

```
1586  static void
1587  UpdateAllNonceTPM(
1588      COMMAND         *command        // IN: controlling structure
1589      )
1590  {
1591      UINT32      i;
1592      SESSION     *session;
1593      for(i = 0; i < command->sessionNum; i++)
1594      {
1595          // If not a PW session, compute the new nonceTPM.
1596          if(s_sessionHandles[i] != TPM_RS_PW)
1597          {
1598              session = SessionGet(s_sessionHandles[i]);
1599              // Update nonceTPM in both internal session and response.
1600              CryptRandomGenerate(session->nonceTPM.t.size,
1601                                  session->nonceTPM.t.buffer);
1602          }
1603      }
1604      return;
1605  }
```

6.4.5.12 BuildResponseSession()

Function to build Session buffer in a response. The authorization data is added to the end of command->*responseBuffer*. The size of the authorization area is accumulated in command->*authSize*. When this is called, command->*responseBuffer* is pointing at the next location in the response buffer to be filled. This is where the authorization sessions will go, if any. command->*parameterSize* is the number of bytes that have been marshaled as parameters in the output buffer.

```
1606  void
1607  BuildResponseSession(
1608      COMMAND         *command        // IN: structure that has relevant command
1609                                      //     information
1610      )
1611  {
1612      pAssert(command->authSize == 0);
1613      // Reset the parameter buffer to point to the start of the parameters so that
1614      // there is a starting point for any rpHash that might be generated and so there
1615      // is a place where parameter encryption would start
1616      command->parameterBuffer = command->responseBuffer - command->parameterSize;
1617      // Session nonces should be updated before parameter encryption
1618      if(command->tag == TPM_ST_SESSIONS)
1619      {
1620          UpdateAllNonceTPM(command);
1621          // Encrypt first parameter if applicable. Parameter encryption should
1622          // happen after nonce update and before any rpHash is computed.
1623          // If the encrypt session is associated with a handle, the authValue of
1624          // this handle will be concatenated with sessionKey to generate
1625          // encryption key, no matter if the handle is the session bound entity
1626          // or not. The authValue is added to sessionKey only when the authValue
1627          // is available.
1628          if(s_encryptSessionIndex != UNDEFINED_INDEX)
1629          {
1630              UINT32          size;
1631              TPM2B_AUTH      extraKey;
1632              extraKey.b.size = 0;
1633              // If this is an authorization session, include the authValue in the
1634              // generation of the encryption key
1635              if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED)
```

```
1636                {
1637                    EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1638                                       &extraKey);
1639                }
1640                size = EncryptSize(command->index);
1641                CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1642                                         &s_nonceCaller[s_encryptSessionIndex].b,
1643                                         (UINT16)size,
1644                                         &extraKey,
1645                                         command->parameterBuffer);
1646            }
1647        }
1648        // Audit sessions should be processed regardless of the tag because
1649        // a command with no session may cause a change of the exclusivity state.
1650        UpdateAuditSessionStatus(command);
1651    #ifdef TPM_CC_GetCommandAuditDigest
1652        // Command Audit
1653        if(CommandAuditIsRequired(command->index))
1654            CommandAudit(command);
1655    #endif
1656        // Process command with sessions.
1657        if(command->tag == TPM_ST_SESSIONS)
1658        {
1659            UINT32          i;
1660            pAssert(command->sessionNum > 0);
1661            // Iterate over each session in the command session area, and create
1662            // corresponding sessions for response.
1663            for(i = 0; i < command->sessionNum; i++)
1664            {
1665                TPM2B_NONCE     *nonceTPM;
1666                TPM2B_DIGEST     responseAuth;
1667                // Make sure that continueSession is SET on any Password session.
1668                // This makes it marginally easier for the management software
1669                // to keep track of the closed sessions.
1670                if(s_sessionHandles[i] == TPM_RS_PW)
1671                {
1672                    s_attributes[i].continueSession = SET;
1673                    responseAuth.t.size = 0;
1674                    nonceTPM = (TPM2B_NONCE *)&responseAuth;
1675                }
1676                else
1677                {
1678                    // Compute the response HMAC and get a pointer to the nonce used.
1679                    // This function will also update the values if needed. Note, the
1680                    nonceTPM = BuildSingleResponseAuth(command, i, &responseAuth);
1681                }
1682                command->authSize += TPM2B_NONCE_Marshal(nonceTPM,
1683                                                         &command->responseBuffer,
1684                                                         NULL);
1685                command->authSize += TPMA_SESSION_Marshal(&s_attributes[i],
1686                                                          &command->responseBuffer,
1687                                                          NULL);
1688                command->authSize += TPM2B_DIGEST_Marshal(&responseAuth,
1689                                                          &command->responseBuffer,
1690                                                          NULL);
1691                if(s_attributes[i].continueSession == CLEAR)
1692                    SessionFlush(s_sessionHandles[i]);
1693            }
1694        }
1695        return;
1696    }
```


6.4.5.13 SessionRemoveAssociationToHandle()

This function deals with the case where an entity associated with an authorization is deleted during command processing. The primary use of this is to support UndefineSpaceSpecial().

```
1697    void
1698    SessionRemoveAssociationToHandle(
1699        TPM_HANDLE       handle
1700        )
1701    {
1702        UINT32               i;
1703        for(i = 0; i < MAX_SESSION_NUM; i++)
1704        {
1705            if(s_associatedHandles[i] == handle)
1706            {
1707                s_associatedHandles[i] = TPM_RH_NULL;
1708            }
1709        }
1710    }
```


7 Command Support Functions

7.1 Introduction

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

7.2 Attestation Command Support (Attest_spt.c)

7.2.1 Includes

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
```

7.2.2 Functions

7.2.2.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

```
void
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
    )
{
    OBJECT              *signObject = HandleToObject(signHandle);
        // Magic number
    attest->magic = TPM_GENERATED_VALUE;
    if(signObject == NULL)
    {
        // The name for a null handle is TPM_RH_NULL
        // This is defined because UINT32_TO_BYTE_ARRAY does a cast. If the
        // size of the cast is smaller than a constant, the compiler warns
        // about the truncation of a constant value.
        TPM_HANDLE      nullHandle = TPM_RH_NULL;
        attest->qualifiedSigner.t.size = sizeof(TPM_HANDLE);
        UINT32_TO_BYTE_ARRAY(nullHandle, attest->qualifiedSigner.t.name);
    }
    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
            attest->qualifiedSigner = signObject->qualifiedName;
    }
    // current clock in plain text
    TimeFillInfo(&attest->clockInfo);
    // Firmware version in plain text
    attest->firmwareVersion = ((UINT64)gp.firmwareV1 << (sizeof(UINT32) * 8));
    attest->firmwareVersion += gp.firmwareV2;
    // Check the hierarchy of sign object.  For NULL sign handle, the hierarchy
    // will be TPM_RH_NULL
    if((signObject == NULL)
       || (!signObject->attributes.epsHierarchy
```

```
            && !signObject->attributes.ppsHierarchy))
    {
        // For signing key that is not in platform or endorsement hierarchy,
        // obfuscate the reset, restart and firmware version information
        UINT64          obfuscation[2];
        CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gp.shProof.b, OBFUSCATE_STRING,
                  &attest->qualifiedSigner.b, NULL, 128,
                  (BYTE *)&obfuscation[0], NULL, FALSE);
        // Obfuscate data
        attest->firmwareVersion += obfuscation[0];
        attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
        attest->clockInfo.restartCount += (UINT32)obfuscation[1];
    }
    // External data
    if(CryptIsSchemeAnonymous(scheme->scheme))
        attest->extraData.t.size = 0;
    else
    {
        // If we move the data to the attestation structure, then it is not
        // used in the signing operation except as part of the signed data
        attest->extraData = *data;
        data->t.size = 0;
    }
}
```

7.2.2.2 SignAttestInfo()

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

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

```
TPM_RC
SignAttestInfo(
    OBJECT              *signKey,           // IN: 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
    )
{
    BYTE                    *buffer;
    HASH_STATE              hashState;
    TPM2B_DIGEST            digest;
    TPM_RC                  result;
    // Marshal TPMS_ATTEST structure for hash
    buffer = attest->t.attestationData;
    attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);
    if(signKey == NULL)
    {
        signature->sigAlg = TPM_ALG_NULL;
        result = TPM_RC_SUCCESS;
    }
    else
    {
```

```
92          TPMI_ALG_HASH           hashAlg;
93          // Compute hash
94          hashAlg = scheme->details.any.hashAlg;
95          // need to set the receive buffer to get something put in it
96          digest.t.size = sizeof(digest.t.buffer);
97          digest.t.size = CryptHashBlock(hashAlg, attest->t.size,
98                                         attest->t.attestationData,
99                                         digest.t.size, digest.t.buffer);
100         // If there is qualifying data, need to rehash the data
101         // hash(qualifyingData || hash(attestationData))
102         if(qualifyingData->t.size != 0)
103         {
104             CryptHashStart(&hashState, hashAlg);
105             CryptDigestUpdate2B(&hashState, &qualifyingData->b);
106             CryptDigestUpdate2B(&hashState, &digest.b);
107             CryptHashEnd2B(&hashState, &digest.b);
108         }
109         // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
110         // TPM_RC_ATTRIBUTES error may be returned at this point
111         result = CryptSign(signKey, scheme, &digest, signature);
112         // Since the clock is used in an attestation, the state in NV is no longer
113         // "orderly" with respect to the data in RAM if the signature is valid
114         if(result == TPM_RC_SUCCESS)
115         {
116             // Command uses the clock so need to clear the orderly state if it is
117             // set.
118             result = NvClearOrderly();
119         }
120     }
121     return result;
122 }
```

7.2.2.3 IsSigningObject()

Checks to see if the object is OK for signing. This is here rather than in Object_spt.c because all the attestation commands use this file but not Object_spt.c.

Return Value	Meaning
TRUE	object may sign
FALSE	object may not sign

```
123 BOOL
124 IsSigningObject(
125     OBJECT          *object         // IN:
126     )
127 {
128     return ((object == NULL) || ((object->publicArea.objectAttributes.sign == SET)
129                                  && object->publicArea.type != TPM_ALG_SYMCIPHER));
130 }
```


7.3 Context Management Command Support (Context_spt.c)

7.3.1 Includes

```
#include "Tpm.h"
#include "Context_spt_fp.h"
```

7.3.2 Functions

7.3.2.1 ComputeContextProtectionKey()

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

```
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);
    cAssert(sizeof(contextBlob->sequence) <= sizeof(sequence2B.t.buffer));
    MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence,
               sizeof(contextBlob->sequence));
    // Get handle value in 2B format
    handle2B.t.size = sizeof(contextBlob->savedHandle);
    cAssert(sizeof(contextBlob->savedHandle) <= sizeof(handle2B.t.buffer));
    MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle,
               sizeof(contextBlob->savedHandle));
    // Get the symmetric encryption key size
    symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
    symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
    // Get the size of the IV for the algorithm
    iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);
    // KDFa to generate symmetric key and IV value
    CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, CONTEXT_KEY, &sequence2B.b,
              &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL,
              FALSE);
         // Copy part of the returned value as the key
    pAssert(symKey->t.size <= sizeof(symKey->t.buffer));
    MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size);
    // Copy the rest as the IV
    pAssert(iv->t.size <= sizeof(iv->t.buffer));
    MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
    return;
}
```


7.3.2.2 ComputeContextIntegrity()

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

```
46  void
47  ComputeContextIntegrity(
48      TPMS_CONTEXT    *contextBlob,   // IN: context blob
49      TPM2B_DIGEST    *integrity      // OUT: integrity
50      )
51  {
52      HMAC_STATE          hmacState;
53      TPM2B_AUTH          *proof;
54      UINT16              integritySize;
55      // Get proof value
56      proof = HierarchyGetProof(contextBlob->hierarchy);
57      // Start HMAC
58      integrity->t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
59                                           &proof->b);
60      // Compute integrity size at the beginning of context blob
61      integritySize = sizeof(integrity->t.size) + integrity->t.size;
62      // Adding total reset counter so that the context cannot be
63      // used after a TPM Reset
64      CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
65                           gp.totalResetCount);
66      // If this is a ST_CLEAR object, add the clear count
67      // so that this contest cannot be loaded after a TPM Restart
68      if(contextBlob->savedHandle == 0x80000002)
69          CryptDigestUpdateInt(&hmacState.hashState, sizeof(gr.clearCount),
70                               gr.clearCount);
71      // Adding sequence number to the HMAC to make sure that it doesn't
72      // get changed
73      CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->sequence),
74                           contextBlob->sequence);
75      // Protect the handle
76      CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->savedHandle),
77                           contextBlob->savedHandle);
78      // Adding sensitive contextData, skip the leading integrity area
79      CryptDigestUpdate(&hmacState.hashState,
80                        contextBlob->contextBlob.t.size - integritySize,
81                        contextBlob->contextBlob.t.buffer + integritySize);
82      // Complete HMAC
83      CryptHmacEnd2B(&hmacState, &integrity->b);
84      return;
85  }
```

7.3.2.3 SequenceDataExport()

This function is used scan through the sequence object and either modify the hash state data for export *(contextSave*) or to import it into the internal format *(contextLoad*). This function should only be called after the sequence object has been copied to the context buffer *(contextSave*) or from the context buffer into the sequence object. The presumption is that the context buffer version of the data is the same size as the internal representation so nothing outsize of the hash context area gets modified.

```
86  void
87  SequenceDataExport(
88      HASH_OBJECT         *object,        // IN: an internal hash object
89      HASH_OBJECT_BUFFER  *exportObject   // OUT: a sequence context in a buffer
90      )
91  {
92      // If the hash object is not an event, then only one hash context is needed
93      int                  count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
94      for(count--; count >= 0; count--)
```

```
 95      {
 96          HASH_STATE          *hash = &object->state.hashState[count];
 97          size_t               offset = (BYTE *)hash - (BYTE *)object;
 98          BYTE                *exportHash = &((BYTE *)exportObject)[offset];
 99          CryptHashExportState(hash, (EXPORT_HASH_STATE *)exportHash);
100      }
101  }
```

7.3.2.4 SequenceDataImport()

This function is used scan through the sequence object and either modify the hash state data for export *(contextSave*) or to import it into the internal format *(contextLoad*). This function should only be called after the sequence object has been copied to the context buffer *(contextSave*) or from the context buffer into the sequence object. The presumption is that the context buffer version of the data is the same size as the internal representation so nothing outsize of the hash context area gets modified.

```
102  void
103  SequenceDataImport(
104      HASH_OBJECT         *object,        // IN/OUT: an internal hash object
105      HASH_OBJECT_BUFFER  *exportObject   // IN/OUT: a sequence context in a buffer
106      )
107  {
108      // If the hash object is not an event, then only one hash context is needed
109      int                  count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
110      for(count--; count >= 0; count--)
111      {
112          HASH_STATE          *hash = &object->state.hashState[count];
113          size_t               offset = (BYTE *)hash - (BYTE *)object;
114          BYTE                *importHash = &((BYTE *)exportObject)[offset];
115  //
116          CryptHashImportState(hash, (EXPORT_HASH_STATE *)importHash);
117      }
118  }
```

7.4 Policy Command Support (Policy_spt.c)

```
#include "Tpm.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           blameNonce,
    TPM_RC           blameCpHash,
    TPM_RC           blameExpiration
    )
{
    // Validate that input nonceTPM is correct if present
    if(nonce != NULL && nonce->t.size != 0)
    {
        if(!MemoryEqual2B(&nonce->b, &session->nonceTPM.b))
            return TPM_RCS_NONCE + blameNonce;
    }
    // If authTimeout is set (expiration != 0...
    if(authTimeout != 0)
    {
        // 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.
        RETURN_IF_NV_IS_NOT_AVAILABLE;
        // if the time has already passed or the time epoch has changed then the
        // time value is no longer good.
        if((authTimeout < g_time)
           || (session->epoch != g_timeEpoch))
            return TPM_RCS_EXPIRED + blameExpiration;
    }
    // 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_RCS_SIZE + blameCpHash;
        // If the cpHash has already been set, then this input value
        // must match the current value.
        if(session->u1.cpHash.b.size != 0
           && !MemoryEqual2B(&cpHashA->b, &session->u1.cpHash.b))
            return TPM_RC_CPHASH;
    }
    return TPM_RC_SUCCESS;
}
```

7.4.2 PolicyContextUpdate()

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


```
50  void
51  PolicyContextUpdate(
52      TPM_CC           commandCode,   // IN: command code
53      TPM2B_NAME      *name,          // IN: name of entity
54      TPM2B_NONCE     *ref,           // IN: the reference data
55      TPM2B_DIGEST    *cpHash,        // IN: the cpHash (optional)
56      UINT64           policyTimeout, // IN: the timeout value for the policy
57      SESSION         *session        // IN/OUT: policy session to be updated
58      )
59  {
60      HASH_STATE           hashState;
61      // Start hash
62     CryptHashStart(&hashState, session->authHashAlg);
63      // policyDigest size should always be the digest size of session hash algorithm.
64      pAssert(session->u2.policyDigest.t.size
65              == CryptHashGetDigestSize(session->authHashAlg));
66      // add old digest
67      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
68      // add commandCode
69      CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
70      // add name if applicable
71      if(name != NULL)
72          CryptDigestUpdate2B(&hashState, &name->b);
73      // Complete the digest and get the results
74      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
75      // If the policy reference is not null, do a second update to the digest.
76      if(ref != NULL)
77      {
78          // Start second hash computation
79          CryptHashStart(&hashState, session->authHashAlg);
80          // add policyDigest
81          CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
82          // add policyRef
83          CryptDigestUpdate2B(&hashState, &ref->b);
84          // Complete second digest
85          CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
86      }
87      // Deal with the cpHash. If the cpHash value is present
88      // then it would have already been checked to make sure that
89      // it is compatible with the current value so all we need
90      // to do here is copy it and set the isCpHashDefined attribute
91      if(cpHash != NULL && cpHash->t.size != 0)
92      {
93          session->u1.cpHash = *cpHash;
94          session->attributes.isCpHashDefined = SET;
95      }
96      // update the timeout if it is specified
97      if(policyTimeout != 0)
98      {
99          // If the timeout has not been set, then set it to the new value
100         // than the current timeout then set it to the new value
101         if(session->timeout == 0 || session->timeout > policyTimeout)
102             session->timeout = policyTimeout;
103     }
104     return;
105 }
```

7.4.2.1 ComputeAuthTimeout()

This function is used to determine what the authorization timeout value for the session should be.

```
106 UINT64
107 ComputeAuthTimeout(
108     SESSION         *session,              // IN: the session containing the time
```

```
                                                //     values
    INT32            expiration,                // IN: either the number of seconds from
                                                //     the start of the session or the
                                                //     time in g_timer;
    TPM2B_NONCE     *nonce                      // IN: indicator of the time base
    )
{
    UINT64           policyTime;
    // If no expiration, policy time is 0
    if(expiration == 0)
        policyTime = 0;
    else
    {
        if(expiration < 0)
            expiration = -expiration;
        if(nonce->t.size == 0)
            // The input time is absolute Time (not Clock), but it is expressed
            // in seconds. To make sure that we don't time out too early, take the
            // current value of milliseconds in g_time and add that to the input
            // seconds value.
            policyTime = (((UINT64)expiration) * 1000) + g_time % 1000;
        else
            // The policy timeout is the absolute value of the expiration in seconds
            // added to the start time of the policy.
            policyTime = session->startTime + (((UINT64)expiration) * 1000);
    }
    return policyTime;
}
```

7.4.2.2 PolicyDigestClear()

Function to reset the *policyDigest* of a session

```
void
PolicyDigestClear(
    SESSION         *session
    )
{
    session->u2.policyDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
    MemorySet(session->u2.policyDigest.t.buffer, 0,
              session->u2.policyDigest.t.size);
}
BOOL
PolicySptCheckCondition(
    TPM_EO          operation,
    BYTE            *opA,
    BYTE            *opB,
    UINT16           size
    )
{
    // Arithmetic Comparison
    switch(operation)
    {
        case TPM_EO_EQ:
            // compare A = B
            return (UnsignedCompareB(size, opA, size, opB) == 0);
            break;
        case TPM_EO_NEQ:
            // compare A != B
            return (UnsignedCompareB(size, opA, size, opB) != 0);
            break;
        case TPM_EO_SIGNED_GT:
            // compare A > B signed
            return (SignedCompareB(size, opA, size, opB) > 0);
```

```
168             break;
169         case TPM_EO_UNSIGNED_GT:
170             // compare A > B unsigned
171             return (UnsignedCompareB(size, opA, size, opB) > 0);
172             break;
173         case TPM_EO_SIGNED_LT:
174             // compare A < B signed
175             return (SignedCompareB(size, opA, size, opB) < 0);
176             break;
177         case TPM_EO_UNSIGNED_LT:
178             // compare A < B unsigned
179             return (UnsignedCompareB(size, opA, size, opB) < 0);
180             break;
181         case TPM_EO_SIGNED_GE:
182             // compare A >= B signed
183             return (SignedCompareB(size, opA, size, opB) >= 0);
184             break;
185         case TPM_EO_UNSIGNED_GE:
186             // compare A >= B unsigned
187             return (UnsignedCompareB(size, opA, size, opB) >= 0);
188             break;
189         case TPM_EO_SIGNED_LE:
190             // compare A <= B signed
191             return (SignedCompareB(size, opA, size, opB) <= 0);
192             break;
193         case TPM_EO_UNSIGNED_LE:
194             // compare A <= B unsigned
195             return (UnsignedCompareB(size, opA, size, opB) <= 0);
196             break;
197         case TPM_EO_BITSET:
198             // All bits SET in B are SET in A. ((A&B)=B)
199         {
200             UINT32 i;
201             for(i = 0; i < size; i++)
202                 if((opA[i] & opB[i]) != opB[i])
203                     return FALSE;
204         }
205         break;
206         case TPM_EO_BITCLEAR:
207             // All bits SET in B are CLEAR in A. ((A&B)=0)
208         {
209             UINT32 i;
210             for(i = 0; i < size; i++)
211                 if((opA[i] & opB[i]) != 0)
212                     return FALSE;
213         }
214         break;
215         default:
216             FAIL(FATAL_ERROR_INTERNAL);
217             break;
218     }
219     return TRUE;
220 }
```


7.5 NV Command Support (NV_spt.c)

7.5.1 Includes

```
#include "Tpm.h"
#include "NV_spt_fp.h"
```

7.5.2 Functions

7.5.2.1 NvReadAccessChecks()

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

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	*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 read
    TPMA_NV          attributes     // IN: the attributes of 'nvHandle'
    )
{
    // If data is read locked, returns an error
   if(IsNv_TPMA_NV_READLOCKED(attributes))
       return TPM_RC_NV_LOCKED;
    // If the authorization was provided by the owner or platform, then check
    // that the attributes allow the read.  If the authorization handle
    // is the same as the index, then the checks were made when the authorization
    // was checked..
    if(authHandle == TPM_RH_OWNER)
    {
        // If Owner provided authorization then ONWERWRITE must be SET
        if(!IsNv_TPMA_NV_OWNERREAD(attributes))
            return TPM_RC_NV_AUTHORIZATION;
    }
    else if(authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided authorization then PPWRITE must be SET
        if(!IsNv_TPMA_NV_PPREAD(attributes))
            return TPM_RC_NV_AUTHORIZATION;
    }
    // If neither Owner nor Platform provided authorization, make sure that it was
    // provided by this index.
    else if(authHandle != nvHandle)
        return TPM_RC_NV_AUTHORIZATION;
// If the index has not been written, then the value cannot be read
// NOTE: This has to come after other access checks to make sure that
// the proper authorization is given to TPM2_NV_ReadLock()
    if(!IsNv_TPMA_NV_WRITTEN(attributes))
        return TPM_RC_NV_UNINITIALIZED;
    return TPM_RC_SUCCESS;
}
```


7.5.2.2 NvWriteAccessChecks()

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

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	Authorization fails
TPM_RC_NV_LOCKED	Write locked

```
41  TPM_RC
42  NvWriteAccessChecks(
43      TPM_HANDLE       authHandle,    // IN: the handle that provided the
44                                      //     authorization
45      TPM_HANDLE       nvHandle,      // IN: the handle of the NV index to be written
46      TPMA_NV          attributes     // IN: the attributes of 'nvHandle'
47      )
48  {
49      // If data is write locked, returns an error
50      if(IsNv_TPMA_NV_WRITELOCKED(attributes))
51          return TPM_RC_NV_LOCKED;
52      // If the authorization was provided by the owner or platform, then check
53      // that the attributes allow the write.  If the authorization handle
54      // is the same as the index, then the checks were made when the authorization
55      // was checked..
56      if(authHandle == TPM_RH_OWNER)
57      {
58          // If Owner provided authorization then ONWERWRITE must be SET
59          if(!IsNv_TPMA_NV_OWNERWRITE(attributes))
60              return TPM_RC_NV_AUTHORIZATION;
61      }
62      else if(authHandle == TPM_RH_PLATFORM)
63      {
64          // If Platform provided authorization then PPWRITE must be SET
65          if(!IsNv_TPMA_NV_PPWRITE(attributes))
66              return TPM_RC_NV_AUTHORIZATION;
67      }
68      // If neither Owner nor Platform provided authorization, make sure that it was
69      // provided by this index.
70      else if(authHandle != nvHandle)
71          return TPM_RC_NV_AUTHORIZATION;
72      return TPM_RC_SUCCESS;
73  }
```

7.5.2.3 NvClearOrderly()

This function is used to cause gp.*orderlyState* to be cleared to the non-orderly state.

```
74  TPM_RC
75  NvClearOrderly(
76      void
77      )
78  {
79      if(gp.orderlyState < SU_DA_USED_VALUE)
80          RETURN_IF_NV_IS_NOT_AVAILABLE;
81      g_clearOrderly = TRUE;
82      return TPM_RC_SUCCESS;
83  }
```


7.5.2.4 NvIsPinPassIndex()

Function to check to see if an NV index is a PIN Pass Index

Return Value	Meaning
TRUE	is pin pass
FALSE	is not pin pass

```
84  BOOL
85  NvIsPinPassIndex(
86      TPM_HANDLE          index       // IN: Handle to check
87      )
88  {
89      if(HandleGetType(index) == TPM_HT_NV_INDEX)
90      {
91          NV_INDEX                *nvIndex = NvGetIndexInfo(index, NULL);
92          return IsNvPinPassIndex(nvIndex->publicArea.attributes);
93      }
94      return FALSE;
95  }
```

7.6 Object Command Support (Object_spt.c)

7.6.1 Includes

```
#include "Tpm.h"
#include "Object_spt_fp.h"
```

7.6.2 Local Functions

7.6.2.1 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

```
static UINT16
GetIV2BSize(
    OBJECT              *protector          // IN: the protector handle
    )
{
    TPM_ALG_ID          symAlg;
    UINT16              keyBits;
    // Determine the symmetric algorithm and size of key
    if(protector == NULL)
    {
        // Use the context encryption algorithm and key size
        symAlg = CONTEXT_ENCRYPT_ALG;
        keyBits = CONTEXT_ENCRYPT_KEY_BITS;
    }
    else
    {
        symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
        keyBits = protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
    }
    // The IV size is a UINT16 size field plus the block size of the symmetric
    // algorithm
    return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
}
```

7.6.2.2 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

```
static void
ComputeProtectionKeyParms(
    OBJECT          *protector,         // IN: the protector object
    TPM_ALG_ID       hashAlg,           // IN: hash algorithm for KDFa
    TPM2B           *name,              // IN: name of the object
    TPM2B           *seedIn,            // IN: optional seed for duplication blob.
                                        //     For non duplication blob, this
                                        //     parameter should be NULL
    TPM_ALG_ID      *symAlg,            // OUT: the symmetric algorithm
    UINT16          *keyBits,           // OUT: the symmetric key size in bits
    TPM2B_SYM_KEY   *symKey             // OUT: the symmetric key
    )
```



```
38  {
39      const TPM2B         *seed = seedIn;
40      // Determine the algorithms for the KDF and the encryption/decryption
41      // For TPM_RH_NULL, using context settings
42      if(protector == NULL)
43      {
44          // Use the context encryption algorithm and key size
45          *symAlg = CONTEXT_ENCRYPT_ALG;
46          symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
47          *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
48      }
49      else
50      {
51          TPMT_SYM_DEF_OBJECT *symDef;
52          symDef = &protector->publicArea.parameters.asymDetail.symmetric;
53          *symAlg = symDef->algorithm;
54          *keyBits = symDef->keyBits.sym;
55          symKey->t.size = (*keyBits + 7) / 8;
56      }
57      // Get seed for KDF
58      if(seed == NULL)
59          seed = GetSeedForKDF(protector);
60      // KDFa to generate symmetric key and IV value
61      CryptKDFa(hashAlg, seed, STORAGE_KEY, name, NULL,
62                symKey->t.size * 8, symKey->t.buffer, NULL, FALSE);
63      return;
64  }
```

7.6.2.3 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.

```
65  static void
66  ComputeOuterIntegrity(
67      TPM2B           *name,              // IN: the name of the object
68      OBJECT          *protector,         // IN: the object that
69                                          //     provides protection. For an object,
70                                          //     it is a parent. For a credential, it
71                                          //     is the encrypt object. For
72                                          //     a Temporary Object, it is NULL
73      TPMI_ALG_HASH    hashAlg,           // IN: algorithm to use for integrity
74      TPM2B           *seedIn,            // IN: an external seed may be provided for
75                                          //     duplication blob. For non duplication
76                                          //     blob, this parameter should be NULL
77      UINT32           sensitiveSize,     // IN: size of the marshaled sensitive data
78      BYTE            *sensitiveData,     // IN: sensitive area
79      TPM2B_DIGEST    *integrity          // OUT: integrity
80      )
81  {
82      HMAC_STATE       hmacState;
83      TPM2B_DIGEST     hmacKey;
84      const TPM2B     *seed = seedIn;
85  //
86      // Get seed for KDF
87      if(seed == NULL)
88          seed = GetSeedForKDF(protector);
89      // Determine the HMAC key bits
90      hmacKey.t.size = CryptHashGetDigestSize(hashAlg);
91      // KDFa to generate HMAC key
92      CryptKDFa(hashAlg, seed, INTEGRITY_KEY, NULL, NULL,
93                hmacKey.t.size * 8, hmacKey.t.buffer, NULL, FALSE);
```

```
    // Start HMAC and get the size of the digest which will become the integrity
    integrity->t.size = CryptHmacStart2B(&hmacState, hashAlg, &hmacKey.b);
    // Adding the marshaled sensitive area to the integrity value
    CryptDigestUpdate(&hmacState.hashState, sensitiveSize, sensitiveData);
    // Adding name
    CryptDigestUpdate2B(&hmacState.hashState, name);
    // Compute HMAC
    CryptHmacEnd2B(&hmacState, &integrity->b);
    return;
}
```

7.6.2.4 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap

```
static void
ComputeInnerIntegrity(
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
    TPM2B           *name,          // IN: the name of the object
    UINT16           dataSize,      // IN: the size of sensitive data
    BYTE            *sensitiveData, // IN: sensitive data
    TPM2B_DIGEST    *integrity      // OUT: inner integrity
    )
{
    HASH_STATE      hashState;
    // Start hash and get the size of the digest which will become the integrity
    integrity->t.size = CryptHashStart(&hashState, hashAlg);
    // Adding the marshaled sensitive area to the integrity value
    CryptDigestUpdate(&hashState, dataSize, sensitiveData);
    // Adding name
    CryptDigestUpdate2B(&hashState, name);
    // Compute hash
    CryptHashEnd2B(&hashState, &integrity->b);
    return;
}
```

7.6.2.5 ProduceInnerIntegrity()

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

```
static UINT16
ProduceInnerIntegrity(
    TPM2B           *name,          // IN: the name of the object
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
    UINT16           dataSize,      // IN: the size of sensitive data, excluding the
                                    //     leading integrity buffer size
    BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
                                    //     it.  At input, the leading bytes of this
                                    //     buffer is reserved for integrity
    )
{
    BYTE            *sensitiveData; // pointer to the sensitive data
    TPM2B_DIGEST     integrity;
    UINT16           integritySize;
    BYTE            *buffer;        // Auxiliary buffer pointer
    // sensitiveData points to the beginning of sensitive data in innerBuffer
    integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
    sensitiveData = innerBuffer + integritySize;
    ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);
```

```
143         // Add integrity at the beginning of inner buffer
144         buffer = innerBuffer;
145         TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
146         return dataSize + integritySize;
147     }
```

7.6.2.6 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

```
148     static TPM_RC
149     CheckInnerIntegrity(
150         TPM2B           *name,          // IN: the name of the object
151         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
152         UINT16           dataSize,      // IN: the size of sensitive data, including the
153                                         //     leading integrity buffer size
154         BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
155                                         //     it
156         )
157     {
158         TPM_RC          result;
159         TPM2B_DIGEST    integrity;
160         TPM2B_DIGEST    integrityToCompare;
161         BYTE            *buffer;                // Auxiliary buffer pointer
162         INT32           size;
163         // Unmarshal integrity
164         buffer = innerBuffer;
165         size = (INT32)dataSize;
166         result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
167         if(result == TPM_RC_SUCCESS)
168         {
169             // Compute integrity to compare
170             ComputeInnerIntegrity(hashAlg, name, (UINT16)size, buffer,
171                                   &integrityToCompare);
172             // Compare outer blob integrity
173             if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
174                 result = TPM_RC_INTEGRITY;
175         }
176         return result;
177     }
```

7.6.3 Public Functions

7.6.3.1 AdjustAuthSize()

This function will validate that the input *authValue* is no larger than the *digestSize* for the *nameAlg*. It will then pad with zeros to the size of the digest.

```
178     BOOL
179     AdjustAuthSize(
180         TPM2B_AUTH          *auth,          // IN/OUT: value to adjust
181         TPMI_ALG_HASH        nameAlg        // IN:
182         )
183     {
184         UINT16               digestSize;
185         // If there is no nameAlg, then this is a LoadExternal and the authVale can
```

```
186         // be any size up to the maximum allowed by the
187         digestSize = (nameAlg == TPM_ALG_NULL) ? sizeof(TPMU_HA)
188             : CryptHashGetDigestSize(nameAlg);
189         if(digestSize < MemoryRemoveTrailingZeros(auth))
190             return FALSE;
191         else if(digestSize > auth->t.size)
192             MemoryPad2B(&auth->b, digestSize);
193         auth->t.size = digestSize;
194         return TRUE;
195     }
```

7.6.3.2 AreAttributesForParent()

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

NOTE: The *isParent* attribute is SET when an object is loaded and it has attributes that are suitable for a parent object.

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

```
196     BOOL
197     ObjectIsParent(
198         OBJECT          *parentObject   // IN: parent handle
199         )
200     {
201         return parentObject->attributes.isParent;
202     }
```

7.6.3.3 CreateChecks()

Attribute checks that are unique to creation.

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*sensitiveDataOrigin* is not consistent with the object type
other	returns from PublicAttributesValidation()

```
203     TPM_RC
204     CreateChecks(
205         OBJECT              *parentObject,
206         TPMT_PUBLIC         *publicArea
207         )
208     {
209         TPMA_OBJECT          attributes = publicArea->objectAttributes;
210         TPM_RC               result = TPM_RC_SUCCESS;
211         switch(publicArea->type)
212         {
213             case TPM_ALG_SYMCIPHER:
214                 // A restricted key must have sensitiveDataOrigin SET unless it has
215                 // fixedParent and fixedTPM CLEAR.
216                 if(attributes.restricted)
217                     if(!attributes.sensitiveDataOrigin)
218                         if(attributes.fixedParent || attributes.fixedTPM)
219                             result = TPM_RCS_ATTRIBUTES;
220                 break;
221             case TPM_ALG_KEYEDHASH:
222                 // if this is a data object (sign == decrypt == CLEAR) then the
223                 // TPM cannot be the data source.
```

```
            if(!attributes.sign && !attributes.decrypt
               && attributes.sensitiveDataOrigin)
            result = TPM_RC_ATTRIBUTES;
            break;
        default: // Asymmetric keys cannot have the sensitive portion provided
            if(!attributes.sensitiveDataOrigin)
                result = TPM_RCS_ATTRIBUTES;
            break;
    }
    if(TPM_RC_SUCCESS == result)
    {
        result = PublicAttributesValidation(parentObject, publicArea);
    }
    return result;
}
```

7.6.3.4 SchemeChecks

This function is called by TPM2_LoadExternal() and PublicAttributesValidation().

Return Value	Meaning This function TPM_RC
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public parameters
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_RCS_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
SchemeChecks(
    OBJECT          *parentObject,  // IN: parent (null if primary seed)
    TPMT_PUBLIC     *publicArea     // IN: public area of the object
    )
{
    TPMT_SYM_DEF_OBJECT     *symAlgs = NULL;
    TPM_ALG_ID               scheme = TPM_ALG_NULL;
    TPMA_OBJECT              attributes = publicArea->objectAttributes;
    TPMU_PUBLIC_PARMS        *parms = &publicArea->parameters;
    switch(publicArea->type)
    {
        case TPM_ALG_SYMCIPHER:
            symAlgs = &parms->symDetail.sym;
            break;
        case TPM_ALG_KEYEDHASH:
            scheme = parms->keyedHashDetail.scheme.scheme;
            // if both sign and decrypt
            if(attributes.sign == attributes.decrypt)
            {
                // if both sign and decrypt are set or clear, then need
                // TPM_ALG_NULL as scheme
                if(scheme != TPM_ALG_NULL)
                    return TPM_RCS_SCHEME;
            }
            else if(attributes.sign && scheme != TPM_ALG_HMAC)
                return TPM_RCS_SCHEME;
```

```
266             else if(attributes.decrypt)
267             {
268                 if(scheme != TPM_ALG_XOR)
269                     return TPM_RCS_SCHEME;
270                 // If this is a derivation parent, then the KDF needs to be
271                 // SP800-108 for this implementation. This is the only derivation
272                 // supported by this implementation. Other implementations could
273                 // support additional schemes. There is no default.
274                 if(attributes.restricted)
275                 {
276                     if(parms->keyedHashDetail.scheme.details.xor.kdf
277                        != TPM_ALG_KDF1_SP800_108)
278                         return TPM_RCS_SCHEME;
279                     // Must select a digest.
280                     if(CryptHashGetDigestSize(
281                         parms->keyedHashDetail.scheme.details.xor.hashAlg) == 0)
282                         return TPM_RCS_HASH;
283                 }
284             }
285             break;
286         default: // handling for asymmetric
287             scheme = parms->asymDetail.scheme.scheme;
288             symAlgs = &parms->asymDetail.symmetric;
289             // if the key is both sign and decrypt, then the scheme must be
290             // TPM_ALG_NULL because there is no way to specify both a sign and a
291             // decrypt scheme in the key.
292             if(attributes.sign == attributes.decrypt)
293             {
294                 // scheme must be TPM_ALG_NULL
295                 if(scheme != TPM_ALG_NULL)
296                     return TPM_RCS_SCHEME;
297             }
298             else if(attributes.sign)
299             {
300                 // If this is a signing key, see if it has a signing scheme
301                 if(CryptIsAsymSignScheme(publicArea->type, scheme))
302                 {
303                     // if proper signing scheme then it needs a proper hash
304                     if(parms->asymDetail.scheme.details.anySig.hashAlg
305                        == TPM_ALG_NULL)
306                         return TPM_RCS_SCHEME;
307                 }
308                 else
309                 {
310                     // signing key that does not have a proper signing scheme.
311                     // This is OK if the key is not restricted and its scheme
312                     // is TPM_ALG_NULL
313                     if(attributes.restricted || scheme != TPM_ALG_NULL)
314                         return TPM_RCS_SCHEME;
315                 }
316             }
317             else if(attributes.decrypt)
318             {
319                 if(attributes.restricted)
320                 {
321                     // for a restricted decryption key (a parent), scheme
322                     // is required to be TPM_ALG_NULL
323                     if(scheme != TPM_ALG_NULL)
324                         return TPM_RCS_SCHEME;
325                 }
326                 else
327                 {
328                     // For an unrestricted decryption key, the scheme has to
329                     // be a valid scheme or TPM_ALG_NULL
330                     if(scheme != TPM_ALG_NULL &&
331                        !CryptIsAsymDecryptScheme(publicArea->type, scheme))
```

```
332                         return TPM_RCS_SCHEME;
333                 }
334             }
335             if(!attributes.restricted || !attributes.decrypt)
336             {
337                 // For an asymmetric key that is not a parent, the symmetric
338                 // algorithms must be TPM_ALG_NULL
339                 if(symAlgs->algorithm != TPM_ALG_NULL)
340                     return TPM_RCS_SYMMETRIC;
341             }
342             // Special checks for an ECC key
343 #ifdef TPM_ALG_ECC
344             if(publicArea->type == TPM_ALG_ECC)
345             {
346                 TPM_ECC_CURVE           curveID;
347                 const TPMT_ECC_SCHEME   *curveScheme;
348                 curveID = publicArea->parameters.eccDetail.curveID;
349                 curveScheme = CryptGetCurveSignScheme(curveID);
350                 // The curveId must be valid or the unmarshaling is busted.
351                 pAssert(curveScheme != NULL);
352                 // If the curveID requires a specific scheme, then the key must
353                 // select the same scheme
354                 if(curveScheme->scheme != TPM_ALG_NULL)
355                 {
356                     TPMS_ECC_PARMS      *ecc = &publicArea->parameters.eccDetail;
357                     if(scheme != curveScheme->scheme)
358                         return TPM_RCS_SCHEME;
359                     // The scheme can allow any hash, or not...
360                     if(curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
361                        && (ecc->scheme.details.anySig.hashAlg
362                            != curveScheme->details.anySig.hashAlg))
363                         return TPM_RCS_SCHEME;
364                 }
365                 // For now, the KDF must be TPM_ALG_NULL
366                 if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
367                     return TPM_RCS_KDF;
368             }
369 #endif
370             break;
371         }
372         // If this is a restricted decryption key with symmetric algorithms, then it
373         // is an ordinary parent (not a derivation parent). It needs to specific
374         // symmetric algorithms other than TPM_ALG_NULL
375         if(symAlgs != NULL && attributes.restricted && attributes.decrypt)
376         {
377             if(symAlgs->algorithm == TPM_ALG_NULL)
378                 return TPM_RCS_SYMMETRIC;
379             // If this parent is not duplicable, then the symmetric algorithms
380             // (encryption and hash) must match those of its parent
381             if(attributes.fixedParent && (parentObject != NULL))
382             {
383                 if(publicArea->nameAlg != parentObject->publicArea.nameAlg)
384                     return TPM_RCS_HASH;
385                 if(!MemoryEqual(symAlgs, &parentObject->publicArea.parameters,
386                                 sizeof(TPMT_SYM_DEF_OBJECT)))
387                     return TPM_RCS_SYMMETRIC;
388             }
389         }
390         return TPM_RC_SUCCESS;
391 }
```


7.6.3.5 PublicAttributesValidation()

This function validates the values in the public area of an object. This function is used in the processing of TPM2_Create(), TPM2_CreatePrimary(), TPM2_CreateLoaded(), TPM2_Load(), TPM2_Import(), and TPM2_LoadExternal(). For TPM2_Import() this is only used if the new parent has *fixedTPM* SET. For TPM2_LoadExternal(), this is not used for a public-only key

Error Returns	Meaning
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	*nameAlg* is TPM_ALG_NULL
TPM_RC_SIZE	*authPolicy* size does not match digest size of the name algorithm in *publicArea*
other	returns from SchemeChecks()

```
392 TPM_RC
393 PublicAttributesValidation(
394     OBJECT          *parentObject,  // IN: input parent object
395     TPMT_PUBLIC     *publicArea     // IN: public area of the object
396     )
397 {
398     TPMA_OBJECT      attributes = publicArea->objectAttributes;
399     TPMA_OBJECT      parentAttributes = {0};
400     if(parentObject != NULL)
401         parentAttributes = parentObject->publicArea.objectAttributes;
402     if(publicArea->nameAlg == TPM_ALG_NULL)
403         return TPM_RCS_HASH;
404     // If there is an authPolicy, it needs to be the size of the digest produced
405     // by the nameAlg of the object
406     if((publicArea->authPolicy.t.size != 0
407         && (publicArea->authPolicy.t.size
408             != CryptHashGetDigestSize(publicArea->nameAlg))))
409         return TPM_RCS_SIZE;
410     // If the parent is fixedTPM (including a Primary Object) the object must have
411     // the same value for fixedTPM and fixedParent
412     if(parentObject == NULL || parentAttributes.fixedTPM == SET)
413     {
414         if(attributes.fixedParent != attributes.fixedTPM)
415             return TPM_RCS_ATTRIBUTES;
416     }
417     else
418     {
419         // The parent is not fixedTPM so the object can't be fixedTPM
420         if(attributes.fixedTPM == SET)
421             return  TPM_RCS_ATTRIBUTES;
422     }
423     // See if sign and decrypt are the same
424     if(attributes.sign == attributes.decrypt)
425     {
426         // a restricted key cannot have both SET or both CLEAR
427         if(attributes.restricted)
428             return TPM_RC_ATTRIBUTES;
429         // only a data object may have both sign and decrypt CLEAR
430         // BTW, since we know that decrypt==sign, no need to check both
431         if(publicArea->type != TPM_ALG_KEYEDHASH && !attributes.sign)
432             return TPM_RC_ATTRIBUTES;
433     }
434     // If the object can't be duplicated (directly or indirectly) then there
```

```
435         // is no justification for having encryptedDuplication SET
436         if(attributes.fixedTPM == SET && attributes.encryptedDuplication == SET)
437             return TPM_RCS_ATTRIBUTES;
438         // If a parent object has fixedTPM CLEAR, the child must have the
439         // same encryptedDuplication value as its parent.
440         // Primary objects are considered to have a fixedTPM parent (the seeds).
441         if(parentObject != NULL && parentAttributes.fixedTPM == CLEAR)
442         {
443             if(attributes.encryptedDuplication != parentAttributes.encryptedDuplication)
444                 return TPM_RCS_ATTRIBUTES;
445         }
446         // Special checks for derived objects
447         if((parentObject != NULL) && (parentObject->attributes.derivation == SET))
448         {
449             // A derived object has the same settings for fixedTPM as its parent
450             if(attributes.fixedTPM != parentAttributes.fixedTPM)
451                 return TPM_RCS_ATTRIBUTES;
452             // A derived object is required to be fixedParent
453             if(!attributes.fixedParent)
454                 return TPM_RCS_ATTRIBUTES;
455         }
456         return SchemeChecks(parentObject, publicArea);
457     }
```

7.6.3.6 FillInCreationData()

Fill in creation data for an object.

```
458     void
459     FillInCreationData(
460         TPMI_DH_OBJECT           parentHandle,  // IN: handle of parent
461         TPMI_ALG_HASH            nameHashAlg,   // IN: name hash algorithm
462         TPML_PCR_SELECTION      *creationPCR,   // IN: PCR selection
463         TPM2B_DATA              *outsideData,   // IN: outside data
464         TPM2B_CREATION_DATA     *outCreation,   // OUT: creation data for output
465         TPM2B_DIGEST            *creationDigest // OUT: creation digest
466         )
467     {
468         BYTE                creationBuffer[sizeof(TPMS_CREATION_DATA)];
469         BYTE               *buffer;
470         HASH_STATE          hashState;
471         // Fill in TPMS_CREATION_DATA in outCreation
472         // Compute PCR digest
473         PCRComputeCurrentDigest(nameHashAlg, creationPCR,
474                                 &outCreation->creationData.pcrDigest);
475         // Put back PCR selection list
476         outCreation->creationData.pcrSelect = *creationPCR;
477         // Get locality
478         outCreation->creationData.locality
479             = LocalityGetAttributes(_plat__LocalityGet());
480         outCreation->creationData.parentNameAlg = TPM_ALG_NULL;
481         // If the parent is either a primary seed or TPM_ALG_NULL, then  the Name
482         // and QN of the parent are the parent's handle.
483         if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
484         {
485             buffer = &outCreation->creationData.parentName.t.name[0];
486             outCreation->creationData.parentName.t.size =
487                 TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
488             // For a primary or temporary object, the parent name (a handle) and the
489             // parent's QN are the same
490             outCreation->creationData.parentQualifiedName
491                 = outCreation->creationData.parentName;
492         }
493         else         // Regular object
```



```
494         {
495             OBJECT          *parentObject = HandleToObject(parentHandle);
496             // Set name algorithm
497             outCreation->creationData.parentNameAlg =
498                 parentObject->publicArea.nameAlg;
499             // Copy parent name
500             outCreation->creationData.parentName = parentObject->name;
501             // Copy parent qualified name
502             outCreation->creationData.parentQualifiedName =
503                 parentObject->qualifiedName;
504         }
505         // Copy outside information
506         outCreation->creationData.outsideInfo = *outsideData;
507         // Marshal creation data to canonical form
508         buffer = creationBuffer;
509         outCreation->size = TPMS_CREATION_DATA_Marshal(&outCreation->creationData,
510                                                        &buffer, NULL);
511                                 // Compute hash for creation field in public template
512         creationDigest->t.size = CryptHashStart(&hashState, nameHashAlg);
513         CryptDigestUpdate(&hashState, outCreation->size, creationBuffer);
514         CryptHashEnd2B(&hashState, &creationDigest->b);
515         return;
516     }
```

7.6.3.7 GetSeedForKDF()

Get a seed for KDF. The KDF for encryption and HMAC key use the same seed.

Return Value	Meaning

```
517     const TPM2B *
518     GetSeedForKDF(
519         OBJECT          *protector          // IN: the protector handle
520         )
521     {
522         // Get seed for encryption key.  Use input seed if provided.
523         // Otherwise, using protector object's seedValue.  TPM_RH_NULL is the only
524         // exception that we may not have a loaded object as protector.  In such a
525         // case, use nullProof as seed.
526         if(protector == NULL)
527             return &gr.nullProof.b;
528         else
529             return &protector->sensitive.seedValue.b;
530     }
```

7.6.3.8 ProduceOuterWrap()

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

a) Add IV before sensitive area if required

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

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

```
531     UINT16
532     ProduceOuterWrap(
533         OBJECT          *protector,      // IN: The handle of the object that provides
534                                         //     protection.  For object, it is parent
```

```
535                                     //     handle. For credential, it is the handle
536                                     //     of encrypt object.
537     TPM2B           *name,          // IN: the name of the object
538     TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
539     TPM2B           *seed,          // IN: an external seed may be provided for
540                                     //     duplication blob. For non duplication
541                                     //     blob, this parameter should be NULL
542     BOOL             useIV,         // IN: indicate if an IV is used
543     UINT16           dataSize,      // IN: the size of sensitive data, excluding the
544                                     //     leading integrity buffer size or the
545                                     //     optional iv size
546     BYTE            *outerBuffer    // IN/OUT: outer buffer with sensitive data in
547                                     //     it
548     )
549 {
550     TPM_ALG_ID      symAlg;
551     UINT16          keyBits;
552     TPM2B_SYM_KEY   symKey;
553     TPM2B_IV        ivRNG;          // IV from RNG
554     TPM2B_IV        *iv = NULL;
555     UINT16          ivSize = 0;     // size of iv area, including the size field
556     BYTE            *sensitiveData; // pointer to the sensitive data
557     TPM2B_DIGEST    integrity;
558     UINT16          integritySize;
559     BYTE            *buffer;        // Auxiliary buffer pointer
560     // Compute the beginning of sensitive data.  The outer integrity should
561     // always exist if this function is called to make an outer wrap
562     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
563     sensitiveData = outerBuffer + integritySize;
564     // If iv is used, adjust the pointer of sensitive data and add iv before it
565     if(useIV)
566     {
567         ivSize = GetIV2BSize(protector);
568         // Generate IV from RNG.  The iv data size should be the total IV area
569         // size minus the size of size field
570         ivRNG.t.size = ivSize - sizeof(UINT16);
571         CryptRandomGenerate(ivRNG.t.size, ivRNG.t.buffer);
572         // Marshal IV to buffer
573         buffer = sensitiveData;
574         TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
575         // adjust sensitive data starting after IV area
576         sensitiveData += ivSize;
577         // Use iv for encryption
578         iv = &ivRNG;
579     }
580     // Compute symmetric key parameters for outer buffer encryption
581     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
582                               &symAlg, &keyBits, &symKey);
583     // Encrypt inner buffer in place
584     CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
585                           symKey.t.buffer, iv, TPM_ALG_CFB, dataSize,
586                           sensitiveData);
587     // Compute outer integrity.  Integrity computation includes the optional IV
588     // area
589     ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
590                           outerBuffer + integritySize, &integrity);
591     // Add integrity at the beginning of outer buffer
592     buffer = outerBuffer;
593     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
594     // return the total size in outer wrap
595     return dataSize + integritySize + ivSize;
596 }
```

7.6.3.9 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_RCS_INSUFFICIENT	error during sensitive data unmarshaling
TPM_RCS_INTEGRITY	sensitive data integrity is broken
TPM_RCS_SIZE	error during sensitive data unmarshaling
TPM_RCS_VALUE	IV size for CFB does not match the encryption algorithm block size

```
597  TPM_RC
598  UnwrapOuter(
599      OBJECT          *protector,     // IN: The object that provides
600                                      //     protection.  For object, it is parent
601                                      //     handle. For credential, it is the
602                                      //     encrypt object.
603      TPM2B           *name,          // IN: the name of the object
604      TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
605      TPM2B           *seed,          // IN: an external seed may be provided for
606                                      //     duplication blob. For non duplication
607                                      //     blob, this parameter should be NULL.
608      BOOL             useIV,         // IN: indicates if an IV is used
609      UINT16           dataSize,      // IN: size of sensitive data in outerBuffer,
610                                      //     including the leading integrity buffer
611                                      //     size, and an optional iv area
612      BYTE            *outerBuffer    // IN/OUT: sensitive data
613      )
614  {
615      TPM_RC          result;
616      TPM_ALG_ID      symAlg = TPM_ALG_NULL;
617      TPM2B_SYM_KEY   symKey;
618      UINT16          keyBits = 0;
619      TPM2B_IV        ivIn;               // input IV retrieved from input buffer
620      TPM2B_IV        *iv = NULL;
621      BYTE            *sensitiveData;     // pointer to the sensitive data
622      TPM2B_DIGEST    integrityToCompare;
623      TPM2B_DIGEST    integrity;
624      INT32           size;
625      // Unmarshal integrity
626      sensitiveData = outerBuffer;
627      size = (INT32)dataSize;
628      result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
629      if(result == TPM_RC_SUCCESS)
630      {
631          // Compute integrity to compare
632          ComputeOuterIntegrity(name, protector, hashAlg, seed,
633                                (UINT16)size, sensitiveData,
634                                &integrityToCompare);
635          // Compare outer blob integrity
636          if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
637              return TPM_RCS_INTEGRITY;
638          // Get the symmetric algorithm parameters used for encryption
639          ComputeProtectionKeyParms(protector, hashAlg, name, seed,
640                                    &symAlg, &keyBits, &symKey);
641          // Retrieve IV if it is used
642          if(useIV)
643          {
644              result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
```

```
645             if(result == TPM_RC_SUCCESS)
646             {
647                 // The input iv size for CFB must match the encryption algorithm
648                 // block size
649                 if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
650                     result = TPM_RC_VALUE;
651                 else
652                     iv = &ivIn;
653             }
654         }
655     }
656     // If no errors, decrypt private in place. Since this function uses CFB,
657     // CryptSymmetricDecrypt() will not return any errors (it may fail but it will
658     // not return an error.
659     if(result == TPM_RC_SUCCESS)
660         CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
661                               symKey.t.buffer, iv, TPM_ALG_CFB,
662                               (UINT16)size, sensitiveData);
663     return result;
664 }
```

7.6.3.10 MarshalSensitive()

This function is used to marshal a sensitive area. Among other things, it adjusts the size of the *authValue* to be no smaller than the digest of *nameAlg* Returns the size of the marshaled area.

```
665 static UINT16
666 MarshalSensitive(
667     BYTE                *buffer,            // OUT: receiving buffer
668     TPMT_SENSITIVE      *sensitive,         // IN: the sensitive area to marshal
669     TPMI_ALG_HASH        nameAlg            // IN:
670     )
671 {
672     BYTE                *sizeField = buffer;
673     UINT16               retVal;
674    // Pad the authValue if needed
675     MemoryPad2B(&sensitive->authValue.b, CryptHashGetDigestSize(nameAlg));
676     buffer += 2;
677     // Marshal the structure
678     retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
679     // Marshal the size
680     retVal = (UINT16)(retVal + UINT16_Marshal(&retVal, &sizeField, NULL));
681     return retVal;
682 }
```

7.6.3.11 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.

```
683 void
684 SensitiveToPrivate(
685     TPMT_SENSITIVE  *sensitive,     // IN: sensitive structure
686     TPM2B           *name,          // IN: the name of the object
687     OBJECT          *parent,        // IN: The parent object
688     TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  This
689                                     //     parameter is used when parentHandle is
690                                     //     NULL, in which case the object is
```

```
691                                     //     temporary.
692     TPM2B_PRIVATE   *outPrivate     // OUT: output private structure
693     )
694 {
695     BYTE                *sensitiveData;     // pointer to the sensitive data
696     UINT16              dataSize;           // data blob size
697     TPMI_ALG_HASH       hashAlg;            // hash algorithm for integrity
698     UINT16              integritySize;
699     UINT16              ivSize;
700 //
701     pAssert(name != NULL && name->size != 0);
702     // Find the hash algorithm for integrity computation
703     if(parent == NULL)
704     {
705         // For Temporary Object, using self name algorithm
706         hashAlg = nameAlg;
707     }
708     else
709     {
710         // Otherwise, using parent's name algorithm
711         hashAlg = ObjectGetNameAlg(parent);
712     }
713     // Starting of sensitive data without wrappers
714     sensitiveData = outPrivate->t.buffer;
715     // Compute the integrity size
716     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
717     // Reserve space for integrity
718     sensitiveData += integritySize;
719     // Get iv size
720     ivSize = GetIV2BSize(parent);
721     // Reserve space for iv
722     sensitiveData += ivSize;
723     // Marshal the sensitive area including authValue size adjustments.
724     dataSize = MarshalSensitive(sensitiveData, sensitive, nameAlg);
725     //Produce outer wrap, including encryption and HMAC
726     outPrivate->t.size = ProduceOuterWrap(parent, name, hashAlg, NULL,
727                                           TRUE, dataSize, outPrivate->t.buffer);
728     return;
729 }
```

7.6.3.12 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_RCS_INTEGRITY	if the private area integrity is bad
TPM_RC_SENSITIVE	unmarshal errors while unmarshaling TPMS_ENCRYPT from input private
TPM_RCS_SIZE	error during sensitive data unmarshaling
TPM_RCS_VALUE	outer wrapper does not have an *iV* of the correct size

```
730 TPM_RC
731 PrivateToSensitive(
732     TPM2B           *inPrivate,     // IN: input private structure
733     TPM2B           *name,          // IN: the name of the object
```

```
734         OBJECT          *parent,        // IN: parent object
735         TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  It is
736                                         //     passed separately because we only pass
737                                         //     name, rather than the whole public area
738                                         //     of the object.  This parameter is used in
739                                         //     the following two cases: 1. primary
740                                         //     objects. 2. duplication blob with inner
741                                         //     wrap.  In other cases, this parameter
742                                         //     will be ignored
743         TPMT_SENSITIVE  *sensitive      // OUT: sensitive structure
744         )
745     {
746         TPM_RC          result;
747         BYTE            *buffer;
748         INT32           size;
749         BYTE            *sensitiveData; // pointer to the sensitive data
750         UINT16          dataSize;
751         UINT16          dataSizeInput;
752         TPMI_ALG_HASH   hashAlg;        // hash algorithm for integrity
753         UINT16          integritySize;
754         UINT16          ivSize;
755     //
756         // Make sure that name is provided
757         pAssert(name != NULL && name->size != 0);
758         // Find the hash algorithm for integrity computation
759         if(parent == NULL)
760         {
761             // For Temporary Object, using self name algorithm
762             hashAlg = nameAlg;
763         }
764         else
765         {
766             // Otherwise, using parent's name algorithm
767             hashAlg = ObjectGetNameAlg(parent);
768         }
769         // unwrap outer
770         result = UnwrapOuter(parent, name, hashAlg, NULL, TRUE,
771                              inPrivate->size, inPrivate->buffer);
772         if(result != TPM_RC_SUCCESS)
773             return result;
774         // Compute the inner integrity size.
775         integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
776         // Get iv size
777         ivSize = GetIV2BSize(parent);
778         // The starting of sensitive data and data size without outer wrapper
779         sensitiveData = inPrivate->buffer + integritySize + ivSize;
780         dataSize = inPrivate->size - integritySize - ivSize;
781         // Unmarshal input data size
782         buffer = sensitiveData;
783         size = (INT32)dataSize;
784         result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
785         if(result == TPM_RC_SUCCESS)
786         {
787             if((dataSizeInput + sizeof(UINT16)) != dataSize)
788                 result = TPM_RC_SENSITIVE;
789             else
790             {
791                 // Unmarshal sensitive buffer to sensitive structure
792                 result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
793                 if(result != TPM_RC_SUCCESS || size != 0)
794                 {
795                     result = TPM_RC_SENSITIVE;
796                 }
797             }
798         }
799         return result;
```



```
}
```

7.6.3.13 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

```
void
SensitiveToDuplicate(
    TPMT_SENSITIVE      *sensitive,     // IN: sensitive structure
    TPM2B               *name,          // IN: the name of the object
    OBJECT              *parent,        // IN: The new parent object
    TPM_ALG_ID           nameAlg,       // IN: hash algorithm in public area. It
                                        //     is passed separately because we
                                        //     only pass name, rather than the
                                        //     whole public area of the object.
    TPM2B               *seed,          // IN: the external seed. If external
                                        //     seed is provided with size of 0,
                                        //     no outer wrap should be applied
                                        //     to duplication blob.
    TPMT_SYM_DEF_OBJECT *symDef,        // IN: Symmetric key definition. If the
                                        //     symmetric key algorithm is NULL,
                                        //     no inner wrap should be applied.
    TPM2B_DATA          *innerSymKey,   // IN/OUT: a symmetric key may be
                                        //     provided to encrypt the inner
                                        //     wrap of a duplication blob. May
                                        //     be generated here if needed.
    TPM2B_PRIVATE       *outPrivate     // OUT: output private structure
    )
{
    BYTE            *buffer;        // Auxiliary buffer pointer
    BYTE            *sensitiveData; // pointer to the sensitive data
    TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;// The hash algorithm for outer wrap
    TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;// The hash algorithm for inner wrap
    UINT16          dataSize;       // data blob size
    BOOL            doInnerWrap = FALSE;
    BOOL            doOuterWrap = FALSE;
//
    // Make sure that name is provided
    pAssert(name != NULL && name->size != 0);
    // Make sure symDef and innerSymKey are not NULL
    pAssert(symDef != NULL && innerSymKey != NULL);
    // Starting of sensitive data without wrappers
    sensitiveData = outPrivate->t.buffer;
    // Find out if inner wrap is required
    if(symDef->algorithm != TPM_ALG_NULL)
    {
        doInnerWrap = TRUE;
        // Use self nameAlg as inner hash algorithm
        innerHash = nameAlg;
        // Adjust sensitive data pointer
        sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(innerHash);
    }
    // Find out if outer wrap is required
    if(seed->size != 0)
    {
        doOuterWrap = TRUE;
        // Use parent nameAlg as outer hash algorithm
        outerHash = ObjectGetNameAlg(parent);
        // Adjust sensitive data pointer
```

```
854             sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
855         }
856         // Marshal sensitive area, leaving the leading 2 bytes for size
857         buffer = sensitiveData + sizeof(UINT16);
858         dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
859         // Adding size before the data area
860         buffer = sensitiveData;
861         UINT16_Marshal(&dataSize, &buffer, NULL);
862         // Adjust the dataSize to include the size field
863         dataSize += sizeof(UINT16);
864         dataSize = MarshalSensitive(sensitiveData, sensitive, nameAlg);
865         // Apply inner wrap for duplication blob.  It includes both integrity and
866         // encryption
867         if(doInnerWrap)
868         {
869             BYTE            *innerBuffer = NULL;
870             BOOL            symKeyInput = TRUE;
871             innerBuffer = outPrivate->t.buffer;
872             // Skip outer integrity space
873             if(doOuterWrap)
874                 innerBuffer += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
875             dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
876                                              innerBuffer);
877             // Generate inner encryption key if needed
878             if(innerSymKey->t.size == 0)
879             {
880                 innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
881                 CryptRandomGenerate(innerSymKey->t.size, innerSymKey->t.buffer);
882                 // TPM generates symmetric encryption.  Set the flag to FALSE
883                 symKeyInput = FALSE;
884             }
885             else
886             {
887                 // assume the input key size should matches the symmetric definition
888                 pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
889             }
890             // Encrypt inner buffer in place
891             CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
892                                   symDef->keyBits.sym, innerSymKey->t.buffer, NULL,
893                                   TPM_ALG_CFB, dataSize, innerBuffer);
894             // If the symmetric encryption key is imported, clear the buffer for
895             // output
896             if(symKeyInput)
897                 innerSymKey->t.size = 0;
898         }
899         // Apply outer wrap for duplication blob.  It includes both integrity and
900         // encryption
901         if(doOuterWrap)
902         {
903             dataSize = ProduceOuterWrap(parent, name, outerHash, seed, FALSE,
904                                         dataSize, outPrivate->t.buffer);
905         }
906         // Data size for output
907         outPrivate->t.size = dataSize;
908         return;
909     }
```


7.6.3.14 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

```
910  TPM_RC
911  DuplicateToSensitive(
912      TPM2B               *inPrivate,     // IN: input private structure
913      TPM2B               *name,          // IN: the name of the object
914      OBJECT              *parent,        // IN: the parent
915      TPM_ALG_ID           nameAlg,       // IN: hash algorithm in public area.
916      TPM2B               *seed,          // IN: an external seed may be provided.
917                                          //     If external seed is provided with
918                                          //     size of 0, no outer wrap is
919                                          //     applied
920      TPMT_SYM_DEF_OBJECT *symDef,        // IN: Symmetric key definition. If the
921                                          //     symmetric key algorithm is NULL,
922                                          //     no inner wrap is applied
923      TPM2B               *innerSymKey,   // IN: a symmetric key may be provided
924                                          //     to decrypt the inner wrap of a
925                                          //     duplication blob.
926      TPMT_SENSITIVE      *sensitive      // OUT: sensitive structure
927      )
928  {
929      TPM_RC               result;
930      BYTE                *buffer;
931      INT32                size;
932      BYTE                *sensitiveData; // pointer to the sensitive data
933      UINT16               dataSize;
934      UINT16               dataSizeInput;
935      // Make sure that name is provided
936      pAssert(name != NULL && name->size != 0);
937      // Make sure symDef and innerSymKey are not NULL
938      pAssert(symDef != NULL && innerSymKey != NULL);
939      // Starting of sensitive data
940      sensitiveData = inPrivate->buffer;
941      dataSize = inPrivate->size;
942      // Find out if outer wrap is applied
943      if(seed->size != 0)
944      {
945          // Use parent nameAlg as outer hash algorithm
946          TPMI_ALG_HASH   outerHash = parent->publicArea.nameAlg;
947          result = UnwrapOuter(parent, name, outerHash, seed, FALSE,
948                               dataSize, sensitiveData);
949          if(result != TPM_RC_SUCCESS)
950              return result;
951          // Adjust sensitive data pointer and size
952          sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
953          dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
954      }
955      // Find out if inner wrap is applied
956      if(symDef->algorithm != TPM_ALG_NULL)
```

```
957         {
958             // assume the input key size matches the symmetric definition
959             pAssert(innerSymKey->size == (symDef->keyBits.sym + 7) / 8);
960             // Decrypt inner buffer in place
961             CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
962                                   symDef->keyBits.sym, innerSymKey->buffer, NULL,
963                                   TPM_ALG_CFB, dataSize, sensitiveData);
964             // Check inner integrity
965             result = CheckInnerIntegrity(name, nameAlg, dataSize, sensitiveData);
966             if(result != TPM_RC_SUCCESS)
967                 return result;
968             // Adjust sensitive data pointer and size
969             sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
970             dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
971         }
972         // Unmarshal input data size
973         buffer = sensitiveData;
974         size = (INT32)dataSize;
975         result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
976         if(result == TPM_RC_SUCCESS)
977         {
978             if((dataSizeInput + sizeof(UINT16)) != dataSize)
979                 result = TPM_RC_SIZE;
980             else
981             {
982                 // Unmarshal sensitive buffer to sensitive structure
983                 result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
984                 // if the results is OK make sure that all the data was unmarshaled
985                 if(result == TPM_RC_SUCCESS && size != 0)
986                     result = TPM_RC_SIZE;
987             }
988         }
989         return result;
990     }
```

7.6.3.15 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

```
991     void
992     SecretToCredential(
993         TPM2B_DIGEST        *secret,        // IN: secret information
994         TPM2B               *name,          // IN: the name of the object
995         TPM2B               *seed,          // IN: an external seed.
996         OBJECT              *protector,     // IN: the protector
997         TPM2B_ID_OBJECT     *outIDObject    // OUT: output credential
998         )
999     {
1000        BYTE                *buffer;        // Auxiliary buffer pointer
1001        BYTE                *sensitiveData; // pointer to the sensitive data
1002        TPMI_ALG_HASH        outerHash;     // The hash algorithm for outer wrap
1003        UINT16               dataSize;      // data blob size
1004        pAssert(secret != NULL && outIDObject != NULL);
1005        // use protector's name algorithm as outer hash
1006        outerHash = ObjectGetNameAlg(protector);
1007        // Marshal secret area to credential buffer, leave space for integrity
```

```
1008        sensitiveData = outIDObject->t.credential
1009            + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1010    // Marshal secret area
1011        buffer = sensitiveData;
1012        dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1013        // Apply outer wrap
1014        outIDObject->t.size = ProduceOuterWrap(protector,
1015                                               name,
1016                                               outerHash,
1017                                               seed,
1018                                               FALSE,
1019                                               dataSize,
1020                                               outIDObject->t.credential);
1021        return;
1022    }
```

7.6.3.16 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

```
1023    TPM_RC
1024    CredentialToSecret(
1025        TPM2B               *inIDObject,    // IN: input credential blob
1026        TPM2B               *name,          // IN: the name of the object
1027        TPM2B               *seed,          // IN: an external seed.
1028        OBJECT              *protector,     // IN: the protector
1029        TPM2B_DIGEST        *secret         // OUT: secret information
1030        )
1031    {
1032        TPM_RC                   result;
1033        BYTE                    *buffer;
1034        INT32                    size;
1035        TPMI_ALG_HASH            outerHash;     // The hash algorithm for outer wrap
1036        BYTE                    *sensitiveData; // pointer to the sensitive data
1037        UINT16                   dataSize;
1038        // use protector's name algorithm as outer hash
1039        outerHash = ObjectGetNameAlg(protector);
1040        // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1041        result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
1042                             inIDObject->size, inIDObject->buffer);
1043        if(result == TPM_RC_SUCCESS)
1044        {
1045            // Compute the beginning of sensitive data
1046            sensitiveData = inIDObject->buffer
1047                + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1048            dataSize = inIDObject->size
1049                - (sizeof(UINT16) + CryptHashGetDigestSize(outerHash));
1050     // Unmarshal secret buffer to TPM2B_DIGEST structure
1051            buffer = sensitiveData;
```

```
1052            size = (INT32)dataSize;
1053            result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1054            // If there were no other unmarshaling errors, make sure that the
1055            // expected amount of data was recovered
1056            if(result == TPM_RC_SUCCESS && size != 0)
1057                return TPM_RC_SIZE;
1058        }
1059        return result;
1060    }
```

7.6.3.17 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.

```
1061    UINT16
1062    MemoryRemoveTrailingZeros(
1063        TPM2B_AUTH      *auth           // IN/OUT: value to adjust
1064        )
1065    {
1066        while((auth->t.size > 0) && (auth->t.buffer[auth->t.size - 1] == 0))
1067            auth->t.size--;
1068        return auth->t.size;
1069    }
```

7.6.3.18 SetLabelAndContext()

This function sets the label and context for a derived key. It is possible that *label* or *context* can end up being an Empty Buffer.

```
1070    TPM_RC
1071    SetLabelAndContext(
1072        TPMT_PUBLIC             *publicArea,    // IN/OUT: the public area containing
1073                                                //         the unmarshaled template
1074        TPM2B_SENSITIVE_DATA    *sensitive      // IN: the sensitive data
1075        )
1076    {
1077        TPM_RC                   result;
1078        INT32                    size;
1079        BYTE                    *buff;
1080        TPM2B_LABEL              label;
1081        // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
1082        size = sensitive->t.size;
1083        // If there is something to unmarshal...
1084        if(size != 0)
1085        {
1086            buff = sensitive->t.buffer;
1087            result = TPM2B_LABEL_Unmarshal(&label, &buff, &size);
1088            if(result != TPM_RC_SUCCESS)
1089                return result;
1090            // If there is a label in the publicArea, it overrides
1091            if(publicArea->unique.derive.label.t.size == 0)
1092                MemoryCopy2B(&publicArea->unique.derive.label.b, &label.b,
1093                             sizeof(publicArea->unique.derive.label.t.buffer));
1094            result = TPM2B_LABEL_Unmarshal(&label, &buff, &size);
1095            if(result != TPM_RC_SUCCESS)
1096                return result;
1097            if(publicArea->unique.derive.context.t.size == 0)
1098                MemoryCopy2B(&publicArea->unique.derive.context.b, &label.b,
1099                             sizeof(publicArea->unique.derive.context.t.buffer));
1100        }
```

```
1101        return TPM_RC_SUCCESS;
1102    }
```

7.6.3.19 UnmarshalToPublic()

Support function to unmarshal the template. This is used because the Input may be a TPMT_TEMPLATE and that structure does not have the same size as a TPMT_PUBIIC() because of the difference between the *unique* and *seed* fields. If *derive* is not NULL, then the *seed* field is assumed to contain a *label* and *context* that are unmarshaled into *derive*.

```
1103    TPM_RC
1104    UnmarshalToPublic(
1105        TPMT_PUBLIC         *tOut,       // OUT: output
1106        TPM2B_TEMPLATE      *tIn,        // IN:
1107        BOOL                 derivation  // IN: indicates if this is for a derivation
1108        )
1109    {
1110        BYTE                *buffer = tIn->t.buffer;
1111        INT32                size = tIn->t.size;
1112        TPM_RC               result;
1113    //
1114        // make sure that tOut is zeroed so that there are no remnants from previous
1115        // uses
1116        MemorySet(tOut, 0, sizeof(TPMT_PUBLIC));
1117        // Unmarshal a TPMT_PUBLIC but don't allow a nameAlg of TPM_ALG_NULL
1118        result = TPMT_PUBLIC_Unmarshal(tOut, &buffer, &size, FALSE);
1119        if((result == TPM_RC_SUCCESS) && (derivation == TRUE))
1120        {
1121    #if ALG_ECC
1122            // If we just unmarshaled an ECC public key, then the label value is in the
1123            // correct spot but the context value is in the wrong place if the
1124            // maximum ECC parameter size is larger than 32 bytes. So, move it. This
1125            if(tOut->type == ALG_ECC_VALUE)
1126            {
1127                // This could probably be a direct copy because we are moving data
1128                // to lower addresses but, just to be safe...
1129                TPM2B_LABEL      context;
1130                MemoryCopy2B(&context.b, &tOut->unique.ecc.y.b,
1131                             sizeof(context.t.buffer));
1132                MemoryCopy2B(&tOut->unique.derive.context.b, &context.b,
1133                             sizeof(tOut->unique.derive.context.t.buffer));
1134            }
1135            else
1136    #endif
1137                // For object types other than ECC, should have completed unmarshaling
1138                // with data left in the buffer so try to unmarshal the remainder as a
1139                // TPM2B_LABEL into the context
1140                result = TPM2B_LABEL_Unmarshal(&tOut->unique.derive.context,
1141                                               &buffer, &size);
1142        }
1143        return result;
1144    }
```

7.6.3.20 ObjectSetHierarchy()

```
1145    void
1146    ObjectSetHierarchy(
1147        OBJECT              *object,
1148        TPM_HANDLE           parentHandle,
1149        OBJECT              *parent
1150        )
1151    {
1152        if(parent == NULL)
```

```
1153        {
1154            switch(parentHandle)
1155            {
1156                case TPM_RH_ENDORSEMENT:
1157                    object->attributes.epsHierarchy = SET;
1158                    break;
1159                case TPM_RH_OWNER:
1160                    object->attributes.spsHierarchy = SET;
1161                    break;
1162                case TPM_RH_PLATFORM:
1163                    object->attributes.ppsHierarchy = SET;
1164                    break;
1165                default:
1166                    break;
1167            }
1168        }
1169        else
1170        {
1171            object->attributes.epsHierarchy = parent->attributes.epsHierarchy;
1172            object->attributes.spsHierarchy = parent->attributes.spsHierarchy;
1173            object->attributes.ppsHierarchy = parent->attributes.ppsHierarchy;
1174        }
1175    }
```

7.6.3.21 ObjectSetExternal()

Set the external attributes for an object.

```
1176    void
1177    ObjectSetExternal(
1178        OBJECT      *object
1179        )
1180    {
1181        object->attributes.external = SET;
1182    }
```

7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c)

```
1   #include "Tpm.h"
2   #include "EncryptDecrypt_fp.h"
3   #ifdef TPM_CC_EncryptDecrypt2
```

Error Returns	Meaning
TPM_RC_KEY	is not a symmetric decryption key with both public and private portions loaded
TPM_RC_SIZE	*IvIn* size is incompatible with the block cipher mode; or *inData* size is not an even multiple of the block size for CBC or ECB mode
TPM_RC_VALUE	*keyHandle* is restricted and the argument *mode* does not match the key's mode

```
4   TPM_RC
5   EncryptDecryptShared(
6       TPMI_DH_OBJECT          keyHandleIn,
7       TPMI_YES_NO             decryptIn,
8       TPMI_ALG_SYM_MODE       modeIn,
9       TPM2B_IV                *ivIn,
10      TPM2B_MAX_BUFFER        *inData,
11      EncryptDecrypt_Out      *out
12      )
13  {
14      OBJECT              *symKey;
15      UINT16               keySize;
16      UINT16               blockSize;
17      BYTE                *key;
18      TPM_ALG_ID           alg;
19      TPM_ALG_ID           mode;
20      TPM_RC               result;
21      BOOL                 OK;
22  // Input Validation
23      symKey = HandleToObject(keyHandleIn);
24      mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;
25      // The input key should be a symmetric key
26      if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
27          return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
28      // The key must be unrestricted and allow the selected operation
29      OK = symKey->publicArea.objectAttributes.restricted != SET;
30      if(YES == decryptIn)
31          OK = OK && symKey->publicArea.objectAttributes.decrypt == SET;
32      else
33          OK = OK && symKey->publicArea.objectAttributes.sign == SET;
34      if(!OK)
35          return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
36      // If the key mode is not TPM_ALG_NULL...
37      // or TPM_ALG_NULL
38      if(mode != TPM_ALG_NULL)
39      {
40          // then the input mode has to be TPM_ALG_NULL or the same as the key
41          if((modeIn != TPM_ALG_NULL) && (modeIn != mode))
42              return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
43      }
44      else
45      {
46          // if the key mode is null, then the input can't be null
47          if(modeIn == TPM_ALG_NULL)
48              return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
49          mode = modeIn;
50      }
```



```
51          // The input iv for ECB mode should be an Empty Buffer.  All the other modes
52          // should have an iv size same as encryption block size
53          keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
54          alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
55          blockSize = CryptGetSymmetricBlockSize(alg, keySize);
56          // Note: When an algorithm is not supported by a TPM, the TPM_ALG_xxx for that
57          // algorithm is not defined. However, it is assumed that the ALG_xxx_VALUE for
58          // the algorithm is always defined. Both have the same numeric value.
59          // ALG_xxx_VALUE is used here so that the code does not get cluttered with
60          // #ifdef's. Having this check does not mean that the algorithm is supported.
61          // If it was not supported the unmarshaling code would have rejected it before
62          // this function were called. This means that, depending on the implementation,
63          // the check could be redundant but it doesn't hurt.
64          if(((modeIn == ALG_ECB_VALUE) && (ivIn->t.size != 0))
65             || ((modeIn != ALG_ECB_VALUE) && (ivIn->t.size != blockSize)))
66              return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
67          // The input data size of CBC mode or ECB mode must be an even multiple of
68          // the symmetric algorithm's block size
69          if(((modeIn == ALG_CBC_VALUE) || (modeIn == ALG_ECB_VALUE))
70             && ((inData->t.size % blockSize) != 0))
71              return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
72          // Copy IV
73          // Note: This is copied here so that the calls to the encrypt/decrypt functions
74          // will modify the output buffer, not the input buffer
75          out->ivOut = *ivIn;
76      // Command Output
77          key = symKey->sensitive.sensitive.sym.t.buffer;
78          // For symmetric encryption, the cipher data size is the same as plain data
79          // size.
80          out->outData.t.size = inData->t.size;
81          if(decryptIn == YES)
82          {
83              // Decrypt data to output
84              result = CryptSymmetricDecrypt(out->outData.t.buffer, alg, keySize, key,
85                                             &(out->ivOut), mode, inData->t.size,
86                                             inData->t.buffer);
87          }
88          else
89          {
90              // Encrypt data to output
91              result = CryptSymmetricEncrypt(out->outData.t.buffer, alg, keySize, key,
92                                             &(out->ivOut), mode, inData->t.size,
93                                             inData->t.buffer);
94          }
95          return result;
96      }
97      #endif // CC_EncryptDecrypt
```


8 Subsystem

8.1 CommandAudit.c

8.1.1 Introduction

This file contains the functions that support command audit.

8.1.2 Includes

```
#include "Tpm.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.auditCommands, 0x00, sizeof(gp.auditCommands));
    // TPM_CC_SetCommandCodeAuditStatus always being audited
    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
    NV_SYNC_PERSISTENT(auditCommands);
    NV_SYNC_PERSISTENT(auditHashAlg);
    NV_SYNC_PERSISTENT(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_RESTART) && (type != SU_RESUME))
    {
        // Reset the digest size to initialize the digest
        gr.commandAuditDigest.t.size = 0;
    }
}
```

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

```
33  BOOL
34  CommandAuditSet(
35      TPM_CC            commandCode     // IN: command code
36      )
37  {
38      COMMAND_INDEX         commandIndex = CommandCodeToCommandIndex(commandCode);
39      // Only SET a bit if the corresponding command is implemented
40      if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
41      {
42          // Can't audit shutdown
43          if(commandCode != TPM_CC_Shutdown)
44          {
45              if(!TEST_BIT(commandIndex, gp.auditCommands))
46              {
47                  // Set bit
48                  SET_BIT(commandIndex, gp.auditCommands);
49                  return TRUE;
50              }
51          }
52      }
53      // No change
54      return FALSE;
55  }
```

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

```
56  BOOL
57  CommandAuditClear(
58      TPM_CC            commandCode     // IN: command code
59      )
60  {
61      COMMAND_INDEX         commandIndex = CommandCodeToCommandIndex(commandCode);
62      // Do nothing if the command is not implemented
```

```
63      if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
64      {
65          // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
66          // cleared
67          if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
68          {
69              if(TEST_BIT(commandIndex, gp.auditCommands))
70              {
71                  // Clear bit
72                  CLEAR_BIT(commandIndex, gp.auditCommands);
73                  return TRUE;
74              }
75          }
76      }
77      // No change
78      return FALSE;
79  }
```

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

```
80  BOOL
81  CommandAuditIsRequired(
82      COMMAND_INDEX    commandIndex   // IN: command index
83      )
84  {
85      // Check the bit map.  If the bit is SET, command audit is required
86      return(TEST_BIT(commandIndex, gp.auditCommands));
87  }
```

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

```
88  TPMI_YES_NO
89  CommandAuditCapGetCCList(
90      TPM_CC           commandCode,   // IN: start command code
91      UINT32           count,         // IN: count of returned TPM_CC
92      TPML_CC         *commandList    // OUT: list of TPM_CC
93      )
94  {
95      TPMI_YES_NO     more = NO;
96      COMMAND_INDEX   commandIndex;
97      // Initialize output handle list
98      commandList->count = 0;
99      // The maximum count of command we may return is MAX_CAP_CC
100     if(count > MAX_CAP_CC) count = MAX_CAP_CC;
101     // Find the implemented command that has a command code that is the same or
```

```
102         // higher than the input
103         // Collect audit commands
104         for(commandIndex = GetClosestCommandIndex(commandCode);
105         commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
106             commandIndex = GetNextCommandIndex(commandIndex))
107         {
108             if(CommandAuditIsRequired(commandIndex))
109             {
110                 if(commandList->count < count)
111                 {
112                     // If we have not filled up the return list, add this command
113                     // code to its
114                     TPM_CC      cc = s_ccAttr[commandIndex].commandIndex;
115                     if(s_ccAttr[commandIndex].V)
116                         cc += (1 << 29);
117                     commandList->commandCodes[commandList->count] = cc;
118                     commandList->count++;
119                 }
120                 else
121                 {
122                     // If the return list is full but we still have command
123                     // available, report this and stop iterating
124                     more = YES;
125                     break;
126                 }
127             }
128         }
129         return more;
130     }
```

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.

```
131     void
132     CommandAuditGetDigest(
133         TPM2B_DIGEST    *digest         // OUT: command digest
134         )
135     {
136         TPM_CC                      commandCode;
137         COMMAND_INDEX               commandIndex;
138         HASH_STATE                  hashState;
139         // Start hash
140         digest->t.size = CryptHashStart(&hashState, gp.auditHashAlg);
141         // Add command code
142         for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
143         {
144             if(CommandAuditIsRequired(commandIndex))
145             {
146                 commandCode = GetCommandCode(commandIndex);
147                 CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
148             }
149         }
150         // Complete hash
151         CryptHashEnd2B(&hashState, &digest->b);
152         return;
153     }
```

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 "Tpm.h"
```

8.2.3 Functions

8.2.3.1 DAPreInstall_Init()

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

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

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

```
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
    NV_SYNC_PERSISTENT(failedTries);
    NV_SYNC_PERSISTENT(maxTries);
    NV_SYNC_PERSISTENT(recoveryTime);
    NV_SYNC_PERSISTENT(lockoutRecovery);
    NV_SYNC_PERSISTENT(lockOutAuthEnabled);
    return;
}
```

8.2.3.2 DAInit()

This function is called during _TPM_INIT() in order to make sure that the DA timers are reset when g_time is reset.

```
void
DAInit(
    void
    )
{
    s_selfHealTimer = g_time;
    s_lockoutTimer = g_time;
}
```


8.2.3.3 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.

```
29  void
30  DAStartup(
31      STARTUP_TYPE     type           // IN: startup type
32      )
33  {
34      NOT_REFERENCED(type);
35      // For any Startup(), if lockoutRecovery is 0, enable use of lockoutAuth.
36      if(gp.lockoutRecovery == 0)
37      {
38          gp.lockOutAuthEnabled = TRUE;
39          // Record the changes to NV
40          NV_SYNC_PERSISTENT(lockOutAuthEnabled);
41      }
42      // If DA has not been disabled and the previous shutdown is not orderly
43      // failedTries is not already at its maximum then increment 'failedTries'
44      if(gp.recoveryTime != 0
45         && gp.failedTries < gp.maxTries
46         && !IS_ORDERLY(g_prevOrderlyState))
47      {
48  #ifdef USE_DA_USED
49          gp.failedTries += g_daUsed;
50          g_daUsed = FALSE;
51  #else
52          gp.failedTries++;
53  #endif
54          // Record the change to NV
55          NV_SYNC_PERSISTENT(failedTries);
56      }
57      return;
58  }
```

8.2.3.4 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.

```
59  void
60  DARegisterFailure(
61      TPM_HANDLE       handle         // IN: handle for failure
62      )
63  {
64      // Reset the timer associated with lockout if the handle is the lockoutAuth.
65      if(handle == TPM_RH_LOCKOUT)
66          s_lockoutTimer = g_time;
67      else
68          s_selfHealTimer = g_time;
69      return;
70  }
```

8.2.3.5 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.

```
71  void
72  DASelfHeal(
73      void
74      )
75  {
76      // Regular authorization self healing logic
77      // If no failed authorization tries, do nothing.  Otherwise, try to
78      // decrease failedTries
79      if(gp.failedTries != 0)
80      {
81          // if recovery time is 0, DA logic has been disabled.  Clear failed tries
82          // immediately
83          if(gp.recoveryTime == 0)
84          {
85              gp.failedTries = 0;
86              // Update NV record
87              NV_SYNC_PERSISTENT(failedTries);
88          }
89          else
90          {
91              UINT64          decreaseCount;
92              // In the unlikely event that failedTries should become larger than
93              // maxTries
94              if(gp.failedTries > gp.maxTries)
95                  gp.failedTries = gp.maxTries;
96              // How much can failedTries be decreased
97              decreaseCount = ((g_time - s_selfHealTimer) / 1000) / gp.recoveryTime;
98              if(gp.failedTries <= (UINT32)decreaseCount)
99                  // should not set failedTries below zero
100                 gp.failedTries = 0;
101             else
102                 gp.failedTries -= (UINT32)decreaseCount;
103             // the cast prevents overflow of the product
104             s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
105             if(decreaseCount != 0)
106                 // If there was a change to the failedTries, record the changes
107                 // to NV
108                 NV_SYNC_PERSISTENT(failedTries);
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                 NV_SYNC_PERSISTENT(lockOutAuthEnabled);
126             }
127         }
128     }
129     return;
130 }
```


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 "Tpm.h"
```

8.3.3 Functions

8.3.3.1 HierarchyPreInstall()

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

```
void
HierarchyPreInstall_Init(
    void
    )
{
    // Allow lockout clear command
    gp.disableClear = FALSE;
    // Initialize Primary Seeds
    gp.EPSeed.t.size = PRIMARY_SEED_SIZE;
    CryptRandomGenerate(PRIMARY_SEED_SIZE, gp.EPSeed.t.buffer);
    gp.SPSeed.t.size = PRIMARY_SEED_SIZE;
    CryptRandomGenerate(PRIMARY_SEED_SIZE, gp.SPSeed.t.buffer);
    gp.PPSeed.t.size = PRIMARY_SEED_SIZE;
#ifdef USE_PLATFORM_EPS
    _plat__GetEPS(PRIMARY_SEED_SIZE, gp.EPSeed.t.buffer);
#else
    CryptRandomGenerate(PRIMARY_SEED_SIZE, gp.PPSeed.t.buffer);
#endif
    // Initialize owner, endorsement and lockout auth
    gp.ownerAuth.t.size = 0;
    gp.endorsementAuth.t.size = 0;
    gp.lockoutAuth.t.size = 0;
    // Initialize owner, endorsement, and lockout policy
    gp.ownerAlg = TPM_ALG_NULL;
    gp.ownerPolicy.t.size = 0;
    gp.endorsementAlg = TPM_ALG_NULL;
    gp.endorsementPolicy.t.size = 0;
    gp.lockoutAlg = TPM_ALG_NULL;
    gp.lockoutPolicy.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;
    CryptRandomGenerate(gp.phProof.t.size, gp.phProof.t.buffer);
    CryptRandomGenerate(gp.shProof.t.size, gp.shProof.t.buffer);
    CryptRandomGenerate(gp.ehProof.t.size, gp.ehProof.t.buffer);
    // Write hierarchy data to NV
    NV_SYNC_PERSISTENT(disableClear);
    NV_SYNC_PERSISTENT(EPSeed);
    NV_SYNC_PERSISTENT(SPSeed);
    NV_SYNC_PERSISTENT(PPSeed);
    NV_SYNC_PERSISTENT(ownerAuth);
```



```
    NV_SYNC_PERSISTENT(endorsementAuth);
    NV_SYNC_PERSISTENT(lockoutAuth);
    NV_SYNC_PERSISTENT(ownerAlg);
    NV_SYNC_PERSISTENT(ownerPolicy);
    NV_SYNC_PERSISTENT(endorsementAlg);
    NV_SYNC_PERSISTENT(endorsementPolicy);
    NV_SYNC_PERSISTENT(lockoutAlg);
    NV_SYNC_PERSISTENT(lockoutPolicy);
    NV_SYNC_PERSISTENT(phProof);
    NV_SYNC_PERSISTENT(shProof);
    NV_SYNC_PERSISTENT(ehProof);
    return;
}
```

8.3.3.2 HierarchyStartup()

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

```
void
HierarchyStartup(
    STARTUP_TYPE     type           // IN: start up type
    )
{
    // phEnable is SET on any startup
    g_phEnable = TRUE;
    // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
    // TPM_RESTART
    if(type != SU_RESUME)
    {
        gc.platformAuth.t.size = 0;
        gc.platformPolicy.t.size = 0;
        // enable the storage and endorsement hierarchies and the platformNV
        gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
    }
    // nullProof and nullSeed are updated at every TPM_RESET
    if((type != SU_RESTART) && (type != SU_RESUME))
    {
        gr.nullProof.t.size = PROOF_SIZE;
        CryptRandomGenerate(gr.nullProof.t.size,
                            gr.nullProof.t.buffer);
        gr.nullSeed.t.size = PRIMARY_SEED_SIZE;
        CryptRandomGenerate(PRIMARY_SEED_SIZE, gr.nullSeed.t.buffer);
    }
    return;
}
```

8.3.3.3 HierarchyGetProof()

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

```
TPM2B_AUTH *
HierarchyGetProof(
    TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy constant
    )
{
    TPM2B_AUTH          *auth = NULL;
    switch(hierarchy)
    {
        case TPM_RH_PLATFORM:
            // phProof for TPM_RH_PLATFORM
            auth = &gp.phProof;
            break;
        case TPM_RH_ENDORSEMENT:
```

```
97              // ehProof for TPM_RH_ENDORSEMENT
98              auth = &gp.ehProof;
99              break;
100         case TPM_RH_OWNER:
101             // shProof for TPM_RH_OWNER
102             auth = &gp.shProof;
103             break;
104         case TPM_RH_NULL:
105             // nullProof for TPM_RH_NULL
106             auth = &gr.nullProof;
107             break;
108         default:
109             FAIL(FATAL_ERROR_INTERNAL);
110             break;
111     }
112     return auth;
113 }
```

8.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

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

8.3.3.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE: The TPM_RH_NULL hierarchy is always enabled.

Return Value	Meaning
TRUE	hierarchy is enabled
FALSE	hierarchy is disabled

```
139 BOOL
140 HierarchyIsEnabled(
```

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

8.4 NVDynamic.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.

The entries in dynamic space are a linked list of entries. Each entry has, as its first field, a size. If the size field is zero, it marks the end of the list.

An Index allocation will contain an NV_INDEX structure. If the Index does not have the orderly attribute, the NV_INDEX is followed immediately by the NV data.

An evict object entry contains a handle followed by an OBJECT structure. This results in both the Index and Evict Object having an identifying handle as the first field following the size field.

When an Index has the orderly attribute, the data is kept in RAM. This RAM is saved to backing store in NV memory on any orderly shutdown. The entries in orderly memory are also a linked list using a size field as the first entry.

The attributes of an orderly index are maintained in RAM memory in order to reduce the number of NV writes needed for orderly data. When an orderly index is created, an entry is made in the dynamic NV memory space that holds the Index authorizations *(authPolicy* and *authValue*) and the size of the data. This entry is only modified if the *authValue* of the index is changed. The more volatile data of the index is kept in RAM. When an orderly Index is created or deleted, the RAM data is copied to NV backing store so that the image in the backing store matches the layout of RAM. In normal operation. The RAM data is also copied on any orderly shutdown. In normal operation, the only other reason for writing to the backing store for RAM is when a counter is first written (TPMA_NV_WRITTEN changes from CLEAR to SET) or when a counter "rolls over."

Static space contains items that are individually modifiable. The values are in the *gp* PERSISTEND_DATA structure in RAM and mapped to locations in NV.

8.4.2 Includes, Defines and Data Definitions

```
1   #define NV_C
2   #include "Tpm.h"
3   #include "PlatformData.h"
```

8.4.3 Local Functions

8.4.3.1 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_REF_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.

```
4   static NV_REF
5   NvNext(
6       NV_REF          *iter,          // IN/OUT: the list iterator
7       TPM_HANDLE      *handle         // OUT: the handle of the next item.
8       )
9   {
10      NV_REF              currentAddr;
11      NV_ENTRY_HEADER     header;
```

```
12      // If iterator is at the beginning of list
13      if(*iter == NV_REF_INIT)
14      {
15          // Initialize iterator
16          *iter = NV_USER_DYNAMIC;
17      }
18      // if we are going to return what the iter is currently pointing to...
19      currentAddr = *iter + sizeof(UINT32);
20      // If iterator reaches the end of NV space, then don't advance and return
21      // that we are at the end of the list. The end of the list occurs when
22      // we don't have space for a size and a handle
23  //    if(*iter + sizeof(UINT32) > s_evictNvEnd)
24  //        return 0;
25      // read the header of the next entry
26      NvRead(&header, *iter, sizeof(NV_ENTRY_HEADER));
27      // if the size field is zero, then we have hit the end of the list
28      if(header.size == 0)
29          // leave the *iter pointing at the end of the list
30          return 0;
31      // advance the header by the size of the entry
32      *iter += header.size;
33      if(handle != NULL)
34          *handle = header.handle;
35      return currentAddr;
36  }
```

8.4.3.2 NvNextByType()

This function returns a reference to the next NV entry of the desired type

Return Value	Meaning
0	end of list
!= 0	the next entry of the indicated type

```
37  static NV_REF
38  NvNextByType(
39      TPM_HANDLE      *handle,        // OUT: the handle of the found type
40      NV_REF          *iter,          // IN: the iterator
41      TPM_HT           type           // IN: the handle type to look for
42      )
43  {
44      NV_REF           addr;
45      TPM_HANDLE       nvHandle;
46      while((addr = NvNext(iter, &nvHandle)) != 0)
47      {
48          // addr: the address of the location containing the handle of the value
49          // iter: the next location.
50          if(HandleGetType(nvHandle) == type)
51              break;
52      }
53      if(handle != NULL)
54          *handle = nvHandle;
55      return addr;
56  }
```

8.4.3.3 NvNextIndex()

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

Return Value	Meaning
0	end of list
!= 0	the next

```
57  #define NvNextIndex(handle, iter)                   \
58      NvNextByType(handle, iter, TPM_HT_NV_INDEX)
```

8.4.3.4 NvNextEvict()

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

```
59  #define NvNextEvict(handle, iter)                   \
60      NvNextByType(handle, iter, TPM_HT_PERSISTENT)
```

8.4.3.5 NvGetEnd()

Function to find the end of the NV dynamic data list

```
61  static NV_REF
62  NvGetEnd(
63      void
64      )
65  {
66      NV_REF          iter = NV_REF_INIT;
67      NV_REF          currentAddr;
68      // Scan until the next address is 0
69      while((currentAddr = NvNext(&iter, NULL)) != 0);
70      return iter;
71  }
```

8.4.3.6 NvGetFreeBytes

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

```
72  static UINT32
73  NvGetFreeBytes(
74      void
75      )
76  {
77      return s_evictNvEnd - NvGetEnd();
78  }
```

8.4.3.7 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

```
79  static BOOL
80  NvTestSpace(
81      UINT32          size,           // IN: size of the entity to be added
82      BOOL            isIndex,        // IN: TRUE if the entity is an index
```



```
83      BOOL            isCounter       // IN: TRUE if the index is a counter
84      )
85  {
86      UINT32      remainBytes = NvGetFreeBytes();
87      UINT32      reserved = sizeof(UINT32)       // size of the forward pointer
88          + sizeof(NV_LIST_TERMINATOR);
89  // Do a compile time sanity check on the setting for NV_MEMORY_SIZE
90  #if NV_MEMORY_SIZE < 1024
91  #error "NV_MEMORY_SIZE probably isn't large enough"
92  #endif
93      // For NV Index, need to make sure that we do not allocate an Index if this
94      // would mean that the TPM cannot allocate the minimum number of evict
95      // objects.
96      if(isIndex)
97      {
98          // Get the number of persistent objects allocated
99          UINT32      persistentNum = NvCapGetPersistentNumber();
100         // If we have not allocated the requisite number of evict objects, then we
101         // need to reserve space for them.
102         // NOTE: some of this is not written as simply as it might seem because
103         // the values are all unsigned and subtracting needs to be done carefully
104         // so that an underflow doesn't cause problems.
105         if(persistentNum < MIN_EVICT_OBJECTS)
106             reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
107     }
108     // If this is not an index or is not a counter, reserve space for the
109     // required number of counter indices
110     if(!isIndex || !isCounter)
111     {
112         // Get the number of counters
113         UINT32      counterNum = NvCapGetCounterNumber();
114         // If the required number of counters have not been allocated, reserved
115         // space for the extra needed counters
116         if(counterNum < MIN_COUNTER_INDICES)
117             reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
118     }
119     // Check that the requested allocation will fit after making sure that there
120     // will be no chance of overflow
121     return ((reserved < remainBytes)
122             && (size <= remainBytes)
123             && (size + reserved <= remainBytes));
124 }
```

8.4.3.8 NvWriteNvListEnd()

Function to write the list terminator.

```
125 NV_REF
126 NvWriteNvListEnd(
127     NV_REF          end
128     )
129 {
130     BYTE        listEndMarker[sizeof(NV_LIST_TERMINATOR)] = {0};
131     UINT64      maxCount = NvReadMaxCount();
132     // This is a constant check that can be resolved at compile time.
133     cAssert(sizeof(UINT64) <= sizeof(NV_LIST_TERMINATOR) - sizeof(UINT32));
134     MemoryCopy(&listEndMarker[sizeof(UINT32)], &maxCount, sizeof(UINT64));
135     pAssert(end + sizeof(NV_LIST_TERMINATOR) <= s_evictNvEnd);
136     NvWrite(end, sizeof(NV_LIST_TERMINATOR), &listEndMarker);
137     return end + sizeof(NV_LIST_TERMINATOR);
138 }
```


8.4.3.9 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).

The *totalSize* will be the size of *entity*. If a handle is added, this function will increase the size accordingly.

```
139  static TPM_RC
140  NvAdd(
141      UINT32           totalSize,     // IN: total size needed for this entity For
142                                      //     evict object, totalSize is the same as
143                                      //     bufferSize.  For NV Index, totalSize is
144                                      //     bufferSize plus index data size
145      UINT32           bufferSize,    // IN: size of initial buffer
146      TPM_HANDLE       handle,        // IN: optional handle
147      BYTE            *entity         // IN: initial buffer
148      )
149  {
150      NV_REF          newAddr;        // IN: where the new entity will start
151      NV_REF          nextAddr;
152      RETURN_IF_NV_IS_NOT_AVAILABLE;
153      // Get the end of data list
154      newAddr = NvGetEnd();
155      // Step over the forward pointer
156      nextAddr = newAddr + sizeof(UINT32);
157      // Optionally write the handle. For indices, the handle is TPM_RH_UNASSIGNED
158      // so that the handle in the nvIndex is used instead of writing this value
159      if(handle != TPM_RH_UNASSIGNED)
160      {
161          NvWrite((UINT32)nextAddr, sizeof(TPM_HANDLE), &handle);
162          nextAddr += sizeof(TPM_HANDLE);
163      }
164      // Write entity data
165      NvWrite((UINT32)nextAddr, bufferSize, entity);
166      // Advance the pointer by the amount of the total
167      nextAddr += totalSize;
168      // Finish by writing the link value
169      // Write the next offset (relative addressing)
170      totalSize = nextAddr - newAddr;
171      // Write link value
172      NvWrite((UINT32)newAddr, sizeof(UINT32), &totalSize);
173      // Write the list terminator
174      NvWriteNvListEnd(nextAddr);
175      return TPM_RC_SUCCESS;
176  }
```

8.4.3.10 NvDelete()

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

```
177  static TPM_RC
178  NvDelete(
179      NV_REF           entityRef      // IN: reference to entity to be deleted
180      )
181  {
182      UINT32          entrySize;
183      // adjust entityAddr to back up and point to the forward pointer
184      NV_REF          entryRef = entityRef - sizeof(UINT32);
185      NV_REF          endRef = NvGetEnd();
186      NV_REF          nextAddr; // address of the next entry
187      RETURN_IF_NV_IS_NOT_AVAILABLE;
188      // Get the offset of the next entry. That is, back up and point to the size
```

```
189         // field of the entry
190         NvRead(&entrySize, entryRef, sizeof(UINT32));
191         // The next entry after the one being deleted is at a relative offset
192         // from the current entry
193         nextAddr = entryRef + entrySize;
194         // If this is not the last entry, move everything up
195         if(nextAddr < endRef)
196         {
197             pAssert(nextAddr > entryRef);
198             _plat__NvMemoryMove(nextAddr,
199                                 entryRef,
200                                 (endRef - nextAddr));
201         }
202         // The end of the used space is now moved up by the amount of space we just
203         // reclaimed
204         endRef -= entrySize;
205         // Write the end marker, and make the new end equal to the first byte after
206         // the just added end value. This will automatically update the NV value for
207         // maxCounte
208         endRef = NvWriteNvListEnd(endRef);
209         // Clear the reclaimed memory
210         _plat__NvMemoryClear(endRef, entrySize);
211         return TPM_RC_SUCCESS;
212     }
```

8.4.4 RAM-based NV Index Data Access Functions

8.4.4.1 Introduction

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

NV storage associated with orderly data is updated when a NV Index is added but NOT when the data or attributes are changed. Orderly data is only updated to NV on an orderly shutdown (TPM2_Shutdown())

```
213     static NV_RAM_REF
214     NvRamNext(
215         NV_RAM_REF      *iter,          // IN/OUT: the list iterator
216         TPM_HANDLE      *handle         // OUT: the handle of the next item.
217         )
218     {
219         NV_RAM_REF          currentAddr;
220         NV_RAM_HEADER       header;
221         // If iterator is at the beginning of list
222         if(*iter == NV_RAM_REF_INIT)
223         {
224             // Initialize iterator
225             *iter = &s_indexOrderlyRam[0];
226         }
227         // if we are going to return what the iter is currently pointing to...
228         currentAddr = *iter;
229         // If iterator reaches the end of NV space, then don't advance and return
230         // that we are at the end of the list. The end of the list occurs when
231         // we don't have space for a size and a handle
232         if(*iter + sizeof(NV_RAM_HEADER) >= RAM_ORDERLY_END)
233             return NULL;
234         // read the header of the next entry
235         MemoryCopy(&header, *iter, sizeof(NV_RAM_HEADER));
236         // if the size field is zero, then we have hit the end of the list
237         if(header.size == 0)
238             // leave the *iter pointing at the end of the list
239             return 0;
240         // advance the header by the size of the entry
```

```
241         *iter += header.size;
242         pAssert(*iter <= RAM_ORDERLY_END);
243         if(handle != NULL)
244             *handle = header.handle;
245         return currentAddr;
246     }
```

8.4.4.2 NvRamGetEnd()

This routine performs the same function as NvGetEnd() but for the RAM data.

```
247     static NV_RAM_REF
248     NvRamGetEnd(
249         void
250         )
251     {
252         NV_RAM_REF           iter = NV_RAM_REF_INIT;
253         NV_RAM_REF           currentAddr;
254         // Scan until the next address is 0
255         while((currentAddr = NvRamNext(&iter, NULL)) != 0);
256         return iter;
257     }
```

8.4.4.3 NvRamTestSpaceIndex()

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

```
258     static BOOL
259     NvRamTestSpaceIndex(
260         UINT32           size           // IN: size of the data to be added to RAM
261         )
262     {
263         UINT32          remaining = RAM_ORDERLY_END - NvRamGetEnd();
264         UINT32          needed = sizeof(NV_RAM_HEADER) + size;
265         // NvRamGetEnd points to the next available byte.
266         return remaining >= needed;
267     }
```

8.4.4.4 NvRamGetIndex()

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.

```
268     static NV_RAM_REF
269     NvRamGetIndex(
270         TPMI_RH_NV_INDEX     handle         // IN: NV handle
271         )
272     {
273         NV_RAM_REF          iter = NV_RAM_REF_INIT;
274         NV_RAM_REF          currentAddr;
275         TPM_HANDLE          foundHandle;
276         while((currentAddr = NvRamNext(&iter, &foundHandle)) != 0)
277         {
278             if(handle == foundHandle)
279                 break;
```

```
280         }
281         pAssert(ORDERLY_RAM_ADDRESS_OK(currentAddr, 0));
282         return currentAddr;
283     }
```

8.4.4.5 NvUpdateIndexOrderlyData()

This function is used to cause an update of the orderly data to the NV backing store.

```
284     void
285     NvUpdateIndexOrderlyData(
286         void
287         )
288     {
289         // Write reserved RAM space to NV
290         NvWrite(NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
291     }
```

8.4.4.6 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.

This function should be called after the NV Index space has been updated and the index removed. This insures that NV is available so that checking for NV availability is not required during this function.

```
292     static void
293     NvAddRAM(
294         TPMS_NV_PUBLIC  *index          // IN: the index descriptor
295         )
296     {
297         NV_RAM_HEADER       header;
298         NV_RAM_REF          end = NvRamGetEnd();
299         header.size = sizeof(NV_RAM_HEADER) + index->dataSize;
300         header.handle = index->nvIndex;
301         MemoryCopy(&header.attributes, &index->attributes, sizeof(TPMA_NV));
302         pAssert(ORDERLY_RAM_ADDRESS_OK(end, header.size));
303         // Copy the header to the memory
304         MemoryCopy(end, &header, sizeof(NV_RAM_HEADER));
305         // Clear the data area (just in case)
306         MemorySet(end + sizeof(NV_RAM_HEADER), 0, index->dataSize);
307         // Step over this new entry
308         end += header.size;
309         // If the end marker will fit, add it
310         if(end + sizeof(NV_RAM_HEADER) < RAM_ORDERLY_END)
311             MemorySet(end, 0, sizeof(NV_RAM_HEADER));
312         // Write reserved RAM space to NV to reflect the newly added NV Index
313         SET_NV_UPDATE(UT_ORDERLY);
314         return;
315     }
```

8.4.4.7 NvDeleteRAM()

This function is used to delete a RAM-backed NV Index data area. The space used by the entry are overwritten by the contents of the Index data that comes after (the data is moved up to fill the hole left by removing this index. The reclaimed space is cleared to zeros. This function assumes the data of NV Index exists in RAM.

This function should be called after the NV Index space has been updated and the index removed. This insures that NV is available so that checking for NV availability is not required during this function.

```
316 static void
317 NvDeleteRAM(
318     TPMI_RH_NV_INDEX     handle         // IN: NV handle
319     )
320 {
321     NV_RAM_REF           nodeAddress;
322     NV_RAM_REF           nextNode;
323     UINT32               size;
324     NV_RAM_REF           lastUsed = NvRamGetEnd();
325     nodeAddress = NvRamGetIndex(handle);
326     pAssert(nodeAddress != 0);
327     // Get node size
328     MemoryCopy(&size, nodeAddress, sizeof(size));
329     // Get the offset of next node
330     nextNode = nodeAddress + size;
331     // Copy the data
332     MemoryCopy(nodeAddress, nextNode, lastUsed - nextNode);
333     // Clear out the reclaimed space
334     MemorySet(lastUsed - size, 0, size);
335     // Write reserved RAM space to NV to reflect the newly delete NV Index
336     SET_NV_UPDATE(UT_ORDERLY);
337     return;
338 }
```

8.4.4.8 NvReadIndex()

This function is used to read the NV Index NV_INDEX. This is used so that the index information can be compressed and only this function would be needed to decompress it. Mostly, compression would only be able to save the space needed by the policy.

```
339 void
340 NvReadNvIndexInfo(
341     NV_REF           ref,           // IN: points to NV where index is located
342     NV_INDEX        *nvIndex        // OUT: place to receive index data
343     )
344 {
345     pAssert(nvIndex != NULL);
346     NvRead(nvIndex, ref, sizeof(NV_INDEX));
347 }
```

8.4.4.9 NvReadObject()

This function is used to read a persistent object. This is used so that the object information can be compressed and only this function would be needed to uncompress it.

```
348 void
349 NvReadObject(
350     NV_REF           ref,           // IN: points to NV where index is located
351     OBJECT          *object         // OUT: place to receive the object data
352     )
353 {
354     NvRead(object, (ref + sizeof(TPM_HANDLE)), sizeof(OBJECT));
355 }
```

8.4.4.10 NvFindEvict()

This function will return the NV offset of an evict object

Return Value	Meaning
0	evict object not found
!= 0	offset of evict object

```
356 static NV_REF
357 NvFindEvict(
358     TPM_HANDLE       nvHandle,
359     OBJECT          *object
360     )
361 {
362     NV_REF          found = NvFindHandle(nvHandle);
363     // If we found the handle and the request included an object pointer, fill it in
364     if(found != 0 && object != NULL)
365         NvReadObject(found, object);
366     return found;
367 }
```

8.4.4.11 NvIndexIsDefined()

See if an index is already defined

```
368 BOOL
369 NvIndexIsDefined(
370     TPM_HANDLE       nvHandle       // IN: Index to look for
371     )
372 {
373     return (NvFindHandle(nvHandle) != 0);
374 }
```

8.4.4.12 NvConditionallyWrite()

Function to check if the data to be written has changed and write it if it has

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

```
375 static TPM_RC
376 NvConditionallyWrite(
377     NV_REF           entryAddr,     // IN: stating address
378     UINT32           size,          // IN: size of the data to write
379     void            *data           // IN: the data to write
380     )
381 {
382     // If the index data is actually changed, then a write to NV is required
383     if(_plat__NvIsDifferent(entryAddr, size, data))
384     {
385         // Write the data if NV is available
386         if(g_NvStatus == TPM_RC_SUCCESS)
387         {
388             NvWrite(entryAddr, size, data);
389             // NV needs an update       //??
390             SET_NV_UPDATE(UT_NV);       //??
391         }
392         return g_NvStatus;
393     }
394     return TPM_RC_SUCCESS;
395 }
```


8.4.4.13 NvReadNvIndexAttributes()

This function returns the attributes of an NV Index.

```
396  static TPMA_NV
397  NvReadNvIndexAttributes(
398      NV_REF           locator        // IN: reference to an NV index
399      )
400  {
401      TPMA_NV                 attributes;
402      NvRead(&attributes,
403             locator + offsetof(NV_INDEX, publicArea.attributes),
404             sizeof(TPMA_NV));
405      return attributes;
406  }
```

8.4.4.14 NvReadRamIndexAttributes()

This function returns the attributes from the RAM header structure. This function is used to deal with the fact that the header structure is only byte aligned.

```
407  static TPMA_NV
408  NvReadRamIndexAttributes(
409      NV_RAM_REF       ref            // IN: pointer to a NV_RAM_HEADER
410      )
411  {
412      TPMA_NV         attributes;
413      MemoryCopy(&attributes, ref + offsetof(NV_RAM_HEADER, attributes),
414                 sizeof(TPMA_NV));
415      return attributes;
416  }
```

8.4.4.15 NvWriteNvIndexAttributes()

This function is used to write just the attributes of an index to NV.

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

```
417  static TPM_RC
418  NvWriteNvIndexAttributes(
419      NV_REF           locator,       // IN: location of the index
420      TPMA_NV          attributes     // IN: attributes to write
421      )
422  {
423      return NvConditionallyWrite(
424          locator + offsetof(NV_INDEX, publicArea.attributes),
425          sizeof(TPMA_NV),
426          &attributes);
427  }
```

8.4.4.16 NvWriteRamIndexAttributes()

This function is used to write the index attributes into an unaligned structure

```
428  static void
429  NvWriteRamIndexAttributes(
```



```
430         NV_RAM_REF        ref,            // IN: address of the header
431         TPMA_NV           attributes      // IN: the attributes to write
432         )
433     {
434         MemoryCopy(ref + offsetof(NV_RAM_HEADER, attributes), &attributes,
435                    sizeof(TPMA_NV));
436     }
```

8.4.5 Externally Accessible Functions

8.4.5.1 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

```
437     BOOL
438     NvIsPlatformPersistentHandle(
439         TPM_HANDLE        handle          // IN: handle
440         )
441     {
442         return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
443     }
```

8.4.5.2 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

```
444     BOOL
445     NvIsOwnerPersistentHandle(
446         TPM_HANDLE        handle          // IN: handle
447         )
448     {
449         return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
450     }
```

8.4.5.3 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 *platformAuth*
TPM_RC_NV_READLOCKED	Index is present but locked for reading and command does not write to the index
TPM_RC_NV_WRITELOCKED	Index is present but locked for writing and command writes to the index

```
451  TPM_RC
452  NvIndexIsAccessible(
453      TPMI_RH_NV_INDEX     handle        // IN: handle
454      )
455  {
456      NV_INDEX            *nvIndex = NvGetIndexInfo(handle, NULL);
457      if(nvIndex == NULL)
458      // If index is not found, return TPM_RC_HANDLE
459          return TPM_RC_HANDLE;
460      if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
461      {
462          // if shEnable is CLEAR, an ownerCreate NV Index should not be
463          // indicated as present
464          if(!IsNv_TPMA_NV_PLATFORMCREATE(nvIndex->publicArea.attributes))
465          {
466              if(gc.shEnable == FALSE)
467                  return TPM_RC_HANDLE;
468          }
469          // if phEnableNV is CLEAR, a platform created Index should not
470          // be visible
471          else if(gc.phEnableNV == FALSE)
472              return TPM_RC_HANDLE;
473      }
474  #if 0 // Writelock test
475      // If the Index is write locked and this is an NV Write operation...
476      if(IsNv_TPMA_NV_WRITELOCKED(nvIndex->publicArea.attributes)
477         &&  IsWriteOperation(commandIndex))
478      {
479          // then return a locked indication unless the command is TPM2_NV_WriteLock
480          if(GetCommandCode(commandIndex) != TPM_CC_NV_WriteLock)
481              return TPM_RC_NV_LOCKED;
482          return TPM_RC_SUCCESS;
483      }
484  #endif
485  #if 0   // Readlock Test
486      // If the Index is read locked and this is an NV Read operation...
487      if(IsNv_TPMA_NV_READLOCKED(nvIndex->publicArea.attributes)
488         && IsReadOperation(commandIndex))
489      {
490          // then return a locked indication unless the command is TPM2_NV_ReadLock
491          if(GetCommandCode(commandIndex) != TPM_CC_NV_ReadLock)
492              return TPM_RC_NV_LOCKED;
493      }
494  #endif
495      // NV Index is accessible
496      return TPM_RC_SUCCESS;
497  }
```

8.4.5.4 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

```
498  TPM_RC
499  NvGetEvictObject(
500      TPM_HANDLE       handle,        // IN: handle
501      OBJECT          *object         // OUT: object data
502      )
503  {
504      NV_REF          entityAddr;         // offset points to the entity
505      // Find the address of evict object and copy to object
506      entityAddr = NvFindEvict(handle, object);
507      // whether there is an error or not, make sure that the evict
508      // status of the object is set so that the slot will get freed on exit
509      // Must do this after NvFindEvict loads the object
510      object->attributes.evict = SET;
511      // If handle is not found, return an error
512      if(entityAddr == 0)
513          return TPM_RC_HANDLE;
514      return TPM_RC_SUCCESS;
515  }
```

8.4.5.5 NvIndexCacheInit()

Function to initialize the Index cache

```
516  void
517  NvIndexCacheInit(
518      void
519      )
520  {
521      s_cachedNvRef = NV_REF_INIT;
522      s_cachedNvRamRef = NV_RAM_REF_INIT;
523      s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
524  }
```

8.4.5.6 NvGetIndexData()

This function is used to access the data in an NV Index. The data is returned as a byte sequence.

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.

```
525  void
526  NvGetIndexData(
527      NV_INDEX        *nvIndex,       // IN: the in RAM index descriptor
528      NV_REF           locator,       // IN: where the data is located
529      UINT32           offset,        // IN: offset of NV data
530      UINT16           size,          // IN: size of NV data
531      void            *data           // OUT: data buffer
532      )
533  {
534      TPMA_NV             nvAttributes;
535      pAssert(nvIndex != NULL);
536      nvAttributes = nvIndex->publicArea.attributes;
537      pAssert(nvAttributes.TPMA_NV_WRITTEN == SET);
538      if(nvAttributes.TPMA_NV_ORDERLY == SET)
539      {
540          // Get data from RAM buffer
541          NV_RAM_REF           ramAddr = NvRamGetIndex(nvIndex->publicArea.nvIndex);
542          pAssert(ramAddr != 0 && (size <=
```

```
543                 ((NV_RAM_HEADER *)ramAddr)->size - sizeof(NV_RAM_HEADER) - offset));
544             MemoryCopy(data, ramAddr + sizeof(NV_RAM_HEADER) + offset, size);
545         }
546         else
547         {
548             // Validate that read falls within range of the index
549             pAssert(offset <= nvIndex->publicArea.dataSize
550                     &&  size <= (nvIndex->publicArea.dataSize - offset));
551             NvRead(data, locator + sizeof(NV_INDEX) + offset, size);
552         }
553         return;
554     }
```

8.4.5.7 NvGetUINT64Data()

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.

```
555     UINT64
556     NvGetUINT64Data(
557         NV_INDEX        *nvIndex,       // IN: the in RAM index descriptor
558         NV_REF           locator        // IN: where index exists in NV
559         )
560     {
561         UINT64                  intVal;
562         // Read the value and convert it to internal format
563         NvGetIndexData(nvIndex, locator, 0, 8, &intVal);
564         return BYTE_ARRAY_TO_UINT64(((BYTE *)&intVal));
565     }
```

8.4.5.8 NvWriteIndexAttributes()

This function is used to write just the attributes of an index.

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

```
566     TPM_RC
567     NvWriteIndexAttributes(
568         TPM_HANDLE       handle,
569         NV_REF           locator,       // IN: location of the index
570         TPMA_NV          attributes     // IN: attributes to write
571         )
572     {
573         TPM_RC              result;
574         if(IsNv_TPMA_NV_ORDERLY(attributes))
575         {
576             NV_RAM_REF      ram = NvRamGetIndex(handle);
577             NvWriteRamIndexAttributes(ram, attributes);
578             result = TPM_RC_SUCCESS;
579         }
580         else
581         {
582             result = NvWriteNvIndexAttributes(locator, attributes);
583         }
584         return result;
585     }
```


8.4.5.9 NvWriteIndexAuth()

This function is used to write the *authValue* of an index. It is used by TPM2_NV_ChangeAuth()

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

```
586 TPM_RC
587 NvWriteIndexAuth(
588     NV_REF           locator,       // IN: location of the index
589     TPM2B_AUTH      *authValue      // IN: the authValue to write
590     )
591 {
592     TPM_RC              result;
593     if(locator == s_cachedNvRef)
594     {
595         MemoryCopy2B(&s_cachedNvIndex.authValue.b, &authValue->b,
596                      sizeof(s_cachedNvIndex.authValue.t.buffer));
597     }
598     result = NvConditionallyWrite(
599         locator + offsetof(NV_INDEX, authValue),
600         sizeof(UINT16) + authValue->t.size,
601         authValue);
602     return result;
603 }
```

8.4.5.10 NvGetIndexInfo()

This function loads the *nvIndex* Info into the NV cache and returns a pointer to the NV_INDEX. If the returned value is zero, the index was not found. The *locator* parameter, if not NULL, will be set to the offset in NV of the Index (the location of the handle of the Index).

This function will set the index cache. If the index is orderly, the attributes from RAM are substituted for the attributes in the cached index

```
604 NV_INDEX *
605 NvGetIndexInfo(
606     TPM_HANDLE       nvHandle,      // IN: the index handle
607     NV_REF          *locator        // OUT: location of the index
608     )
609 {
610     if(s_cachedNvIndex.publicArea.nvIndex != nvHandle)
611     {
612         s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
613         s_cachedNvRamRef = 0;
614         s_cachedNvRef = NvFindHandle(nvHandle);
615         if(s_cachedNvRef == 0)
616             return NULL;
617         NvReadNvIndexInfo(s_cachedNvRef, &s_cachedNvIndex);
618         if(IsNv_TPMA_NV_ORDERLY(s_cachedNvIndex.publicArea.attributes))
619         {
620             s_cachedNvRamRef = NvRamGetIndex(nvHandle);
621             s_cachedNvIndex.publicArea.attributes =
622                 NvReadRamIndexAttributes(s_cachedNvRamRef);
623         }
624     }
625     if(locator != NULL)
626         *locator = s_cachedNvRef;
627     return &s_cachedNvIndex;
628 }
```

8.4.5.11 NvWriteIndexData()

This function is used to write NV index data. It is intended to be used to update the data associated with the default index.

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

Index data is only written due to a command that modifies the data in a single index. There is no case where changes are made to multiple indices data at the same time. Multiple attributes may be change but not multiple index data. This is important because we will normally be handling the index for which we have the cached pointer values.

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

```
629 TPM_RC
630 NvWriteIndexData(
631     NV_INDEX        *nvIndex,       // IN: the description of the index
632     UINT32           offset,        // IN: offset of NV data
633     UINT32           size,          // IN: size of NV data
634     void            *data           // IN: data buffer
635     )
636 {
637     TPM_RC               result = TPM_RC_SUCCESS;
638     pAssert(nvIndex != NULL);
639     // Make sure that this is dealing with the 'default' index.
640     // Note: it is tempting to change the calling sequence so that the 'default' is
641     // presumed.
642     pAssert(nvIndex->publicArea.nvIndex == s_cachedNvIndex.publicArea.nvIndex);
643     // Validate that write falls within range of the index
644     pAssert(offset <= nvIndex->publicArea.dataSize
645             &&  size <= (nvIndex->publicArea.dataSize - offset));
646     // Update TPMA_NV_WRITTEN bit if necessary
647     if(!IsNv_TPMA_NV_WRITTEN(nvIndex->publicArea.attributes))
648     {
649         // Update the in memory version of the attributes
650         nvIndex->publicArea.attributes.TPMA_NV_WRITTEN = SET;
651         // If this is not orderly, then update the NV version of
652         // the attributes
653         if(!IsNv_TPMA_NV_ORDERLY(nvIndex->publicArea.attributes))
654         {
655             result = NvWriteNvIndexAttributes(s_cachedNvRef,
656                                               nvIndex->publicArea.attributes);
657             if(result != TPM_RC_SUCCESS)
658                 return result;
659             // If this is a partial write of an ordinary index, clear the whole
660             // index.
661             if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes)
662                && (nvIndex->publicArea.dataSize > size))
663                 _plat__NvMemoryClear(s_cachedNvRef + sizeof(NV_INDEX),
664                                      nvIndex->publicArea.dataSize);
665         }
666         else
667         {
668             // This is orderly so update the RAM version
669             MemoryCopy(s_cachedNvRamRef + offsetof(NV_RAM_HEADER, attributes),
670                        &nvIndex->publicArea.attributes, sizeof(TPMA_NV));
671             // If setting WRITTEN for an orderly counter, make sure that the
672             // state saved version of the counter is saved
673             if(IsNvCounterIndex(nvIndex->publicArea.attributes))
674                 SET_NV_UPDATE(UT_ORDERLY);
```

```
675             // If setting the written attribute on an ordinary index, make sure that
676             // the data is all cleared out in case there is a partial write. This
677             // is only necessary for ordinary indices because all of the other types
678             // are always written in total.
679             else if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes))
680                 MemorySet(s_cachedNvRamRef + sizeof(NV_RAM_HEADER),
681                           0, nvIndex->publicArea.dataSize);
682         }
683     }
684     // If this is orderly data, write it to RAM
685     if(IsNv_TPMA_NV_ORDERLY(nvIndex->publicArea.attributes))
686     {
687         // Note: if this is the first write to a counter, the code above will queue
688         // the write to NV of the RAM data in order to update TPMA_NV_WRITTEN. In
689         // process of doing that write, it will also write the initial counter value
690         // Update RAM
691         MemoryCopy(s_cachedNvRamRef + sizeof(NV_RAM_HEADER) + offset, data, size);
692         // And indicate that the TPM is no longer orderly
693         g_clearOrderly = TRUE;
694     }
695     else
696     {
697         // Offset into the index to the first byte of the data to be written to NV
698         result = NvConditionallyWrite(s_cachedNvRef + sizeof(NV_INDEX) + offset,
699                                       size, data);
700     }
701     return result;
702 }
```

8.4.5.12 NvWriteUINT64Data()

This function to write back a UINT64 value. The various UINT64 values (bits, counters, and PINs()) are kept in canonical format but manipulate in native format. This takes a native format value converts it and saves it back as in canonical format.

This function will return the value from NV or RAM depending on the type of the index (orderly or not)

```
703 TPM_RC
704 NvWriteUINT64Data(
705     NV_INDEX        *nvIndex,       // IN: the description of the index
706     UINT64           intValue       // IN: the value to write
707     )
708 {
709     BYTE            bytes[8];
710     UINT64_TO_BYTE_ARRAY(intValue, bytes);
711     return NvWriteIndexData(nvIndex, 0, 8, &bytes);
712 }
```

8.4.5.13 NvGetIndexName()

This function computes the Name of an 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.

```
713 TPM2B_NAME *
714 NvGetIndexName(
715     NV_INDEX        *nvIndex,       // IN: the index over which the name is to be
716                                     //     computed
717     TPM2B_NAME      *name           // OUT: name of the index
718     )
719 {
720     UINT16              dataSize, digestSize;
```

```
721     BYTE                marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
722     BYTE               *buffer;
723     HASH_STATE          hashState;
724     // Marshal public area
725     buffer = marshalBuffer;
726     dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex->publicArea, &buffer, NULL);
727     // hash public area
728     digestSize = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
729     CryptDigestUpdate(&hashState, dataSize, marshalBuffer);
730     // Complete digest leaving room for the nameAlg
731     CryptHashEnd(&hashState, digestSize, &name->b.buffer[2]);
732     // Include the nameAlg
733     UINT16_TO_BYTE_ARRAY(nvIndex->publicArea.nameAlg, name->b.buffer);
734     name->t.size = digestSize + 2;
735     return name;
736 }
```

8.4.5.14 NvGetNameByIndexHandle()

This function is used to compute the Name of an NV Index referenced by handle.

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.

```
737 TPM2B_NAME *
738 NvGetNameByIndexHandle(
739     TPMI_RH_NV_INDEX     handle,        // IN: handle of the index
740     TPM2B_NAME          *name           // OUT: name of the index
741     )
742 {
743     NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
744     return NvGetIndexName(nvIndex, name);
745 }
```

8.4.5.15 NvDefineIndex()

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

Error Returns	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```
746 TPM_RC
747 NvDefineIndex(
748     TPMS_NV_PUBLIC  *publicArea,    // IN: A template for an area to create.
749     TPM2B_AUTH      *authValue      // IN: The initial authorization value
750     )
751 {
752     // The buffer to be written to NV memory
753     NV_INDEX        nvIndex;            // the index data
754     UINT16          entrySize;          // size of entry
755     TPM_RC          result;
756     entrySize = sizeof(NV_INDEX);
757     // only allocate data space for Indices that are going to be written to NV.
758     // Orderly indices don't need space.
759     if(!IsNv_TPMA_NV_ORDERLY(publicArea->attributes))
760         entrySize += publicArea->dataSize;
761     // Check if we have enough space to create the NV Index
762     // In this implementation, the only resource limitation is the available NV
763     // space (and possibly RAM space.)  Other implementation may have other
764     // limitation on counter or on NV slots
```

```
765         if(!NvTestSpace(entrySize, TRUE, IsNvCounterIndex(publicArea->attributes)))
766             return TPM_RC_NV_SPACE;
767         // if the index to be defined is RAM backed, check RAM space availability
768         // as well
769         if(IsNv_TPMA_NV_ORDERLY(publicArea->attributes)
770            &&  !NvRamTestSpaceIndex(publicArea->dataSize))
771             return TPM_RC_NV_SPACE;
772         // Copy input value to nvBuffer
773         nvIndex.publicArea = *publicArea;
774         // Copy the authValue
775         nvIndex.authValue = *authValue;
776         // Add index to NV memory
777         result = NvAdd(entrySize, sizeof(NV_INDEX), TPM_RH_UNASSIGNED, (BYTE *)&nvIndex);
778         if(result == TPM_RC_SUCCESS)
779         {
780         // If the data of NV Index is RAM backed, add the data area in RAM as well
781             if(IsNv_TPMA_NV_ORDERLY(publicArea->attributes))
782                 NvAddRAM(publicArea);
783         }
784         return result;
785     }
```

8.4.5.16 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

```
786     TPM_RC
787     NvAddEvictObject(
788         TPMI_DH_OBJECT   evictHandle,   // IN: new evict handle
789         OBJECT          *object         // IN: object to be added
790         )
791     {
792         TPM_HANDLE       temp = object->evictHandle;
793         TPM_RC           result;
794         // Check if we have enough space to add the evict object
795         // An evict object needs 8 bytes in index table + sizeof OBJECT
796         // In this implementation, the only resource limitation is the available NV
797         // space.  Other implementation may have other limitation on evict object
798         // handle space
799         if(!NvTestSpace(sizeof(OBJECT) + sizeof(TPM_HANDLE), FALSE, FALSE))
800             return TPM_RC_NV_SPACE;
801         // Set evict attribute and handle
802         object->attributes.evict = SET;
803         object->evictHandle = evictHandle;
804         // Now put this in NV
805         result = NvAdd(sizeof(OBJECT), sizeof(OBJECT), evictHandle, (BYTE *)object);
806         // Put things back the way they were
807         object->attributes.evict = CLEAR;
808         object->evictHandle = temp;
809         return result;
810     }
```

8.4.5.17 NvDeleteIndex()

This function is used to delete an NV Index.

Error Returns	Meaning
TPM_RC_NV_UNAVAILABLE	NV is not accessible
TPM_RC_NV_RATE	NV is rate limiting

```
811  TPM_RC
812  NvDeleteIndex(
813      NV_INDEX        *nvIndex,       // IN: an in RAM index descriptor
814      NV_REF           entityAddr     // IN: location in NV
815      )
816  {
817      TPM_RC           result;
818      if(nvIndex != NULL)
819      {
820          // Whenever a counter is deleted, make sure that the MaxCounter value is
821          // updated to reflect the value
822          if(IsNvCounterIndex(nvIndex->publicArea.attributes)
823             && IsNv_TPMA_NV_WRITTEN(nvIndex->publicArea.attributes))
824              NvUpdateMaxCount(NvGetUINT64Data(nvIndex, entityAddr));
825          result = NvDelete(entityAddr);
826          if(result != TPM_RC_SUCCESS)
827              return result;
828          // If the NV Index is RAM back, delete the RAM data as well
829          if(IsNv_TPMA_NV_ORDERLY(nvIndex->publicArea.attributes))
830              NvDeleteRAM(nvIndex->publicArea.nvIndex);
831          NvIndexCacheInit();
832      }
833      return TPM_RC_SUCCESS;
834  }
```

8.4.5.18 NvDeleteEvict()

This function will delete a NV evict object. Will return success if object deleted or if it does not exist

```
835  TPM_RC
836  NvDeleteEvict(
837      TPM_HANDLE       handle         // IN: handle of entity to be deleted
838      )
839  {
840      NV_REF      entityAddr = NvFindEvict(handle, NULL);     // pointer to entity
841      TPM_RC      result = TPM_RC_SUCCESS;
842      if(entityAddr != 0)
843          result = NvDelete(entityAddr);
844      return result;
845  }
```

8.4.5.19 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 defined using *ownerAuth*.

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

```
846  TPM_RC
847  NvFlushHierarchy(
848      TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flushed.
849      )
```

```
{
    NV_REF           iter = NV_REF_INIT;
    NV_REF           currentAddr;
    TPM_HANDLE       entityHandle;
    TPM_RC           result = TPM_RC_SUCCESS;
    while((currentAddr = NvNext(&iter, &entityHandle)) != 0)
    {
        if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
        {
            NV_INDEX        nvIndex;
            // If flush endorsement or platform hierarchy, no NV Index would be
            // flushed
            if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
                continue;
            // Get the index information
            NvReadNvIndexInfo(currentAddr, &nvIndex);
            // For storage hierarchy, flush OwnerCreated index
            if(!IsNv_TPMA_NV_PLATFORMCREATE(nvIndex.publicArea.attributes))
            {
                // Delete the index (including RAM for orderly)
                result = NvDeleteIndex(&nvIndex, currentAddr);
                if(result != TPM_RC_SUCCESS)
                    break;
                // Re-iterate from beginning after a delete
                iter = NV_REF_INIT;
            }
        }
        else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
        {
            OBJECT_ATTRIBUTES           attributes;
            NvRead(&attributes,
                   (UINT32)(currentAddr
                            + sizeof(TPM_HANDLE)
                            + offsetof(OBJECT, attributes)),
                   sizeof(OBJECT_ATTRIBUTES));
               // If the evict object belongs to the hierarchy to be flushed
            if((hierarchy == TPM_RH_PLATFORM &&  attributes.ppsHierarchy == SET)
               || (hierarchy == TPM_RH_OWNER &&  attributes.spsHierarchy == SET)
               || (hierarchy == TPM_RH_ENDORSEMENT
                   &&  attributes.epsHierarchy == SET))
            {
                // Delete the evict object
                result = NvDelete(currentAddr);
                if(result != TPM_RC_SUCCESS)
                    break;
                // Re-iterate from beginning after a delete
                iter = NV_REF_INIT;
            }
        }
        else
        {
            FAIL(FATAL_ERROR_INTERNAL);
        }
    }
    return result;
}
```

8.4.5.20 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().

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

```
906  TPM_RC
907  NvSetGlobalLock(
908      void
909      )
910  {
911      NV_REF           iter = NV_REF_INIT;
912      NV_RAM_REF       ramIter = NV_RAM_REF_INIT;
913      NV_REF           currentAddr;
914      NV_RAM_REF       currentRamAddr;
915      TPM_RC           result = TPM_RC_SUCCESS;
916      // Check all normal Indices
917      while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
918      {
919          TPMA_NV         attributes = NvReadNvIndexAttributes(currentAddr);
920          // See if it should be locked
921          if(!IsNv_TPMA_NV_ORDERLY(attributes)
922             &&  IsNv_TPMA_NV_GLOBALLOCK(attributes))
923          {
924              attributes.TPMA_NV_WRITELOCKED = SET;
925              result = NvWriteNvIndexAttributes(currentAddr, attributes);
926              if(result != TPM_RC_SUCCESS)
927                  return result;
928          }
929      }
930      // Now search all the orderly attributes
931      while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
932      {
933          // See if it should be locked
934          TPMA_NV         attributes = NvReadRamIndexAttributes(currentRamAddr);
935          if(IsNv_TPMA_NV_GLOBALLOCK(attributes))
936          {
937              attributes.TPMA_NV_WRITELOCKED = SET;
938              NvWriteRamIndexAttributes(currentRamAddr, attributes);
939          }
940      }
941      return result;
942  }
```

8.4.5.21 InsertSort()

Sort a handle into handle list in ascending order. The total handle number in the list should not exceed MAX_CAP_HANDLES

```
943  static void
944  InsertSort(
945      TPML_HANDLE     *handleList,    // IN/OUT: sorted handle list
946      UINT32           count,         // IN: maximum count in the handle list
947      TPM_HANDLE       entityHandle   // IN: handle to be inserted
948      )
949  {
950      UINT32          i, j;
951      UINT32          originalCount;
952      // For a corner case that the maximum count is 0, do nothing
953      if(count == 0)
954          return;
955      // For empty list, add the handle at the beginning and return
956      if(handleList->count == 0)
957      {
```

```
958             handleList->handle[0] = entityHandle;
959             handleList->count++;
960             return;
961         }
962         // Check if the maximum of the list has been reached
963         originalCount = handleList->count;
964         if(originalCount < count)
965             handleList->count++;
966         // Insert the handle to the list
967         for(i = 0; i < originalCount; i++)
968         {
969             if(handleList->handle[i] > entityHandle)
970             {
971                 for(j = handleList->count - 1; j > i; j--)
972                 {
973                     handleList->handle[j] = handleList->handle[j - 1];
974                 }
975                 break;
976             }
977         }
978         // If a slot was found, insert the handle in this position
979         if(i < originalCount || handleList->count > originalCount)
980             handleList->handle[i] = entityHandle;
981         return;
982     }
```

8.4.5.22 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

```
983     TPMI_YES_NO
984     NvCapGetPersistent(
985         TPMI_DH_OBJECT   handle,        // IN: start handle
986         UINT32           count,         // IN: maximum number of returned handles
987         TPML_HANDLE     *handleList     // OUT: list of handle
988         )
989     {
990         TPMI_YES_NO             more = NO;
991         NV_REF                  iter = NV_REF_INIT;
992         NV_REF                  currentAddr;
993         TPM_HANDLE              entityHandle;
994         pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
995         // Initialize output handle list
996         handleList->count = 0;
997         // The maximum count of handles we may return is MAX_CAP_HANDLES
998         if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
999         while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
1000        {
1001            // Ignore persistent handles that have values less than the input handle
1002            if(entityHandle < handle)
1003                continue;
1004            // if the handles in the list have reached the requested count, and there
1005            // are still handles need to be inserted, indicate that there are more.
1006            if(handleList->count == count)
1007                more = YES;
```

```
1008            // A handle with a value larger than start handle is a candidate
1009            // for return. Insert sort it to the return list.  Insert sort algorithm
1010            // is chosen here for simplicity based on the assumption that the total
1011            // number of NV Indices is small.  For an implementation that may allow
1012            // large number of NV Indices, a more efficient sorting algorithm may be
1013            // used here.
1014            InsertSort(handleList, count, entityHandle);
1015        }
1016        return more;
1017    }
```

8.4.5.23 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

```
1018    TPMI_YES_NO
1019    NvCapGetIndex(
1020        TPMI_DH_OBJECT   handle,        // IN: start handle
1021        UINT32           count,         // IN: max number of returned handles
1022        TPML_HANDLE     *handleList     // OUT: list of handle
1023        )
1024    {
1025        TPMI_YES_NO              more = NO;
1026        NV_REF                   iter = NV_REF_INIT;
1027        NV_REF                   currentAddr;
1028        TPM_HANDLE               nvHandle;
1029        pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1030        // Initialize output handle list
1031        handleList->count = 0;
1032        // The maximum count of handles we may return is MAX_CAP_HANDLES
1033        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1034        while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1035        {
1036            // Ignore index handles that have values less than the 'handle'
1037            if(nvHandle < handle)
1038                continue;
1039            // if the count of handles in the list has reached the requested count,
1040            // and there are still handles to report, set more.
1041            if(handleList->count == count)
1042                more = YES;
1043            // A handle with a value larger than start handle is a candidate
1044            // for return. Insert sort it to the return list.  Insert sort algorithm
1045            // is chosen here for simplicity based on the assumption that the total
1046            // number of NV Indices is small.  For an implementation that may allow
1047            // large number of NV Indices, a more efficient sorting algorithm may be
1048            // used here.
1049            InsertSort(handleList, count, nvHandle);
1050        }
1051        return more;
1052    }
```

8.4.5.24 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```
1053    UINT32
```

```
1054  NvCapGetIndexNumber(
1055      void
1056      )
1057  {
1058      UINT32          num = 0;
1059      NV_REF         iter = NV_REF_INIT;
1060      while(NvNextIndex(NULL, &iter) != 0)
1061          num++;
1062      return num;
1063  }
```

8.4.5.25 NvCapGetPersistentNumber()

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

```
1064  UINT32
1065  NvCapGetPersistentNumber(
1066      void
1067      )
1068  {
1069      UINT32          num = 0;
1070      NV_REF         iter = NV_REF_INIT;
1071      TPM_HANDLE      handle;
1072      while(NvNextEvict(&handle, &iter) != 0)
1073          num++;
1074      return num;
1075  }
```

8.4.5.26 NvCapGetPersistentAvail()

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

```
1076  UINT32
1077  NvCapGetPersistentAvail(
1078      void
1079      )
1080  {
1081      UINT32          availNVSpace;
1082      UINT32          counterNum = NvCapGetCounterNumber();
1083      UINT32          reserved = sizeof(NV_LIST_TERMINATOR);
1084      // Get the available space in NV storage
1085      availNVSpace = NvGetFreeBytes();
1086      if(counterNum < MIN_COUNTER_INDICES)
1087      {
1088          // Some space has to be reserved for counter objects.
1089          reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
1090          if(reserved > availNVSpace)
1091              availNVSpace = 0;
1092          else
1093              availNVSpace -= reserved;
1094      }
1095      return availNVSpace / NV_EVICT_OBJECT_SIZE;
1096  }
```

8.4.5.27 NvCapGetCounterNumber()

Get the number of defined NV Indexes that are counter indices.

```
1097  UINT32
1098  NvCapGetCounterNumber(
```

```
1099        void
1100        )
1101    {
1102        NV_REF           iter = NV_REF_INIT;
1103        NV_REF           currentAddr;
1104        UINT32           num = 0;
1105        while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1106        {
1107            TPMA_NV             attributes = NvReadNvIndexAttributes(currentAddr);
1108            if(IsNvCounterIndex(attributes))
1109                num++;
1110        }
1111        return num;
1112    }
```

8.4.5.28 NvSetStartupAttributes()

Local function to set the attributes of an Index at TPM Reset and TPM Restart.

```
1113    static TPMA_NV
1114    NvSetStartupAttributes(
1115        TPMA_NV         attributes,         // IN: attributes to change
1116        STARTUP_TYPE     type           // IN: start up type
1117        )
1118    {
1119        // Clear read lock
1120        attributes.TPMA_NV_READLOCKED = CLEAR;
1121        // Will change a non counter index to the unwritten state if:
1122        // a) TPMA_NV_CLEAR_STCLEAR is SET
1123        // b) orderly and TPM Reset
1124        if(!IsNvCounterIndex(attributes))
1125        {
1126            if(IsNv_TPMA_NV_CLEAR_STCLEAR(attributes)
1127               || (IsNv_TPMA_NV_ORDERLY(attributes) && (type == SU_RESET)))
1128                attributes.TPMA_NV_WRITTEN = CLEAR;
1129        }
1130        // Unlock any index that is not written or that does not have
1131        // TPMA_NV_WRITEDEFINE SET.
1132        if(!IsNv_TPMA_NV_WRITTEN(attributes) || !IsNv_TPMA_NV_WRITEDEFINE(attributes))
1133            attributes.TPMA_NV_WRITELOCKED = CLEAR;
1134        return attributes;
1135    }
```

8.4.5.29 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.

```
1136    void
1137    NvEntityStartup(
1138        STARTUP_TYPE     type           // IN: start up type
1139        )
1140    {
1141        NV_REF              iter = NV_REF_INIT;
1142        NV_RAM_REF          ramIter = NV_RAM_REF_INIT;
1143        NV_REF              currentAddr;        // offset points to the current entity
```

```
    NV_RAM_REF          currentRamAddr;
    TPM_HANDLE          nvHandle;
    TPMA_NV             attributes;
    // Restore RAM index data
    NvRead(s_indexOrderlyRam, NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam));
    // Initialize the max NV counter value
    NvSetMaxCount(NvGetMaxCount());
    // If recovering from state save, do nothing else
    if(type == SU_RESUME)
        return;
    // Iterate all the NV Index to clear the locks
    while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
    {
        attributes = NvReadNvIndexAttributes(currentAddr);
        // If this is an orderly index, defer processing until loop below
        if(IsNv_TPMA_NV_ORDERLY(attributes))
            continue;
        // Set the attributes appropriate for this startup type
        attributes = NvSetStartupAttributes(attributes, type);
        NvWriteNvIndexAttributes(currentAddr, attributes);
    }
    // Iterate all the orderly indices to clear the locks and initialize counters
    while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
    {
        attributes = NvReadRamIndexAttributes(currentRamAddr);
        attributes = NvSetStartupAttributes(attributes, type);
        // update attributes in RAM
        NvWriteRamIndexAttributes(currentRamAddr, attributes);
        // Set the lower bits in an orderly counter to 1 for a non-orderly startup
        if(IsNvCounterIndex(attributes)
           && (g_prevOrderlyState == SU_NONE_VALUE))
        {
            UINT64      counter;
            // Read the counter value last saved to NV.
            counter = BYTE_ARRAY_TO_UINT64(currentRamAddr + sizeof(NV_RAM_HEADER));
            // Set the lower bits of counter to 1's
            counter |= MAX_ORDERLY_COUNT;
            // Write back to RAM
            // NOTE: Do not want to force a write to NV here. The counter value will
            // stay in RAM until the next shutdown or rollover.
            UINT64_TO_BYTE_ARRAY(counter, currentRamAddr + sizeof(NV_RAM_HEADER));
        }
    }
    return;
}
```

8.4.5.30 NvCapGetCounterAvail()

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

```
UINT32
NvCapGetCounterAvail(
    void
    )
{
    UINT32          availNVSpace;
    UINT32          availRAMSpace;
    UINT32          persistentNum = NvCapGetPersistentNumber();
    UINT32          reserved = sizeof(NV_LIST_TERMINATOR);
    // Get the available space in NV storage
    availNVSpace = NvGetFreeBytes();
    if(persistentNum < MIN_EVICT_OBJECTS)
    {
        // Some space has to be reserved for evict object. Adjust availNVSpace.
```

```
1203            reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
1204            if(reserved > availNVSpace)
1205                availNVSpace = 0;
1206            else
1207                availNVSpace -= reserved;
1208        }
1209        // Compute the available space in RAM
1210        availRAMSpace = RAM_ORDERLY_END - NvRamGetEnd();
1211        // Return the min of counter number in NV and in RAM
1212        if(availNVSpace / NV_INDEX_COUNTER_SIZE
1213            > availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE)
1214            return availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE;
1215        else
1216            return availNVSpace / NV_INDEX_COUNTER_SIZE;
1217    }
```

8.4.5.31 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.

```
1218    NV_REF
1219    NvFindHandle(
1220        TPM_HANDLE       handle
1221        )
1222    {
1223        NV_REF           addr;
1224        NV_REF           iter = NV_REF_INIT;
1225        TPM_HANDLE       nextHandle;
1226        while((addr = NvNext(&iter, &nextHandle)) != 0)
1227        {
1228            if(nextHandle == handle)
1229                break;
1230        }
1231        return addr;
1232    }
```

8.4.6 NV Max Counter

8.4.6.1 Introduction

The TPM keeps track of the highest value of a deleted counter index. When an index is deleted, this value is updated if the deleted counter index is greater than the previous value. When a new index is created and first incremented, it will get a value that is at least one greater than any other index than any previously deleted index. This insures that it is not possible to roll back an index.

The highest counter value is keep in NV in a special end-of-list marker. This marker is only updated when an index is deleted. Otherwise it just moves.

When the TPM starts up, it searches NV for the end of list marker and initializes an in memory value (*s_maxCounter*).

8.4.6.2 NvReadMaxCount()

This function returns the max NV counter value.

```
1233    UINT64
1234    NvReadMaxCount(
1235        void
1236        )
```

```
1237    {
1238        return s_maxCounter;
1239    }
```

8.4.6.3 NvUpdateMaxCount()

This function updates the max counter value to NV memory. This is just staging for the actual write that will occur when the NV index memory is modified.

```
1240    void
1241    NvUpdateMaxCount(
1242        UINT64          count
1243        )
1244    {
1245        if(count > s_maxCounter)
1246            s_maxCounter = count;
1247    }
```

8.4.6.4 NvSetMaxCount()

This function is used at NV initialization time to set the initial value of the maximum counter.

```
1248    void
1249    NvSetMaxCount(
1250        UINT64          value
1251        )
1252    {
1253        s_maxCounter = value;
1254    }
```

8.4.6.5 NvGetMaxCount()

Function to get the NV max counter value from the end-of-list marker

```
1255    UINT64
1256    NvGetMaxCount(
1257        void
1258        )
1259    {
1260        NV_REF              iter = NV_REF_INIT;
1261        NV_REF              currentAddr;
1262        UINT64              maxCount;
1263        // Find the end of list marker and initialize the NV Max Counter value.
1264        while((currentAddr = NvNext(&iter, NULL )) != 0);
1265        // 'iter' should be pointing at the end of list marker so read in the current
1266        // value of the s_maxCounter.
1267        NvRead(&maxCount, iter + sizeof(UINT32), sizeof(maxCount));
1268        return maxCount;
1269    }
```

8.5 NVReserved.c

8.5.1 Introduction

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

The entries in dynamic space are a linked list of entries. Each entry has, as its first field, a size. If the size field is zero, it marks the end of the list.

An allocation of an Index or evict object may use almost all of the remaining NV space such that the size field will not fit. The functions that search the list are aware of this and will terminate the search if they either find a zero size or recognize that there is insufficient space for the size field.

An Index allocation will contain an NV_INDEX structure. If the Index does not have the orderly attribute, the NV_INDEX is followed immediately by the NV data.

An evict object entry contains a handle followed by an OBJECT structure. This results in both the Index and Evict Object having an identifying handle as the first field following the size field.

When an Index has the orderly attribute, the data is kept in RAM. This RAM is saved to backing store in NV memory on any orderly shutdown. The entries in orderly memory are also a linked list using a size field as the first entry. As with the NV memory, the list is terminated by a zero size field or when the last entry leaves insufficient space for the terminating size field.

The attributes of an orderly index are maintained in RAM memory in order to reduce the number of NV writes needed for orderly data. When an orderly index is created, an entry is made in the dynamic NV memory space that holds the Index authorizations *(authPolicy* and *authValue*) and the size of the data. This entry is only modified if the *authValue* of the index is changed. The more volatile data of the index is kept in RAM. When an orderly Index is created or deleted, the RAM data is copied to NV backing store so that the image in the backing store matches the layout of RAM. In normal operation. The RAM data is also copied on any orderly shutdown. In normal operation, the only other reason for writing to the backing store for RAM is when a counter is first written (TPMA_NV_WRITTEN changes from CLEAR to SET) or when a counter "rolls over."

Static space contains items that are individually modifiable. The values are in the *gp* PERSISTEND_DATA structure in RAM and mapped to locations in NV.

8.5.2 Includes, Defines and Data Definitions

```
1   #define NV_C
2   #include    "Tpm.h"
```

8.5.3 Functions

8.5.3.1 NvInitStatic()

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

```
3   static void
4   NvInitStatic(
5       void
6       )
7   {
8       // In some implementations, the end of NV is variable and is set at boot time.
9       // This value will be the same for each boot, but is not necessarily known
10      // at compile time.
11      s_evictNvEnd = (NV_REF)NV_MEMORY_SIZE;
12      return;
```

```
}
```

8.5.3.2 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 check of *g_NvStatus*.

```
void
NvCheckState(
    void
    )
{
    int     func_return;
//
    func_return = _plat__IsNvAvailable();
    if(func_return == 0)
        g_NvStatus = TPM_RC_SUCCESS;
    else if(func_return == 1)
        g_NvStatus = TPM_RC_NV_UNAVAILABLE;
    else
        g_NvStatus = TPM_RC_NV_RATE;
    return;
}
```

8.5.3.3 NvCommit

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

```
BOOL
NvCommit(
    void
    )
{
    return (_plat__NvCommit() == 0);
}
```

8.5.3.4 NvPowerOn()

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

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

```
BOOL
NvPowerOn(
    void
    )
{
    int          nvError = 0;
    // If power was lost, need to re-establish the RAM data that is loaded from
    // NV and initialize the static variables
    if(g_powerWasLost)
    {
        if((nvError = _plat__NVEnable(0)) < 0)
            FAIL(FATAL_ERROR_NV_UNRECOVERABLE);
```

```
49          NvInitStatic();
50      }
51      return nvError == 0;
52  }
```

8.5.3.5 NvManufacture()

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.

```
53  void
54  NvManufacture(
55      void
56      )
57  {
58  #ifdef SIMULATION
59      // Simulate the NV memory being in the erased state.
60      _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
61  #endif
62      // Initialize static variables
63      NvInitStatic();
64      // Clear the RAM used for Orderly Index data
65      MemorySet(s_indexOrderlyRam, 0, RAM_INDEX_SPACE);
66      // Write that Orderly Index data to NV
67      NvWrite(NV_ORDERLY_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
68      // Initialize the next offset of the first entry in evict/index list to 0 (the
69      // end of list marker) and the initial s_maxCounterValue;
70      NvSetMaxCount(0);
71      // Put the end of list marker at the end of memory. This contains the MaxCount
72      // value as well as the end marker.
73      NvWriteNvListEnd(NV_USER_DYNAMIC);
74      return;
75  }
```

8.5.3.6 NvRead()

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

```
76  void
77  NvRead(
78      void            *outBuffer,     // OUT: buffer to receive data
79      UINT32           nvOffset,      // IN: offset in NV of value
80      UINT32           size           // IN: size of the value to read
81      )
82  {
83      // Input type should be valid
84      pAssert(nvOffset + size < NV_MEMORY_SIZE);
85      _plat__NvMemoryRead(nvOffset, size, outBuffer);
86      return;
87  }
```

8.5.3.7 NvWrite()

This function is used to post reserved data for writing to NV memory. Before the TPM completes the operation, the value will be written.

```
88  void
89  NvWrite(
90      UINT32           nvOffset,      // IN: location in NV to receive data
```

```
 91      UINT32          size,           // IN: size of the data to move
 92      void           *inBuffer        // IN: location containing data to write
 93      )
 94  {
 95      // Input type should be valid
 96      pAssert(nvOffset + size <= NV_MEMORY_SIZE);
 97      _plat__NvMemoryWrite(nvOffset, size, inBuffer);
 98      // Set the flag that a NV write happened
 99      SET_NV_UPDATE(UT_NV);
100      return;
101  }
```

8.5.3.8 NvUpdatePersistent()

This function is used to update a value in the PERSISTENT_DATA structure and commits the value to NV.

```
102  void
103  NvUpdatePersistent(
104      UINT32          offset,         // IN: location in PERMANENT_DATA to be updated
105      UINT32          size,           // IN: size of the value
106      void           *buffer          // IN: the new data
107      )
108  {
109      pAssert(offset + size <= sizeof(gp));
110      MemoryCopy(&gp + offset, buffer, size);
111      NvWrite(offset, size, buffer);
112  }
```

8.5.3.9 NvClearPersistent()

This function is used to clear a persistent data entry and commit it to NV

```
113  void
114  NvClearPersistent(
115      UINT32          offset,         // IN: the offset in the PERMANENT_DATA
116                                      //     structure to be cleared (zeroed)
117      UINT32          size            // IN: number of bytes to clear
118      )
119  {
120      MemorySet((&gp) + offset, 0, size);
121      NvWrite(offset, size, (&gp) + offset);
122  }
```

8.5.3.10 NvReadPersistent()

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

```
123  void
124  NvReadPersistent(
125      void
126      )
127  {
128      NvRead(&gp, NV_PERSISTENT_DATA, sizeof(gp));
129      return;
130  }
```


8.6 Object.c

8.6.1 Introduction

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

8.6.2 Includes and Data Definitions

```
1   #define OBJECT_C
2   #include "Tpm.h"
```

8.6.3 Functions

8.6.3.1 ObjectFlush()

This function marks an object slot as available. Since there is no checking of the input parameters, it should be used judiciously.

NOTE: This could be converted to a macro.

```
3   void
4   ObjectFlush(
5       OBJECT          *object
6       )
7   {
8       object->attributes.occupied = CLEAR;
9   //    MemorySet(&object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));
10  }
```

8.6.3.2 ObjectSetInUse()

This access function sets the occupied attribute of an object slot.

```
11  void
12  ObjectSetInUse(
13      OBJECT          *object
14      )
15  {
16      object->attributes.occupied = SET;
17  }
```

8.6.3.3 ObjectStartup()

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

```
18  void
19  ObjectStartup(
20      void
21      )
22  {
23      UINT32      i;
24      // object slots initialization
25      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
26      {
27          //Set the slot to not occupied
28          ObjectFlush(&s_objects[i]);
29      }
```

```
30      return;
31  }
```

8.6.3.4 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().

```
32  void
33  ObjectCleanupEvict(
34      void
35      )
36  {
37      UINT32      i;
38      // This has to be iterated because a command may have two handles
39      // and they may both be persistent.
40      // This could be made to be more efficient so that a search is not needed.
41      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
42      {
43          // If an object is a temporary evict object, flush it from slot
44          OBJECT      *object = &s_objects[i];
45          if(object->attributes.evict == SET)
46              ObjectFlush(object);
47      }
48      return;
49  }
```

8.6.3.5 IsObjectPresent()

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

```
50  BOOL
51  IsObjectPresent(
52      TPMI_DH_OBJECT   handle         // IN: handle to be checked
53      )
54  {
55      UINT32          slotIndex = handle - TRANSIENT_FIRST;
56      // Since the handle is just an index into the array that is zero based, any
57      // handle value outsize of the range of:
58      //    TRANSIENT_FIRST -- (TRANSIENT_FIRST + MAX_LOADED_OBJECT - 1)
59      // will now be greater than or equal to MAX_LOADED_OBJECTS
60      if(slotIndex >= MAX_LOADED_OBJECTS)
61          return FALSE;
62      // Indicate if the slot is occupied
63      return (s_objects[slotIndex].attributes.occupied == TRUE);
64  }
```

8.6.3.6 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

```
65  BOOL
66  ObjectIsSequence(
67      OBJECT          *object         // IN: handle to be checked
68      )
69  {
70      pAssert(object != NULL);
71      return (object->attributes.hmacSeq == SET
72              || object->attributes.hashSeq == SET
73              || object->attributes.eventSeq == SET);
74  }
```

8.6.3.7 HandleToObject()

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

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

```
75  OBJECT*
76  HandleToObject(
77      TPMI_DH_OBJECT   handle         // IN: handle of the object
78      )
79  {
80      UINT32              index;
81      // Return NULL if the handle references a permanent handle because there is no
82      // associated OBJECT.
83      if(HandleGetType(handle) == TPM_HT_PERMANENT)
84          return NULL;
85      // In this implementation, the handle is determined by the slot occupied by the
86      // object.
87      index = handle - TRANSIENT_FIRST;
88      pAssert(index < MAX_LOADED_OBJECTS);
89      pAssert(s_objects[index].attributes.occupied);
90      return &s_objects[index];
91  }
```

8.6.3.8 ObjectGetNameAlg()

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

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

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

8.6.3.9 GetQualifiedName()

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.

```
 99  void
100  GetQualifiedName(
101      TPMI_DH_OBJECT   handle,        // IN: handle of the object
102      TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
103      )
104  {
105      OBJECT      *object;
106      switch(HandleGetType(handle))
107      {
108          case TPM_HT_PERMANENT:
109              qualifiedName->t.size = sizeof(TPM_HANDLE);
110              UINT32_TO_BYTE_ARRAY(handle, qualifiedName->t.name);
111              break;
112          case TPM_HT_TRANSIENT:
113              object = HandleToObject(handle);
114              if(object == NULL || object->publicArea.nameAlg == TPM_ALG_NULL)
115                  qualifiedName->t.size = 0;
116              else
117                  // Copy the name
118                  *qualifiedName = object->qualifiedName;
119              break;
120          default:
121              FAIL(FATAL_ERROR_INTERNAL);
122      }
123      return;
124  }
```

8.6.3.10 ObjectGetHierarchy()

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

```
125  TPMI_RH_HIERARCHY
126  ObjectGetHierarchy(
127      OBJECT          *object         // IN :object
128      )
129  {
130      if(object->attributes.spsHierarchy)
131      {
132          return TPM_RH_OWNER;
133      }
134      else if(object->attributes.epsHierarchy)
135      {
136          return TPM_RH_ENDORSEMENT;
137      }
138      else if(object->attributes.ppsHierarchy)
139      {
140          return TPM_RH_PLATFORM;
141      }
142      else
143      {
144          return TPM_RH_NULL;
145      }
146  }
```

8.6.3.11 GetHeriarchy()

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

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

```
147 TPMI_RH_HIERARCHY
148 GetHeriarchy(
149     TPMI_DH_OBJECT   handle         // IN :object handle
150     )
151 {
152     OBJECT          *object = HandleToObject(handle);
153     return ObjectGetHierarchy(object);
154 }
```

8.6.3.12 FindEmptyObjectSlot()

This function finds an open object slot, if any. It will clear the attributes but will not set the occupied attribute. This is so that a slot may be used and discarded if everything does not go as planned.

Return Value	Meaning
null	no open slot found
!=null	pointer to available slot

```
155 OBJECT *
156 FindEmptyObjectSlot(
157     TPMI_DH_OBJECT  *handle         // OUT: (optional)
158     )
159 {
160     UINT32              i;
161     OBJECT             *object;
162     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
163     {
164         object = &s_objects[i];
165         if(object->attributes.occupied == CLEAR)
166         {
167             if(handle)
168                 *handle = i + TRANSIENT_FIRST;
169             // Initialize the object attributes
170             MemorySet(&object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));
171             return object;
172         }
173     }
174     return NULL;
175 }
```

8.6.3.13 ObjectAllocateSlot()

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

Return Value	Meaning

```
176 OBJECT *
177 ObjectAllocateSlot(
178     TPMI_DH_OBJECT  *handle        // OUT: handle of allocated object
179     )
180 {
181     OBJECT          *object = FindEmptyObjectSlot(handle);
182     if(object != NULL)
183     {
184         // if found, mark as occupied
185         ObjectSetInUse(object);
186     }
187     return object;
188 }
```


8.6.3.14 ObjectSetLoadedAttributes()

This function sets the internal attributes for a loaded object. It is called to finalize the OBJECT attributes (not the TPMA_OBJECT attributes) for a loaded object.

```
189  void
190  ObjectSetLoadedAttributes(
191      OBJECT          *object,        // IN: object attributes to finalize
192      TPM_HANDLE       parentHandle   // IN: the parent handle
193      )
194  {
195      OBJECT              *parent = NULL;
196      // Copy the stClear attribute from the public area. This could be overwritten
197      // if the parent has stClear SET
198      object->attributes.stClear = object->publicArea.objectAttributes.stClear;
199      // If parent handle is a permanent handle, it is a primary or temporary
200      // object
201      if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
202      {
203          // is this a temporary object with TPM_ALG_NULL as a parent?
204          // For an external object with the sensitive area loaded, the hierarchy
205          // is TPM_RH_NULL. If only the public part is loaded, then the
206          // hierarchy can be anything. Since LoadExternal only passes the hierarchy
207          // need to make sure that we don't indicate that the object is permanent
208          // Any key with TPM_RH_NULL as a parent is a temporary object.
209          if(parentHandle == TPM_RH_NULL || object->attributes.external == SET)
210              object->attributes.temporary = SET;
211          else
212              object->attributes.primary = SET;
213      }
214      else
215      {
216          // Check for stClear object
217          parent = HandleToObject(parentHandle);
218          if(object->publicArea.objectAttributes.stClear == SET
219             || ((parent != NULL) && (parent->attributes.stClear == SET)))
220              object->attributes.stClear = SET;
221      }
222      // For a LoadExternal object, the parent will be TPM_ALG_NULL if the sensitive
223      // portion is loaded so no hierarchy will be set here.
224      ObjectSetHierarchy(object, parentHandle, parent);
225      // If this is an external object, set the QN == name but don't SET other
226      // key properties ('parent' or 'derived')
227      if(object->attributes.external)
228          object->qualifiedName = object->name;
229      else
230      {
231          // check attributes for different types of parents
232          if(object->publicArea.objectAttributes.restricted
233             && !object->attributes.publicOnly
234             && object->publicArea.objectAttributes.decrypt
235             && object->publicArea.nameAlg != TPM_ALG_NULL)
236          {
237              // This is a parent. If it is not a KEYEDHASH, it is an ordinary parent.
238              // Otherwise, it is a derivation parent.
239              if(object->publicArea.type == TPM_ALG_KEYEDHASH)
240                  object->attributes.derivation = SET;
241              else
242                  object->attributes.isParent = SET;
243          }
244          ComputeQualifiedName(parentHandle, object->publicArea.nameAlg,
245                               &object->name, &object->qualifiedName);
246      }
247      // Set slot occupied
248      ObjectSetInUse(object);
```

```
249         return;
250     }
```

8.6.3.15 ObjectLoad()

Common function to load an object. A loaded object has its public area validated (unless its *nameAlg* is TPM_ALG_NULL). If a sensitive part is loaded, it is verified to be correct and if both public and sensitive parts are loaded, then the cryptographic binding between the objects is validated. This function does not cause the allocated slot to be marked as in use.

```
251     TPM_RC
252     ObjectLoad(
253         OBJECT          *object,        // IN: pointer to object slot
254                                         //     object
255         OBJECT          *parent,        // IN: (optional) the parent object
256         TPMT_PUBLIC     *publicArea,    // IN: public area to be installed in the object
257         TPMT_SENSITIVE  *sensitive,     // IN: (optional) sensitive area to be
258                                         //      installed in the object
259         TPM_RC           blamePublic,   // IN: parameter number to associate with the
260                                         //     publicArea errors
261         TPM_RC           blameSensitive,// IN: parameter number to associate with the
262                                         //     sensitive area errors
263         TPM2B_NAME      *name           // IN: (optional)
264         )
265     {
266         TPM_RC           result = TPM_RC_SUCCESS;
267         BOOL             doCheck;
268     // Do validations of public area object descriptions
269         // Is this public only or a no-name object?
270         if(sensitive == NULL || publicArea->nameAlg == TPM_ALG_NULL)
271         {
272             // Need to have schemes checked so that we do the right thing with the
273             // public key.
274             result = SchemeChecks(NULL, publicArea);
275         }
276         else
277         {
278             // Check attributes and schemes for consistency
279             result = PublicAttributesValidation(parent, publicArea);
280         }
281         if(result != TPM_RC_SUCCESS)
282             return RcSafeAddToResult(result, blamePublic);
283         // If object == NULL, then this is am import. For import, load is not called
284         // unless the parent is fixedTPM.
285         if(object == NULL)
286             doCheck = TRUE;// //
287         // If the parent is not NULL, then this is an ordinary load and we only check
288         // if the parent is not fixedTPM
289         else if(parent != NULL)
290             doCheck = parent->publicArea.objectAttributes.fixedTPM == CLEAR;
291         else
292             // This is a loadExternal. Check everything.
293             // Note: the check functions will filter things based on the name algorithm
294             // and whether or not both parts are loaded.
295             doCheck = TRUE;
296         // Note: the parent will be NULL if this is a load external. CryptValidateKeys()
297         // will only check the parts that need to be checked based on the settings
298         // of publicOnly and nameAlg.
299         // Note: For an RSA key, the keys sizes are checked but the binding is not
300         // checked.
301         if(doCheck)
302         {
303             // Do the cryptographic key validation
304             result = CryptValidateKeys(publicArea, sensitive, blamePublic,
```

```
                                      blameSensitive);
    }
    // If this is an import, we are done
    if(object == NULL || result != TPM_RC_SUCCESS)
        return result;
    // Set the name, if one was provided
    if(name != NULL)
        object->name = *name;
    else
        object->name.t.size = 0;
    // Initialize public
    object->publicArea = *publicArea;
    // If there is a sensitive area, load it
    if(sensitive == NULL)
        object->attributes.publicOnly = SET;
    else
    {
        object->sensitive = *sensitive;
#ifdef TPM_ALG_RSA
        // If this is an RSA key that is not a parent, complete the load by
        // computing the private exponent.
        if(publicArea->type == ALG_RSA_VALUE)
            result = CryptRsaLoadPrivateExponent(object);
#endif
    }
    return result;
}
```

8.6.3.16 AllocateSequenceSlot()

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

```
static HASH_OBJECT *
AllocateSequenceSlot(
    TPM_HANDLE      *newHandle,     // OUT: receives the allocated handle
    TPM2B_AUTH      *auth           // IN: the authValue for the slot
    )
{
    HASH_OBJECT      *object = (HASH_OBJECT *)ObjectAllocateSlot(newHandle);
//
    // Validate that the proper location of the hash state data relative to the
    // object state data. It would be good if this could have been done at compile
    // time but it can't so do it in something that can be removed after debug.
    cAssert(offsetof(HASH_OBJECT, auth) == offsetof(OBJECT, publicArea.authPolicy));
    if(object != NULL)
    {
    // Set the common values that a sequence object shares with an ordinary object
    // The type is TPM_ALG_NULL
        object->type = TPM_ALG_NULL;
        // This has no name algorithm and the name is the Empty Buffer
        object->nameAlg = TPM_ALG_NULL;
        // A sequence object is considered to be in the NULL hierarchy so it should
        // be marked as temporary so that it can't be persisted
        object->attributes.temporary = SET;
        // A sequence object is DA exempt.
        object->objectAttributes.noDA = SET;
        // Copy the authorization value
        if(auth != NULL)
            object->auth = *auth;
        else
            object->auth.t.size = 0;
    }
    return object;
```

```
363 }
```

8.6.3.17 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

```
364 TPM_RC
365 ObjectCreateHMACSequence(
366     TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
367     OBJECT           *keyObject,    // IN: the object containing the HMAC key
368     TPM2B_AUTH       *auth,         // IN: authValue
369     TPMI_DH_OBJECT   *newHandle     // OUT: HMAC sequence object handle
370     )
371 {
372     HASH_OBJECT         *hmacObject;
373     // Try to allocate a slot for new object
374     hmacObject = AllocateSequenceSlot(newHandle, auth);
375     if(hmacObject == NULL)
376         return TPM_RC_OBJECT_MEMORY;
377     // Set HMAC sequence bit
378     hmacObject->attributes.hmacSeq = SET;
379     CryptHmacStart(&hmacObject->state.hmacState, hashAlg,
380                    keyObject->sensitive.sensitive.bits.b.size,
381                    keyObject->sensitive.sensitive.bits.b.buffer);
382     return TPM_RC_SUCCESS;
383 }
```

8.6.3.18 ObjectCreateHashSequence()

This function creates a hash sequence object.

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

```
384 TPM_RC
385 ObjectCreateHashSequence(
386     TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
387     TPM2B_AUTH       *auth,         // IN: authValue
388     TPMI_DH_OBJECT   *newHandle     // OUT: sequence object handle
389     )
390 {
391     HASH_OBJECT         *hashObject = AllocateSequenceSlot(newHandle, auth);
392     // See if slot allocated
393     if(hashObject == NULL)
394         return TPM_RC_OBJECT_MEMORY;
395     // Set hash sequence bit
396     hashObject->attributes.hashSeq = SET;
397     // Start hash for hash sequence
398     CryptHashStart(&hashObject->state.hashState[0], hashAlg);
399     return TPM_RC_SUCCESS;
400 }
```

8.6.3.19 ObjectCreateEventSequence()

This function creates an event sequence object.

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

```
401 TPM_RC
402 ObjectCreateEventSequence(
403     TPM2B_AUTH      *auth,          // IN: authValue
404     TPMI_DH_OBJECT  *newHandle      // OUT: sequence object handle
405     )
406 {
407     HASH_OBJECT         *hashObject = AllocateSequenceSlot(newHandle, auth);
408     UINT32               count;
409     TPM_ALG_ID           hash;
410     // See if slot allocated
411     if(hashObject == NULL)
412         return TPM_RC_OBJECT_MEMORY;
413     // Set the event sequence attribute
414     hashObject->attributes.eventSeq = SET;
415     // Initialize hash states for each implemented PCR algorithms
416     for(count = 0; (hash = CryptHashGetAlgByIndex(count)) != TPM_ALG_NULL; count++)
417         CryptHashStart(&hashObject->state.hashState[count], hash);
418     return TPM_RC_SUCCESS;
419 }
```

8.6.3.20 ObjectTerminateEvent()

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

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

8.6.3.21 ObjectContextLoad()

This function loads an object from a saved object context.

Return Value	Meaning
NULL	if there is no free slot for an object
NON_NULL	points to the loaded object

```
445 OBJECT *
446 ObjectContextLoad(
447     ANY_OBJECT_BUFFER   *object,        // IN: pointer to object structure in saved
448                                         //     context
449     TPMI_DH_OBJECT      *handle         // OUT: object handle
450     )
451 {
452     OBJECT      *newObject = ObjectAllocateSlot(handle);
453     // Try to allocate a slot for new object
454     if(newObject != NULL)
455     {
456         // Copy the first part of the object
457         MemoryCopy(newObject, object, offsetof(HASH_OBJECT, state));
458         // See if this is a sequence object
459         if(ObjectIsSequence(newObject))
460         {
461             // If this is a sequence object, import the data
462             SequenceDataImport((HASH_OBJECT *)newObject,
463                                (HASH_OBJECT_BUFFER *)object);
464         }
465         else
466         {
467             // Copy input object data to internal structure
468             MemoryCopy(newObject, object, sizeof(OBJECT));
469         }
470     }
471     return newObject;
472 }
```

8.6.3.22 FlushObject()

This function frees an object slot.

This function requires that the object is loaded.

```
473 void
474 FlushObject(
475     TPMI_DH_OBJECT   handle         // IN: handle to be freed
476     )
477 {
478     UINT32      index = handle - TRANSIENT_FIRST;
479     pAssert(index < MAX_LOADED_OBJECTS);
480     // Clear all the object attributes
481     MemorySet((BYTE*)&(s_objects[index].attributes),
482               0, sizeof(OBJECT_ATTRIBUTES));
483     return;
484 }
```

8.6.3.23 ObjectFlushHierarchy()

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

```
485 void
486 ObjectFlushHierarchy(
487     TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flush
```

```
488     )
489 {
490     UINT16          i;
491     // iterate object slots
492     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
493     {
494         if(s_objects[i].attributes.occupied)          // If found an occupied slot
495         {
496             switch(hierarchy)
497             {
498                 case TPM_RH_PLATFORM:
499                     if(s_objects[i].attributes.ppsHierarchy == SET)
500                         s_objects[i].attributes.occupied = FALSE;
501                     break;
502                 case TPM_RH_OWNER:
503                     if(s_objects[i].attributes.spsHierarchy == SET)
504                         s_objects[i].attributes.occupied = FALSE;
505                     break;
506                 case TPM_RH_ENDORSEMENT:
507                     if(s_objects[i].attributes.epsHierarchy == SET)
508                         s_objects[i].attributes.occupied = FALSE;
509                     break;
510                 default:
511                     FAIL(FATAL_ERROR_INTERNAL);
512                     break;
513             }
514         }
515     }
516     return;
517 }
```

8.6.3.24 ObjectLoadEvict()

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

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

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

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

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

8.6.3.25 ObjectComputeName()

This does the name computation from a public area (can be marshaled or not).

```
562     TPM2B_NAME *
563     ObjectComputeName(
564         UINT32           size,          // IN: the size of the area to digest
565         BYTE            *publicArea,    // IN: the public area to digest area
566         TPM_ALG_ID       nameAlg,       // IN: the hash algorithm to use
567         TPM2B_NAME      *name           // OUT: Computed name
568         )
569     {
570         HASH_STATE           hashState;         // hash state
571     // Start hash stack
572         name->t.size = CryptHashStart(&hashState, nameAlg);
573         // Adding public area
574         CryptDigestUpdate(&hashState, size, publicArea);
575         // Complete hash leaving room for the name algorithm
576         CryptHashEnd(&hashState, name->t.size, &name->t.name[2]);
577         // set the nameAlg
578         UINT16_TO_BYTE_ARRAY(nameAlg, name->t.name);
579         name->t.size += 2;
580         return name;
581     }
```

8.6.3.26 PublicMarshalAndComputeName()

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

```
582     TPM2B_NAME *
583     PublicMarshalAndComputeName(
584         TPMT_PUBLIC     *publicArea,    // IN: public area of an object
585         TPM2B_NAME      *name           // OUT: name of the object
586         )
587     {
588         // Will marshal a public area into a template. This is because the internal
589         // format for a TPM2B_PUBLIC is a structure and not a simple BYTE buffer.
590         TPM2B_TEMPLATE       marshaled;     // this is big enough to hold a
591                                             //  marshaled TPMT_PUBLIC
```

```
592         BYTE                *buffer = (BYTE *)&marshaled.t.buffer;
593         // if the nameAlg is NULL then there is no name.
594         if(publicArea->nameAlg == TPM_ALG_NULL)
595             name->t.size = 0;
596         else
597         {
598             // Marshal the public area into its canonical form
599             marshaled.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
600             // and compute the name
601             ObjectComputeName(marshaled.t.size, marshaled.t.buffer,
602                               publicArea->nameAlg, name);
603         }
604         return name;
605     }
```

8.6.3.27 AlgOfName()

This function as a macro returns the *nameAlg* from a TPM2B_NAME.

```
606     TPMI_ALG_HASH
607     AlgOfName(
608         TPM2B_NAME      *name
609         )
610     {
611         return BYTE_ARRAY_TO_UINT16(name->t.name);
612     }
```

8.6.3.28 ComputeQualifiedName()

This function computes the qualified name of an object.

```
613     void
614     ComputeQualifiedName(
615         TPM_HANDLE       parentHandle,  // IN: parent's name
616         TPM_ALG_ID       nameAlg,       // IN: name hash
617         TPM2B_NAME      *name,          // IN: name of the object
618         TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
619         )
620     {
621         HASH_STATE      hashState;   // hash state
622         TPM2B_NAME      parentName;
623         if(parentHandle == TPM_RH_UNASSIGNED)
624         {
625             *qualifiedName = *name;
626         }
627         else
628         {
629             GetQualifiedName(parentHandle, &parentName);
630             //      QN_A = hash_A (QN of parent || NAME_A)
631             // Start hash
632             qualifiedName->t.size = CryptHashStart(&hashState, nameAlg);
633             // Add parent's qualified name
634             CryptDigestUpdate2B(&hashState, &parentName.b);
635             // Add self name
636             CryptDigestUpdate2B(&hashState, &name->b);
637             // Complete hash leaving room for the name algorithm
638             CryptHashEnd(&hashState, qualifiedName->t.size,
639                          &qualifiedName->t.name[2]);
640             UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
641             qualifiedName->t.size += 2;
642         }
643         return;
644     }
```

8.6.3.29 ObjectIsAsymParent()

This function determines if an object has the attributes associated with an asymmetric parent. An asymmetric parent is an asymmetric key 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

```
645 BOOL
646 ObjectIsStorage(
647     TPMI_DH_OBJECT   handle         // IN: object handle
648     )
649 {
650     OBJECT           *object = HandleToObject(handle);
651     TPMT_PUBLIC      *publicArea = ((object != NULL) ? &object->publicArea : NULL);
652     return (publicArea != NULL
653             && publicArea->objectAttributes.restricted == SET
654             && publicArea->objectAttributes.decrypt == SET
655             && publicArea->objectAttributes.sign == CLEAR
656             && (object->publicArea.type == ALG_RSA_VALUE
657                 || object->publicArea.type == ALG_ECC_VALUE));
658 }
```

8.6.3.30 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

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

```
684                 handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
685                 handleList->count++;
686             }
687             else
688             {
689                 // If the return list is full but we still have loaded object
690                 // available, report this and stop iterating
691                 more = YES;
692                 break;
693             }
694         }
695     }
696     return more;
697 }
```

8.6.3.31 ObjectCapGetTransientAvail()

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

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

8.6.3.32 ObjectGetPublicAttributes()

Returns the attributes associated with an object handles.

```
712 TPMA_OBJECT
713 ObjectGetPublicAttributes(
714     TPM_HANDLE       handle
715     )
716 {
717     return HandleToObject(handle)->publicArea.objectAttributes;
718 }
719 OBJECT_ATTRIBUTES
720 ObjectGetProperties(
721     TPM_HANDLE       handle
722     )
723 {
724     return HandleToObject(handle)->attributes;
725 }
```


8.7 PCR.c

8.7.1 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.

8.7.2 Includes, Defines, and Data Definitions

```
1   #define PCR_C
2   #include "Tpm.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.

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

8.7.3 Functions

8.7.3.1 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 authorization group
FALSE:	PCR does not belong an authorization group

```
20  BOOL
21  PCRBelongsAuthGroup(
22      TPMI_DH_PCR      handle,        // IN: handle of PCR
23      UINT32          *groupIndex     // OUT: group index if PCR belongs a
24                                      //      group that allows authValue.  If PCR
```

```
                                    //      does not belong to an authorization
                                    //      group, the value in this parameter is
                                    //      invalid
    )
{
#if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
    // Platform specification determines to which authorization group a PCR belongs
    // (if any). In this implementation, we assume there is only
    // one authorization group which contains PCR[20-22].  If the platform
    // specification requires differently, the implementation should be changed
    // accordingly
    if(handle >= 20 && handle <= 22)
    {
        *groupIndex = 0;
        return TRUE;
    }
#endif
    return FALSE;
}
```

8.7.3.2 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

```
BOOL
PCRBelongsPolicyGroup(
    TPMI_DH_PCR      handle,        // IN: handle of PCR
    UINT32          *groupIndex     // OUT: group index if PCR belongs a group that
                                    //     allows policy.  If PCR does not belong to
                                    //     a policy group, the value in this
                                    //     parameter is invalid
    )
{
#if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
    // Platform specification decides if a PCR belongs to a policy group and
    // belongs to which group.  In this implementation, we assume there is only
    // one policy group which contains PCR20-22.  If the platform specification
    // requires differently, the implementation should be changed accordingly
    if(handle >= 20 && handle <= 22)
    {
        *groupIndex = 0;
        return TRUE;
    }
#endif
    return FALSE;
}
```

8.7.3.3 PCRBelongsTCBGroup()

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

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

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

8.7.3.4 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

```
81  BOOL
82  PCRPolicyIsAvailable(
83      TPMI_DH_PCR      handle         // IN: PCR handle
84      )
85  {
86      UINT32          groupIndex;
87      return PCRBelongsPolicyGroup(handle, &groupIndex);
88  }
```

8.7.3.5 PCRGetAuthValue()

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

```
89  TPM2B_AUTH *
90  PCRGetAuthValue(
91      TPMI_DH_PCR      handle         // IN: PCR handle
92      )
93  {
94      UINT32      groupIndex;
95      if(PCRBelongsAuthGroup(handle, &groupIndex))
96      {
97          return &gc.pcrAuthValues.auth[groupIndex];
98      }
99      else
100     {
101         return NULL;
102     }
103 }
```

8.7.3.6 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.

```
104  TPMI_ALG_HASH
105  PCRGetAuthPolicy(
106      TPMI_DH_PCR      handle,        // IN: PCR handle
107      TPM2B_DIGEST    *policy         // OUT: policy of PCR
108      )
109  {
110      UINT32           groupIndex;
111      if(PCRBelongsPolicyGroup(handle, &groupIndex))
112      {
113          *policy = gp.pcrPolicies.policy[groupIndex];
114          return gp.pcrPolicies.hashAlg[groupIndex];
115      }
116      else
117      {
118          policy->t.size = 0;
119          return TPM_ALG_NULL;
120      }
121  }
```

8.7.3.7 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.

```
122  void
123  PCRSimStart(
124      void
125      )
126  {
127      UINT32  i;
128  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
129      for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
130      {
131          gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
132          gp.pcrPolicies.policy[i].t.size = 0;
133      }
134  #endif
135  #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
136      for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
137      {
138          gc.pcrAuthValues.auth[i].t.size = 0;
139      }
140  #endif
141      // We need to give an initial configuration on allocated PCR before
142      // receiving any TPM2_PCR_Allocate command to change this configuration
143      // When the simulation environment starts, we allocate all the PCRs
144      for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
145      gp.pcrAllocated.count++)
146      {
147          gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
148              = CryptHashGetAlgByIndex(gp.pcrAllocated.count);
149          gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
150              = PCR_SELECT_MAX;
151          for(i = 0; i < PCR_SELECT_MAX; i++)
152              gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
153              = 0xFF;
154      }
155      // Store the initial configuration to NV
```

```
156         NV_SYNC_PERSISTENT(pcrPolicies);
157         NV_SYNC_PERSISTENT(pcrAllocated);
158         return;
159     }
```

8.7.3.8 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

```
160     static BYTE *
161     GetSavedPcrPointer(
162         TPM_ALG_ID       alg,           // IN: algorithm for bank
163         UINT32           pcrIndex       // IN: PCR index in PCR_SAVE
164         )
165     {
166         switch(alg)
167         {
168     #ifdef TPM_ALG_SHA1
169             case TPM_ALG_SHA1:
170                 return gc.pcrSave.sha1[pcrIndex];
171                 break;
172     #endif
173     #ifdef TPM_ALG_SHA256
174             case TPM_ALG_SHA256:
175                 return gc.pcrSave.sha256[pcrIndex];
176                 break;
177     #endif
178     #ifdef TPM_ALG_SHA384
179             case TPM_ALG_SHA384:
180                 return gc.pcrSave.sha384[pcrIndex];
181                 break;
182     #endif
183     #ifdef TPM_ALG_SHA512
184             case TPM_ALG_SHA512:
185                 return gc.pcrSave.sha512[pcrIndex];
186                 break;
187     #endif
188     #ifdef TPM_ALG_SM3_256
189             case TPM_ALG_SM3_256:
190                 return gc.pcrSave.sm3_256[pcrIndex];
191                 break;
192     #endif
193             default:
194                 break;
195         }
196         FAIL(FATAL_ERROR_INTERNAL);
197     }
```

8.7.3.9 PcrIsAllocated()

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

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

```
198  BOOL
199  PcrIsAllocated(
200      UINT32           pcr,           // IN: The number of the PCR
201      TPMI_ALG_HASH    hashAlg        // IN: The PCR algorithm
202      )
203  {
204      UINT32               i;
205      BOOL                 allocated = FALSE;
206      if(pcr < IMPLEMENTATION_PCR)
207      {
208          for(i = 0; i < gp.pcrAllocated.count; i++)
209          {
210              if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
211              {
212                  if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr / 8])
213                      & (1 << (pcr % 8))) != 0)
214                      allocated = TRUE;
215                  else
216                      allocated = FALSE;
217                  break;
218              }
219          }
220      }
221      return allocated;
222  }
```

8.7.3.10 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

```
223  static BYTE *
224  GetPcrPointer(
225      TPM_ALG_ID       alg,           // IN: algorithm for bank
226      UINT32           pcrNumber      // IN: PCR number
227      )
228  {
229      static BYTE     *pcr = NULL;
230      if(!PcrIsAllocated(pcrNumber, alg))
231          return NULL;
232      switch(alg)
233      {
234  #ifdef TPM_ALG_SHA1
235          case TPM_ALG_SHA1:
236              pcr = s_pcrs[pcrNumber].sha1Pcr;
237              break;
238  #endif
239  #ifdef TPM_ALG_SHA256
240          case TPM_ALG_SHA256:
241              pcr = s_pcrs[pcrNumber].sha256Pcr;
242              break;
243  #endif
244  #ifdef TPM_ALG_SHA384
```

```
245             case TPM_ALG_SHA384:
246                 pcr = s_pcrs[pcrNumber].sha384Pcr;
247                 break;
248     #endif
249     #ifdef TPM_ALG_SHA512
250             case TPM_ALG_SHA512:
251                 pcr = s_pcrs[pcrNumber].sha512Pcr;
252                 break;
253     #endif
254     #ifdef TPM_ALG_SM3_256
255             case TPM_ALG_SM3_256:
256                 pcr = s_pcrs[pcrNumber].sm3_256Pcr;
257                 break;
258     #endif
259             default:
260                 FAIL(FATAL_ERROR_INTERNAL);
261                 break;
262         }
263         return pcr;
264     }
```

8.7.3.11 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

```
265     static BOOL
266     IsPcrSelected(
267         UINT32              pcr,            // IN: The number of the PCR
268         TPMS_PCR_SELECTION  *selection      // IN: The selection structure
269         )
270     {
271         BOOL                selected;
272         selected = (pcr < IMPLEMENTATION_PCR
273                     && ((selection->pcrSelect[pcr / 8]) & (1 << (pcr % 8))) != 0);
274         return selected;
275     }
```

8.7.3.12 FilterPcr()

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

```
276     static void
277     FilterPcr(
278         TPMS_PCR_SELECTION  *selection      // IN: input PCR selection
279         )
280     {
281         UINT32     i;
282         TPMS_PCR_SELECTION      *allocated = NULL;
283         // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
284         for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
285             selection->pcrSelect[i] = 0;
286         // Find the internal configuration for the bank
287         for(i = 0; i < gp.pcrAllocated.count; i++)
288         {
289             if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
290             {
291                 allocated = &gp.pcrAllocated.pcrSelections[i];
```

```
292             break;
293         }
294     }
295     for(i = 0; i < selection->sizeofSelect; i++)
296     {
297         if(allocated == NULL)
298         {
299             // If the required bank does not exist, clear input selection
300             selection->pcrSelect[i] = 0;
301         }
302         else
303             selection->pcrSelect[i] &= allocated->pcrSelect[i];
304     }
305     return;
306 }
```

8.7.3.13 PcrDrtm()

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

```
307 void
308 PcrDrtm(
309     const TPMI_DH_PCR        pcrHandle,     // IN: the index of the PCR to be
310                                             //     modified
311     const TPMI_ALG_HASH      hash,          // IN: the bank identifier
312     const TPM2B_DIGEST      *digest         // IN: the digest to modify the PCR
313     )
314 {
315     BYTE        *pcrData = GetPcrPointer(hash, pcrHandle);
316     if(pcrData != NULL)
317     {
318         // Rest the PCR to zeros
319         MemorySet(pcrData, 0, digest->t.size);
320         // if the TPM has not started, then set the PCR to 0...04 and then extend
321         if(!TPMIsStarted())
322         {
323             pcrData[digest->t.size - 1] = 4;
324         }
325         // Now, extend the value
326         PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
327     }
328 }
```

8.7.3.14 PCR_ClearAuth()

This function is used to reset the PCR authorization values. It is called on TPM2_Startup(CLEAR) and TPM2_Clear().

```
329 void
330 PCR_ClearAuth(
331     void
332     )
333 {
334 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
335     int         j;
336     for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
337     {
338         gc.pcrAuthValues.auth[j].t.size = 0;
339     }
340 #endif
341 }
```

8.7.3.15 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```
342 void
343 PCRStartup(
344     STARTUP_TYPE     type,          // IN: startup type
345     BYTE             locality       // IN: startup locality
346     )
347 {
348     UINT32              pcr, j;
349     UINT32              saveIndex = 0;
350     g_pcrReConfig = FALSE;
351     // Don't test for SU_RESET because that should be the default when nothing
352     // else is selected
353     if(type != SU_RESUME && type != SU_RESTART)
354     {
355         // PCR generation counter is cleared at TPM_RESET
356         gr.pcrCounter = 0;
357     }
358     // Initialize/Restore PCR values
359     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
360     {
361         // On resume, need to know if this PCR had its state saved or not
362         UINT32      stateSaved;
363         if(type == SU_RESUME
364            &&  s_initAttributes[pcr].stateSave == SET)
365         {
366             stateSaved = 1;
367         }
368         else
369         {
370             stateSaved = 0;
371             PCRChanged(pcr);
372         }
373         // If this is the H-CRTM PCR and we are not doing a resume and we
374         // had an H-CRTM event, then we don't change this PCR
375         if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
376             continue;
377         // Iterate each hash algorithm bank
378         for(j = 0; j < gp.pcrAllocated.count; j++)
379         {
380             TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
381             BYTE            *pcrData = GetPcrPointer(hash, pcr);
382             UINT16           pcrSize = CryptHashGetDigestSize(hash);
383             if(pcrData != NULL)
384             {
385                 // if state was saved
386                 if(stateSaved == 1)
387                 {
388                     // Restore saved PCR value
389                     BYTE    *pcrSavedData;
390                     pcrSavedData = GetSavedPcrPointer(
391                         gp.pcrAllocated.pcrSelections[j].hash,
392                         saveIndex);
393                     MemoryCopy(pcrData, pcrSavedData, pcrSize);
394                 }
395                 else
396                     // PCR was not restored by state save
397                 {
398                     // If the reset locality of the PCR is 4, then
399                     // the reset value is all one's, otherwise it is
400                     // all zero.
401                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
402                         MemorySet(pcrData, 0xFF, pcrSize);
```

```
403                     else
404                     {
405                         MemorySet(pcrData, 0, pcrSize);
406                         if(pcr == HCRTM_PCR)
407                             pcrData[pcrSize - 1] = locality;
408                     }
409                 }
410             }
411         }
412         saveIndex += stateSaved;
413     }
414     // Reset authValues on TPM2_Startup(CLEAR)
415     if(type != SU_RESUME)
416         PCR_ClearAuth();
417 }
```

8.7.3.16 PCRStateSave()

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

```
418 void
419 PCRStateSave(
420     TPM_SU           type           // IN: startup type
421     )
422 {
423     UINT32           pcr, j;
424     UINT32           saveIndex = 0;
425     // if state save CLEAR, nothing to be done.  Return here
426     if(type == TPM_SU_CLEAR)
427         return;
428     // Copy PCR values to the structure that should be saved to NV
429     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
430     {
431         UINT32  stateSaved = (s_initAttributes[pcr].stateSave == SET) ? 1 : 0;
432         // Iterate each hash algorithm bank
433         for(j = 0; j < gp.pcrAllocated.count; j++)
434         {
435             BYTE    *pcrData;
436             UINT32  pcrSize;
437             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
438             if(pcrData != NULL)
439             {
440                 pcrSize
441                     = CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
442                 if(stateSaved == 1)
443                 {
444                     // Restore saved PCR value
445                     BYTE    *pcrSavedData;
446                     pcrSavedData
447                         = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
448                                              saveIndex);
449                     MemoryCopy(pcrSavedData, pcrData, pcrSize);
450                 }
451             }
452         }
453         saveIndex += stateSaved;
454     }
455     return;
456 }
```

8.7.3.17 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

```
457  BOOL
458  PCRIsStateSaved(
459      TPMI_DH_PCR      handle         // IN: PCR handle to be extended
460      )
461  {
462      UINT32              pcr = handle - PCR_FIRST;
463      if(s_initAttributes[pcr].stateSave == SET)
464          return TRUE;
465      else
466          return FALSE;
467  }
```

8.7.3.18 PCRIsResetAllowed()

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

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

```
468  BOOL
469  PCRIsResetAllowed(
470      TPMI_DH_PCR      handle         // IN: PCR handle to be extended
471      )
472  {
473      UINT8               commandLocality;
474      UINT8               localityBits = 1;
475      UINT32              pcr = handle - PCR_FIRST;
476      // Check for the locality
477      commandLocality = _plat__LocalityGet();
478  #ifdef DRTM_PCR
479      // For a TPM that does DRTM, Reset is not allowed at locality 4
480      if(commandLocality == 4)
481          return FALSE;
482  #endif
483      localityBits = localityBits << commandLocality;
484      if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
485          return FALSE;
486      else
487          return TRUE;
488  }
```

8.7.3.19 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. Will also bump the counter if the handle is zero which means that PCR 0 can not be in the TCB group. Bump on zero is used by TPM2_Clear().

```
489 void
490 PCRChanged(
491     TPM_HANDLE       pcrHandle      // IN: the handle of the PCR that changed.
492     )
493 {
494     // For the reference implementation, the only change that does not cause
495     // increment is a change to a PCR in the TCB group.
496     if((pcrHandle == 0) || !PCRBelongsTCBGroup(pcrHandle))
497     {
498         gr.pcrCounter++;
499         if(gr.pcrCounter == 0)
500             FAIL(FATAL_ERROR_COUNTER_OVERFLOW);
501     }
502 }
```

8.7.3.20 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

```
503 BOOL
504 PCRIsExtendAllowed(
505     TPMI_DH_PCR      handle         // IN: PCR handle to be extended
506     )
507 {
508     UINT8               commandLocality;
509     UINT8               localityBits = 1;
510     UINT32              pcr = handle - PCR_FIRST;
511     // Check for the locality
512     commandLocality = _plat__LocalityGet();
513     localityBits = localityBits << commandLocality;
514     if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
515         return FALSE;
516     else
517         return TRUE;
518 }
```

8.7.3.21 PCRExtend()

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

```
519 void
520 PCRExtend(
521     TPMI_DH_PCR      handle,        // IN: PCR handle to be extended
522     TPMI_ALG_HASH    hash,          // IN: hash algorithm of PCR
523     UINT32           size,          // IN: size of data to be extended
524     BYTE            *data           // IN: data to be extended
525     )
526 {
527     BYTE               *pcrData;
528     HASH_STATE          hashState;
529     UINT16              pcrSize;
530     pcrData = GetPcrPointer(hash, handle - PCR_FIRST);
531     // Extend PCR if it is allocated
532     if(pcrData != NULL)
533     {
534         pcrSize = CryptHashGetDigestSize(hash);
```

```
535             CryptHashStart(&hashState, hash);
536             CryptDigestUpdate(&hashState, pcrSize, pcrData);
537             CryptDigestUpdate(&hashState, size, data);
538             CryptHashEnd(&hashState, pcrSize, pcrData);
539             // PCR has changed so update the pcrCounter if necessary
540             PCRChanged(handle);
541         }
542         return;
543     }
```

8.7.3.22 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.

```
544     void
545     PCRComputeCurrentDigest(
546         TPMI_ALG_HASH        hashAlg,       // IN: hash algorithm to compute digest
547         TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
548                                             //     output)
549         TPM2B_DIGEST        *digest         // OUT: digest
550         )
551     {
552         HASH_STATE               hashState;
553         TPMS_PCR_SELECTION      *select;
554         BYTE                    *pcrData;   // will point to a digest
555         UINT32                   pcrSize;
556         UINT32                   pcr;
557         UINT32                   i;
558         // Initialize the hash
559         digest->t.size = CryptHashStart(&hashState, hashAlg);
560         pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
561         // Iterate through the list of PCR selection structures
562         for(i = 0; i < selection->count; i++)
563         {
564             // Point to the current selection
565             select = &selection->pcrSelections[i]; // Point to the current selection
566             FilterPcr(select);      // Clear out the bits for unimplemented PCR
567             // Need the size of each digest
568             pcrSize = CryptHashGetDigestSize(selection->pcrSelections[i].hash);
569             // Iterate through the selection
570             for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
571             {
572                 if(IsPcrSelected(pcr, select))         // Is this PCR selected
573                 {
574                     // Get pointer to the digest data for the bank
575                     pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
576                     pAssert(pcrData != NULL);
577                     CryptDigestUpdate(&hashState, pcrSize, pcrData);  // add to digest
578                 }
579             }
580         }
581         // Complete hash stack
582         CryptHashEnd2B(&hashState, &digest->b);
583         return;
584     }
```

8.7.3.23 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.


```
585 void
586 PCRRead(
587     TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
588                                         //     output)
589     TPML_DIGEST         *digest,        // OUT: digest
590     UINT32              *pcrCounter     // OUT: the current value of PCR generation
591                                         //     number
592     )
593 {
594     TPMS_PCR_SELECTION      *select;
595     BYTE                    *pcrData;       // will point to a digest
596     UINT32                   pcr;
597     UINT32                   i;
598     digest->count = 0;
599     // Iterate through the list of PCR selection structures
600     for(i = 0; i < selection->count; i++)
601     {
602         // Point to the current selection
603         select = &selection->pcrSelections[i]; // Point to the current selection
604         FilterPcr(select);      // Clear out the bits for unimplemented PCR
605         // Iterate through the selection
606         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
607         {
608             if(IsPcrSelected(pcr, select))         // Is this PCR selected
609             {
610                 // Check if number of digest exceed upper bound
611                 if(digest->count > 7)
612                 {
613                     // Clear rest of the current select bitmap
614                     while(pcr < IMPLEMENTATION_PCR
615                               // do not round up!
616                           && (pcr / 8) < select->sizeofSelect)
617                     {
618                         // do not round up!
619                         select->pcrSelect[pcr / 8] &= (BYTE)~(1 << (pcr % 8));
620                         pcr++;
621                     }
622                     // Exit inner loop
623                     break;;
624                 }
625                 // Need the size of each digest
626                 digest->digests[digest->count].t.size =
627                     CryptHashGetDigestSize(selection->pcrSelections[i].hash);
628                 // Get pointer to the digest data for the bank
629                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
630                 pAssert(pcrData != NULL);
631                 // Add to the data to digest
632                 MemoryCopy(digest->digests[digest->count].t.buffer,
633                            pcrData,
634                            digest->digests[digest->count].t.size);
635                 digest->count++;
636             }
637         }
638         // If we exit inner loop because we have exceed the output upper bound
639         if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
640         {
641             // Clear rest of the selection
642             while(i < selection->count)
643             {
644                 MemorySet(selection->pcrSelections[i].pcrSelect, 0,
645                           selection->pcrSelections[i].sizeofSelect);
646                 i++;
647             }
648             // exit outer loop
649             break;
650         }
```

```
651         }
652         *pcrCounter = gr.pcrCounter;
653         return;
654     }
```

8.7.3.24 PcrWrite()

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

```
655     void
656     PcrWrite(
657         TPMI_DH_PCR      handle,        // IN: PCR handle to be extended
658         TPMI_ALG_HASH    hash,          // IN: hash algorithm of PCR
659         TPM2B_DIGEST    *digest         // IN: the new value
660         )
661     {
662         UINT32              pcr = handle - PCR_FIRST;
663         BYTE               *pcrData;
664         // Copy value to the PCR if it is allocated
665         pcrData = GetPcrPointer(hash, pcr);
666         if(pcrData != NULL)
667         {
668             MemoryCopy(pcrData, digest->t.buffer, digest->t.size);
669         }
670         return;
671     }
```

8.7.3.25 PCRAllocate()

This function is used to change the PCR allocation.

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

```
672     TPM_RC
673     PCRAllocate(
674         TPML_PCR_SELECTION  *allocate,      // IN: required allocation
675         UINT32              *maxPCR,        // OUT: Maximum number of PCR
676         UINT32              *sizeNeeded,    // OUT: required space
677         UINT32              *sizeAvailable  // OUT: available space
678         )
679     {
680         UINT32                  i, j, k;
681         TPML_PCR_SELECTION      newAllocate;
682         // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
683         BOOL                    pcrHcrtm = FALSE;
684         BOOL                    pcrDrtm = FALSE;
685         // Create the expected new PCR allocation based on the existing allocation
686         // and the new input:
687         //  1. if a PCR bank does not appear in the new allocation, the existing
688         //     allocation of this PCR bank will be preserved.
689         //  2. if a PCR bank appears multiple times in the new allocation, only the
690         //     last one will be in effect.
691         newAllocate = gp.pcrAllocated;
692         for(i = 0; i < allocate->count; i++)
693         {
694             for(j = 0; j < newAllocate.count; j++)
695             {
```

```
696             // If hash matches, the new allocation covers the old allocation
697             // for this particular bank.
698             // The assumption is the initial PCR allocation (from manufacture)
699             // has all the supported hash algorithms with an assigned bank
700             // (possibly empty).  So there must be a match for any new bank
701             // allocation from the input.
702             if(newAllocate.pcrSelections[j].hash ==
703                allocate->pcrSelections[i].hash)
704             {
705                 newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
706                 break;
707             }
708         }
709         // The j loop must exit with a match.
710         pAssert(j < newAllocate.count);
711     }
712     // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
713     *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
714     if(*maxPCR > IMPLEMENTATION_PCR)
715         *maxPCR = IMPLEMENTATION_PCR;
716     // Compute required size for allocation
717     *sizeNeeded = 0;
718     for(i = 0; i < newAllocate.count; i++)
719     {
720         UINT32      digestSize
721             = CryptHashGetDigestSize(newAllocate.pcrSelections[i].hash);
722 #if defined(DRTM_PCR)
723         // Make sure that we end up with at least one DRTM PCR
724         pcrDrtm = pcrDrtm || TestBit(DRTM_PCR,
725                                      newAllocate.pcrSelections[i].pcrSelect,
726                                      newAllocate.pcrSelections[i].sizeofSelect);
727 #else   // if DRTM PCR is not required, indicate that the allocation is OK
728         pcrDrtm = TRUE;
729 #endif
730 #if defined(HCRTM_PCR)
731         // and one HCRTM PCR (since this is usually PCR 0...)
732         pcrHcrtm = pcrHcrtm || TestBit(HCRTM_PCR,
733                                        newAllocate.pcrSelections[i].pcrSelect,
734                                        newAllocate.pcrSelections[i].sizeofSelect);
735 #else
736         pcrHcrtm = TRUE;
737 #endif
738         for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
739         {
740             BYTE        mask = 1;
741             for(k = 0; k < 8; k++)
742             {
743                 if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
744                     *sizeNeeded += digestSize;
745                 mask = mask << 1;
746             }
747         }
748     }
749     if(!pcrDrtm || !pcrHcrtm)
750         return TPM_RC_PCR;
751     // In this particular implementation, we always have enough space to
752     // allocate PCR.  Different implementation may return a sizeAvailable less
753     // than the sizeNeed.
754     *sizeAvailable = sizeof(s_pcrs);
755     // Save the required allocation to NV.  Note that after NV is written, the
756     // PCR allocation in NV is no longer consistent with the RAM data
757     // gp.pcrAllocated.  The NV version reflect the allocate after next
758     // TPM_RESET, while the RAM version reflects the current allocation
759     NV_WRITE_PERSISTENT(pcrAllocated, newAllocate);
760     return TPM_RC_SUCCESS;
761 }
```


8.7.3.26 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.

```
762 void
763 PCRSetValue(
764     TPM_HANDLE       handle,        // IN: the handle of the PCR to set
765     INT8             initialValue   // IN: the value to set
766     )
767 {
768     int              i;
769     UINT32           pcr = handle - PCR_FIRST;
770     TPMI_ALG_HASH    hash;
771     UINT16           digestSize;
772     BYTE            *pcrData;
773     // Iterate supported PCR bank algorithms to reset
774     for(i = 0; i < HASH_COUNT; i++)
775     {
776         hash = CryptHashGetAlgByIndex(i);
777         // Prevent runaway
778         if(hash == TPM_ALG_NULL)
779             break;
780         // Get a pointer to the data
781         pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
782         // If the PCR is allocated
783         if(pcrData != NULL)
784         {
785             // And the size of the digest
786             digestSize = CryptHashGetDigestSize(hash);
787             // Set the LSO to the input value
788             pcrData[digestSize - 1] = initialValue;
789             // Sign extend
790             if(initialValue >= 0)
791                 MemorySet(pcrData, 0, digestSize - 1);
792             else
793                 MemorySet(pcrData, -1, digestSize - 1);
794         }
795     }
796 }
```

8.7.3.27 PCRResetDynamics

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

```
797 void
798 PCRResetDynamics(
799     void
800     )
801 {
802     UINT32              pcr, i;
803     // Initialize PCR values
804     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
805     {
806         // Iterate each hash algorithm bank
807         for(i = 0; i < gp.pcrAllocated.count; i++)
808         {
809             BYTE    *pcrData;
810             UINT32  pcrSize;
811             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
812             if(pcrData != NULL)
813             {
814                 pcrSize =
```

```
815                     CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
816                 // Reset PCR
817                 // Any PCR can be reset by locality 4 should be reset to 0
818                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
819                     MemorySet(pcrData, 0, pcrSize);
820             }
821         }
822     }
823     return;
824 }
```

8.7.3.28 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

```
825 TPMI_YES_NO
826 PCRCapGetAllocation(
827     UINT32               count,         // IN: count of return
828     TPML_PCR_SELECTION  *pcrSelection   // OUT: PCR allocation list
829     )
830 {
831     if(count == 0)
832     {
833         pcrSelection->count = 0;
834         return YES;
835     }
836     else
837     {
838         *pcrSelection = gp.pcrAllocated;
839         return NO;
840     }
841 }
```

8.7.3.29 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```
842 static void
843 PCRSetSelectBit(
844     UINT32           pcr,           // IN: PCR number
845     BYTE            *bitmap         // OUT: bit map to be set
846     )
847 {
848     bitmap[pcr / 8] |= (1 << (pcr % 8));
849     return;
850 }
```

8.7.3.30 PCRGetProperty()

This function returns the selected PCR property.

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

```
851 static BOOL
852 PCRGetProperty(
853     TPM_PT_PCR               property,
854     TPMS_TAGGED_PCR_SELECT  *select
855     )
856 {
857     UINT32              pcr;
858     UINT32              groupIndex;
859     select->tag = property;
860     // Always set the bitmap to be the size of all PCR
861     select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
862     // Initialize bitmap
863     MemorySet(select->pcrSelect, 0, select->sizeofSelect);
864     // Collecting properties
865     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
866     {
867         switch(property)
868         {
869             case TPM_PT_PCR_SAVE:
870                 if(s_initAttributes[pcr].stateSave == SET)
871                     PCRSetSelectBit(pcr, select->pcrSelect);
872                 break;
873             case TPM_PT_PCR_EXTEND_L0:
874                 if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
875                     PCRSetSelectBit(pcr, select->pcrSelect);
876                 break;
877             case TPM_PT_PCR_RESET_L0:
878                 if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
879                     PCRSetSelectBit(pcr, select->pcrSelect);
880                 break;
881             case TPM_PT_PCR_EXTEND_L1:
882                 if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
883                     PCRSetSelectBit(pcr, select->pcrSelect);
884                 break;
885             case TPM_PT_PCR_RESET_L1:
886                 if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
887                     PCRSetSelectBit(pcr, select->pcrSelect);
888                 break;
889             case TPM_PT_PCR_EXTEND_L2:
890                 if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
891                     PCRSetSelectBit(pcr, select->pcrSelect);
892                 break;
893             case TPM_PT_PCR_RESET_L2:
894                 if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
895                     PCRSetSelectBit(pcr, select->pcrSelect);
896                 break;
897             case TPM_PT_PCR_EXTEND_L3:
898                 if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
899                     PCRSetSelectBit(pcr, select->pcrSelect);
900                 break;
901             case TPM_PT_PCR_RESET_L3:
902                 if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
903                     PCRSetSelectBit(pcr, select->pcrSelect);
904                 break;
905             case TPM_PT_PCR_EXTEND_L4:
906                 if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
907                     PCRSetSelectBit(pcr, select->pcrSelect);
908                 break;
909             case TPM_PT_PCR_RESET_L4:
```



```
910                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
911                     PCRSetSelectBit(pcr, select->pcrSelect);
912                 break;
913             case TPM_PT_PCR_DRTM_RESET:
914                 // DRTM reset PCRs are the PCR reset by locality 4
915                 if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
916                     PCRSetSelectBit(pcr, select->pcrSelect);
917                 break;
918 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
919             case TPM_PT_PCR_POLICY:
920                 if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
921                     PCRSetSelectBit(pcr, select->pcrSelect);
922                 break;
923 #endif
924 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
925             case TPM_PT_PCR_AUTH:
926                 if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
927                     PCRSetSelectBit(pcr, select->pcrSelect);
928                 break;
929 #endif
930 #if ENABLE_PCR_NO_INCREMENT == YES
931             case TPM_PT_PCR_NO_INCREMENT:
932                 if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
933                     PCRSetSelectBit(pcr, select->pcrSelect);
934                 break;
935 #endif
936             default:
937                 // If property is not supported, stop scanning PCR attributes
938                 // and return.
939                 return FALSE;
940                 break;
941         }
942     }
943     return TRUE;
944 }
```

8.7.3.31 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

```
945 TPMI_YES_NO
946 PCRCapGetProperties(
947     TPM_PT_PCR                  property,       // IN: the starting PCR property
948     UINT32                      count,          // IN: count of returned properties
949     TPML_TAGGED_PCR_PROPERTY    *select         // OUT: PCR select
950     )
951 {
952     TPMI_YES_NO     more = NO;
953     UINT32          i;
954     // Initialize output property list
955     select->count = 0;
956     // The maximum count of properties we may return is MAX_PCR_PROPERTIES
957     if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
958     // TPM_PT_PCR_FIRST is defined as 0 in spec.  It ensures that property
959     // value would never be less than TPM_PT_PCR_FIRST
960     cAssert(TPM_PT_PCR_FIRST == 0);
961     // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
962     // implemented on the TPM.
```

```
963         for(i = property; i <= TPM_PT_PCR_LAST; i++)
964         {
965             if(select->count < count)
966             {
967                 // If we have not filled up the return list, add more properties to it
968                 if(PCRGetProperty(i, &select->pcrProperty[select->count]))
969                     // only increment if the property is implemented
970                     select->count++;
971             }
972             else
973             {
974                 // If the return list is full but we still have properties
975                 // available, report this and stop iterating.
976                 more = YES;
977                 break;
978             }
979         }
980         return more;
981     }
```

8.7.3.32 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

```
982     TPMI_YES_NO
983     PCRCapGetHandles(
984         TPMI_DH_PCR      handle,         // IN: start handle
985         UINT32           count,          // IN: count of returned handles
986         TPML_HANDLE     *handleList      // OUT: list of handle
987         )
988     {
989         TPMI_YES_NO     more = NO;
990         UINT32          i;
991         pAssert(HandleGetType(handle) == TPM_HT_PCR);
992         // Initialize output handle list
993         handleList->count = 0;
994         // The maximum count of handles we may return is MAX_CAP_HANDLES
995         if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
996         // Iterate PCR handle range
997         for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
998         {
999             if(handleList->count < count)
1000            {
1001                // If we have not filled up the return list, add this PCR
1002                // handle to it
1003                handleList->handle[handleList->count] = i + PCR_FIRST;
1004                handleList->count++;
1005            }
1006            else
1007            {
1008                // If the return list is full but we still have PCR handle
1009                // available, report this and stop iterating
1010                more = YES;
1011                break;
1012            }
1013        }
1014        return more;
```

```
1015    }
```

8.8 PP.c

8.8.1 Introduction

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

8.8.2 Includes

```
#include "Tpm.h"
```

8.8.3 Functions

8.8.3.1 PhysicalPresencePreInstall_Init()

This function is used to initialize the array of commands that always 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.

When set, these cannot be cleared.

```
void
PhysicalPresencePreInstall_Init(
    void
    )
{
    COMMAND_INDEX        commandIndex;
    // Clear all the PP commands
    MemorySet(&gp.ppList, 0, sizeof(gp.ppList));
    // Any command that is PP_REQUIRED should be SET
    for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
    {
        if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED
           &&  s_commandAttributes[commandIndex] & PP_REQUIRED)
            SET_BIT(commandIndex, gp.ppList);
    }
    // Write PP list to NV
    NV_SYNC_PERSISTENT(ppList);
    return;
}
```

8.8.3.2 PhysicalPresenceCommandSet()

This function is used to set the indicator that a command requires PP confirmation.

```
void
PhysicalPresenceCommandSet(
    TPM_CC           commandCode    // IN: command code
    )
{
    COMMAND_INDEX       commandIndex = CommandCodeToCommandIndex(commandCode);
    // if the command isn't implemented, the do nothing
    if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
        return;
    // only set the bit if this is a command for which PP is allowed
    if(s_commandAttributes[commandIndex] & PP_COMMAND)
        SET_BIT(commandIndex, gp.ppList);
    return;
}
```


8.8.3.3 PhysicalPresenceCommandClear()

This function is used to clear the indicator that a command requires PP confirmation.

```
35  void
36  PhysicalPresenceCommandClear(
37      TPM_CC           commandCode    // IN: command code
38      )
39  {
40      COMMAND_INDEX       commandIndex = CommandCodeToCommandIndex(commandCode);
41      // If the command isn't implemented, then don't do anything
42      if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
43          return;
44      // Only clear the bit if the command does not require PP
45      if((s_commandAttributes[commandIndex] & PP_REQUIRED) == 0)
46          CLEAR_BIT(commandIndex, gp.ppList);
47      return;
48  }
```

8.8.3.4 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

```
49  BOOL
50  PhysicalPresenceIsRequired(
51      COMMAND_INDEX    commandIndex   // IN: command index
52      )
53  {
54      // Check the bit map.  If the bit is SET, PP authorization is required
55      return (TEST_BIT(commandIndex, gp.ppList));
56  }
```

8.8.3.5 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

```
57  TPMI_YES_NO
58  PhysicalPresenceCapGetCCList(
59      TPM_CC           commandCode,   // IN: start command code
60      UINT32           count,         // IN: count of returned TPM_CC
61      TPML_CC         *commandList    // OUT: list of TPM_CC
62      )
63  {
64      TPMI_YES_NO     more = NO;
65      COMMAND_INDEX   commandIndex;
66      // Initialize output handle list
67      commandList->count = 0;
68      // The maximum count of command we may return is MAX_CAP_CC
69      if(count > MAX_CAP_CC) count = MAX_CAP_CC;
```



```
70      // Collect PP commands
71      for(commandIndex = GetClosestCommandIndex(commandCode);
72      commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
73          commandIndex = GetNextCommandIndex(commandIndex))
74      {
75          if(PhysicalPresenceIsRequired(commandIndex))
76          {
77              if(commandList->count < count)
78              {
79                  // If we have not filled up the return list, add this command
80                  // code to it
81                  commandList->commandCodes[commandList->count]
82                      = GetCommandCode(commandIndex);
83                  commandList->count++;
84              }
85              else
86              {
87                  // If the return list is full but we still have PP command
88                  // available, report this and stop iterating
89                  more = YES;
90                  break;
91              }
92          }
93      }
94      return more;
95  }
```


8.9 Session.c

8.9.1 Introduction

The code in this file is used to manage the session context counter. The scheme implemented here is a "truncated counter". This scheme allows the TPM to not need TPM_SU_CLEAR for a very long period of time and still not have the context count for a session repeated. The counter *(contextCounter*)in this implementation is a UINT64 but can be smaller. The "tracking array" *(contextArray*) only has 16-bits per context. The tracking array is the data that needs to be saved and restored across TPM_SU_STATE so that sessions are not lost when the system enters the sleep state. Also, when the TPM is active, the tracking array is kept in RAM making it important that the number of bytes for each entry be kept as small as possible. The TPM prevents **collisions** of these truncated values by not allowing a *contextID* to be assigned if it would be the same as an existing value. Since the array holds 16 bits, after a context has been saved, an additional 2^16-1 contexts may be saved before the count would again match. The normal expectation is that the context will be flushed before its count value is needed again but it is always possible to have long-lived sessions. The *contextID* is assigned when the context is saved (TPM2_ContextSave()). At that time, the TPM will compare the low-order 16 bits of *contextCounter* to the existing values in *contextArray* and if one matches, the TPM will return TPM_RC_CONTEXT_GAP (by construction, the entry that contains the matching value is the oldest context). The expected remediation by the TRM is to load the oldest saved session context (the one found by the TPM), and save it. Since loading the oldest session also eliminates its *contextID* value from *contextArray*, there TPM will always be able to load and save the oldest existing context. In the worst case, software may have to load and save several contexts in order to save an additional one. This should happen very infrequently. When the TPM searches *contextArray* and finds that none of the *contextIDs* match the low-order 16-bits of *contextCount*, the TPM can copy the low bits to the *contextArray* associated with the session, and increment *contextCount*. There is one entry in *contextArray* for each of the active sessions allowed by the TPM implementation. This array contains either a context count, an index, or a value indicating the slot is available (0). The index into the *contextArray* is the handle for the session with the region selector byte of the session set to zero. If an entry in *contextArray* contains 0, then the corresponding handle may be assigned to a session. If the entry contains a value that is less than or equal to the number of loaded sessions for the TPM, then the array entry is the slot in which the context is loaded.

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

NOTE: When the TPM firmware determines that the array entry is for a loaded session, it will subtract 1 to create the zero-based slot number. There is one significant corner case in this scheme. When the *contextCount* is equal to a value in the *contextArray*, the oldest session needs to be recycled or flushed. In order to recycle the session, it must be loaded. To be loaded, there must be an available slot. Rather than require that a spare slot be available all the time, the TPM will check to see if the *contextCount* is equal to some value in the *contextArray* when a session is created. This prevents the last session slot from being used when it is likely that a session will need to be recycled. If a TPM with both 1.2 and 2.0 functionality uses this scheme for both 1.2 and 2.0 sessions, and the list of active contexts is read with TPM_GetCapabiltiy(), the TPM will create 32-bit representations of the list that contains 16-bit values (the TPM2_GetCapability() returns a list of handles for active sessions rather than a list of *contextID*). The full *contextID* has high-order bits that are either the same as the current *contextCount* or one less. It is one less if the 16-bits of the *contextArray* has a value that is larger than the low-order 16 bits of *contextCount*.

8.9.2 Includes, Defines, and Local Variables

```
1   #define SESSION_C
2   #include "Tpm.h"
```

8.9.3 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.

```
3   static void
4   ContextIdSetOldest(
5       void
6       )
7   {
8       CONTEXT_SLOT    lowBits;
9       CONTEXT_SLOT    entry;
10      CONTEXT_SLOT    smallest = ((CONTEXT_SLOT)~0);
11      UINT32  i;
12      // Set oldestSaveContext to a value indicating none assigned
13      s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
14      lowBits = (CONTEXT_SLOT)gr.contextCounter;
15      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
16      {
17          entry = gr.contextArray[i];
18          // only look at entries that are saved contexts
19          if(entry > MAX_LOADED_SESSIONS)
20          {
21              // Use a less than or equal in case the oldest
22              // is brand new (= lowBits-1) and equal to our initial
23              // value for smallest.
24              if(((CONTEXT_SLOT)(entry - lowBits)) <= smallest)
25              {
26                  smallest = (entry - lowBits);
27                  s_oldestSavedSession = i;
28              }
29          }
30      }
31      // When we finish, either the s_oldestSavedSession still has its initial
32      // value, or it has the index of the oldest saved context.
33  }
```

8.9.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```
34  void
35  SessionStartup(
36      STARTUP_TYPE     type
37      )
38  {
39      UINT32                  i;
40      // Initialize session slots.  At startup, all the in-memory session slots
41      // are cleared and marked as not occupied
42      for(i = 0; i < MAX_LOADED_SESSIONS; i++)
43          s_sessions[i].occupied = FALSE;   // session slot is not occupied
44      // The free session slots the number of maximum allowed loaded sessions
45      s_freeSessionSlots = MAX_LOADED_SESSIONS;
46      // Initialize context ID data.  On a ST_SAVE or hibernate sequence, it will
47      // scan the saved array of session context counts, and clear any entry that
48      // references a session that was in memory during the state save since that
49      // memory was not preserved over the ST_SAVE.
50      if(type == SU_RESUME || type == SU_RESTART)
51      {
52          // On ST_SAVE we preserve the contexts that were saved but not the ones
53          // in memory
54          for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
```

```
55          {
56              // If the array value is unused or references a loaded session then
57              // that loaded session context is lost and the array entry is
58              // reclaimed.
59              if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
60                  gr.contextArray[i] = 0;
61          }
62          // Find the oldest session in context ID data and set it in
63          // s_oldestSavedSession
64          ContextIdSetOldest();
65      }
66      else
67      {
68          // For STARTUP_CLEAR, clear out the contextArray
69          for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
70              gr.contextArray[i] = 0;
71          // reset the context counter
72          gr.contextCounter = MAX_LOADED_SESSIONS + 1;
73          // Initialize oldest saved session
74          s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
75      }
76      return;
77  }
```

8.9.5 Access Functions

8.9.5.1 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

```
78  BOOL
79  SessionIsLoaded(
80      TPM_HANDLE       handle         // IN: session handle
81      )
82  {
83      pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
84              || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
85      handle = handle & HR_HANDLE_MASK;
86      // if out of range of possible active session, or not assigned to a loaded
87      // session return false
88      if(handle >= MAX_ACTIVE_SESSIONS
89         || gr.contextArray[handle] == 0
90         || gr.contextArray[handle] > MAX_LOADED_SESSIONS)
91          return FALSE;
92      return TRUE;
93  }
```

8.9.5.2 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;
}
```

8.9.5.3 SequenceNumbereForSavedContextIsValid()

```
BOOL
SequenceNumbereForSavedContextIsValid(
    TPMS_CONTEXT    *context        // IN: pointer to a context structure to be
                                    //     validated
    )
{
#define MAX_CONTEXT_GAP ((UINT64)((CONTEXT_SLOT) ~0) + 1)
    TPM_HANDLE          handle = context->savedHandle & HR_HANDLE_MASK;
    if(// Handle must be with the range of active sessions
       handle >= MAX_ACTIVE_SESSIONS
       // the array entry must be for a saved context
       || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
           // the array entry must agree with the sequence number
       || gr.contextArray[handle] != (CONTEXT_SLOT)context->sequence
           // the provided sequence number has to be less than the current counter
       || context->sequence > gr.contextCounter
           // but not so much that it could not be a valid sequence number
       || gr.contextCounter - context->sequence > MAX_CONTEXT_GAP)
        return FALSE;
    return TRUE;
}
```

8.9.5.4 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

```
132 BOOL
133 SessionPCRValueIsCurrent(
134     SESSION         *session        // IN: session structure
135     )
136 {
137     if(session->pcrCounter != 0
138        && session->pcrCounter != gr.pcrCounter
139        )
140         return FALSE;
141     else
142         return TRUE;
143 }
```

8.9.5.5 SessionGet()

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

The function requires that the session is loaded.

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

8.9.6 Utility Functions

8.9.6.1 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

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

8.9.6.2 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

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

```
261             session->attributes.isTrialPolicy = SET;
262         SessionSetStartTime(session);
263         // Initialize policyDigest.  policyDigest is initialized with a string of 0
264         // of session algorithm digest size. Since the session is already clear.
265         // Just need to set the size
266         session->u2.policyDigest.t.size =
267             CryptHashGetDigestSize(session->authHashAlg);
268     }
269     // Create initial session nonce
270     session->nonceTPM.t.size = nonceCaller->t.size;
271     CryptRandomGenerate(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
272     MemoryCopy2B(&nonceTpm->b, &session->nonceTPM.b,
273                  sizeof(nonceTpm->t.buffer));
274     // Set up session parameter encryption algorithm
275     session->symmetric = *symmetric;
276     // If there is a bind object or a session secret, then need to compute
277     // a sessionKey.
278     if(bind != TPM_RH_NULL || seed->t.size != 0)
279     {
280         // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
281         //                   nonceCaller, bits)
282         // The HMAC key for generating the sessionSecret can be the concatenation
283         // of an authorization value and a seed value
284         TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
285         TPM2B_KEY           key;
286         // Get hash size, which is also the length of sessionKey
287         session->sessionKey.t.size = CryptHashGetDigestSize(session->authHashAlg);
288         // Get authValue of associated entity
289         EntityGetAuthValue(bind, (TPM2B_AUTH *)&key);
290         pAssert(key.t.size + seed->t.size <= sizeof(key.t.buffer));
291         // Concatenate authValue and seed
292         MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));
293         // Compute the session key
294         CryptKDFa(session->authHashAlg, &key.b, SESSION_KEY, &session->nonceTPM.b,
295                   &nonceCaller->b,
296                   session->sessionKey.t.size * 8, session->sessionKey.t.buffer,
297                   NULL, FALSE);
298     }
299     // Copy the name of the entity that the HMAC session is bound to
300     // Policy session is not bound to an entity
301     if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
302     {
303         session->attributes.isBound = SET;
304         SessionComputeBoundEntity(bind, &session->u1.boundEntity);
305     }
306     // If there is a bind object and it is subject to DA, then use of this session
307     // is subject to DA regardless of how it is used.
308     session->attributes.isDaBound = (bind != TPM_RH_NULL)
309         && (IsDAExempted(bind) == FALSE);
310 // If the session is bound, then check to see if it is bound to lockoutAuth
311     session->attributes.isLockoutBound = (session->attributes.isDaBound == SET)
312         && (bind == TPM_RH_LOCKOUT);
313     return TPM_RC_SUCCESS;
314 }
```

8.9.6.3 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

```
315 TPM_RC
316 SessionContextSave(
317     TPM_HANDLE           handle,        // IN: session handle
318     CONTEXT_COUNTER     *contextID      // OUT: assigned contextID
319     )
320 {
321     UINT32                      contextIndex;
322     CONTEXT_SLOT                slotIndex;
323     pAssert(SessionIsLoaded(handle));
324     // check to see if the gap is already maxed out
325     // Need to have a saved session
326     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
327           // if the oldest saved session has the same value as the low bits
328           // of the contextCounter, then the GAP is maxed out.
329        && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
330         return TPM_RC_CONTEXT_GAP;
331     // if the caller wants the context counter, set it
332     if(contextID != NULL)
333         *contextID = gr.contextCounter;
334     contextIndex = handle & HR_HANDLE_MASK;
335     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
336     // Extract the session slot number referenced by the contextArray
337     // because we are going to overwrite this with the low order
338     // contextID value.
339     slotIndex = gr.contextArray[contextIndex] - 1;
340     // Set the contextID for the contextArray
341     gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
342     // Increment the counter
343     gr.contextCounter++;
344     // In the unlikely event that the 64-bit context counter rolls over...
345     if(gr.contextCounter == 0)
346     {
347         // back it up
348         gr.contextCounter--;
349         // return an error
350         return TPM_RC_TOO_MANY_CONTEXTS;
351     }
352     // if the low-order bits wrapped, need to advance the value to skip over
353     // the values used to indicate that a session is loaded
354     if(((CONTEXT_SLOT)gr.contextCounter) == 0)
355         gr.contextCounter += MAX_LOADED_SESSIONS + 1;
356     // If no other sessions are saved, this is now the oldest.
357     if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
358         s_oldestSavedSession = contextIndex;
359     // Mark the session slot as unoccupied
360     s_sessions[slotIndex].occupied = FALSE;
361     // and indicate that there is an additional open slot
362     s_freeSessionSlots++;
363     return TPM_RC_SUCCESS;
364 }
```

8.9.6.4 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

```
365  TPM_RC
366  SessionContextLoad(
367      SESSION_BUF     *session,       // IN: session structure from saved context
368      TPM_HANDLE      *handle         // IN/OUT: session handle
369      )
370  {
371      UINT32              contextIndex;
372      CONTEXT_SLOT        slotIndex;
373      pAssert(HandleGetType(*handle) == TPM_HT_POLICY_SESSION
374              || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
375      // Don't bother looking if no openings
376      if(s_freeSessionSlots == 0)
377          return TPM_RC_SESSION_MEMORY;
378      // Find a free session slot to load the session
379      for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
380          if(s_sessions[slotIndex].occupied == FALSE) break;
381      // if no spot found, then this is an internal error
382      pAssert(slotIndex < MAX_LOADED_SESSIONS);
383      contextIndex = *handle & HR_HANDLE_MASK;   // extract the index
384      // If there is only one slot left, and the gap is at maximum, the only session
385      // context that we can safely load is the oldest one.
386      if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
387         && s_freeSessionSlots == 1
388         && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
389         && contextIndex != s_oldestSavedSession)
390          return TPM_RC_CONTEXT_GAP;
391      pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
392      // set the contextArray value to point to the session slot where
393      // the context is loaded
394      gr.contextArray[contextIndex] = slotIndex + 1;
395      // if this was the oldest context, find the new oldest
396      if(contextIndex == s_oldestSavedSession)
397          ContextIdSetOldest();
398      // Copy session data to session slot
399      MemoryCopy(&s_sessions[slotIndex].session, session, sizeof(SESSION));
400             // Set session slot as occupied
401      s_sessions[slotIndex].occupied = TRUE;
402      // Reduce the number of open spots
403      s_freeSessionSlots--;
404      return TPM_RC_SUCCESS;
405  }
```

8.9.6.5 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.

```
406  void
407  SessionFlush(
408      TPM_HANDLE       handle         // IN: loaded or saved session handle
409      )
410  {
411      CONTEXT_SLOT         slotIndex;
412      UINT32               contextIndex;   // Index into contextArray
```

```
413         pAssert((HandleGetType(handle) == TPM_HT_POLICY_SESSION
414                  || HandleGetType(handle) == TPM_HT_HMAC_SESSION
415                  )
416                 && (SessionIsLoaded(handle) || SessionIsSaved(handle))
417                 );
418              // Flush context ID of this session
419              // Convert handle to an index into the contextArray
420         contextIndex = handle & HR_HANDLE_MASK;
421         pAssert(contextIndex < sizeof(gr.contextArray) / sizeof(gr.contextArray[0]));
422         // Get the current contents of the array
423         slotIndex = gr.contextArray[contextIndex];
424         // Mark context array entry as available
425         gr.contextArray[contextIndex] = 0;
426         // Is this a saved session being flushed
427         if(slotIndex > MAX_LOADED_SESSIONS)
428         {
429             // Flushing the oldest session?
430             if(contextIndex == s_oldestSavedSession)
431                 // If so, find a new value for oldest.
432                 ContextIdSetOldest();
433         }
434         else
435         {
436             // Adjust slot index to point to session array index
437             slotIndex -= 1;
438             // Free session array index
439             s_sessions[slotIndex].occupied = FALSE;
440             s_freeSessionSlots++;
441         }
442         return;
443     }
```

8.9.6.6 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.

```
444     void
445     SessionComputeBoundEntity(
446         TPMI_DH_ENTITY       entityHandle,  // IN: handle of entity
447         TPM2B_NAME          *bind           // OUT: binding value
448         )
449     {
450         TPM2B_AUTH           auth;
451         BYTE                *pAuth = auth.t.buffer;
452         UINT16               i;
453         // Get name
454         EntityGetName(entityHandle, bind);
455     //    // The bound value of a reserved handle is the handle itself
456     //    if(bind->t.size == sizeof(TPM_HANDLE)) return;
457         // For all the other entities, concatenate the authorization value to the name.
458         // Get a local copy of the authorization value because some overlapping
459         // may be necessary.
460         EntityGetAuthValue(entityHandle, &auth);
461         // Make sure that the extra space is zeroed
462         MemorySet(&bind->t.name[bind->t.size], 0, sizeof(bind->t.name) - bind->t.size);
463         // XOR the authValue at the end of the name
464         for(i = sizeof(bind->t.name) - auth.t.size; i < sizeof(bind->t.name); i++)
465             bind->t.name[i] ^= *pAuth++;
466         // Set the bind value to the maximum size
467         bind->t.size = sizeof(bind->t.name);
468         return;
```

```
469 }
```

8.9.6.7 SessionSetStartTime()

This function is used to initialize the session timing

```
470 void
471 SessionSetStartTime(
472     SESSION         *session        // IN: the session to update
473     )
474 {
475     session->startTime = g_time;
476     session->epoch = g_timeEpoch;
477     session->timeout = 0;
478 }
```

8.9.6.8 SessionResetPolicyData()

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

```
479 void
480 SessionResetPolicyData(
481     SESSION         *session        // IN: the session to reset
482     )
483 {
484     SESSION_ATTRIBUTES      oldAttributes;
485     pAssert(session != NULL);
486     // Will need later
487     oldAttributes = session->attributes;
488     // No command
489     session->commandCode = 0;
490     // No locality selected
491     MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
492     // The cpHash size to zero
493     session->u1.cpHash.b.size = 0;
494     // No timeout
495     session->timeout = 0;
496     // Reset the pcrCounter
497     session->pcrCounter = 0;
498     // Reset the policy hash
499     MemorySet(&session->u2.policyDigest.t.buffer, 0,
500               session->u2.policyDigest.t.size);
501     // Reset the session attributes
502     MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
503     // Restore the policy attributes
504     session->attributes.isPolicy = SET;
505     session->attributes.isTrialPolicy = oldAttributes.isTrialPolicy;
506     // Restore the bind attributes
507     session->attributes.isDaBound = oldAttributes.isDaBound;
508     session->attributes.isLockoutBound = oldAttributes.isLockoutBound;
509 }
```

8.9.6.9 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

```
510 TPMI_YES_NO
511 SessionCapGetLoaded(
512     TPMI_SH_POLICY   handle,        // IN: start handle
513     UINT32           count,         // IN: count of returned handles
514     TPML_HANDLE     *handleList     // OUT: list of handle
515     )
516 {
517     TPMI_YES_NO     more = NO;
518     UINT32          i;
519     pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
520     // Initialize output handle list
521     handleList->count = 0;
522     // The maximum count of handles we may return is MAX_CAP_HANDLES
523     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
524     // Iterate session context ID slots to get loaded session handles
525     for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
526     {
527         // If session is active
528         if(gr.contextArray[i] != 0)
529         {
530             // If session is loaded
531             if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
532             {
533                 if(handleList->count < count)
534                 {
535                     SESSION         *session;
536                     // If we have not filled up the return list, add this
537                     // session handle to it
538                     // assume that this is going to be an HMAC session
539                     handle = i + HMAC_SESSION_FIRST;
540                     session = SessionGet(handle);
541                     if(session->attributes.isPolicy)
542                         handle = i + POLICY_SESSION_FIRST;
543                     handleList->handle[handleList->count] = handle;
544                     handleList->count++;
545                 }
546                 else
547                 {
548                     // If the return list is full but we still have loaded object
549                     // available, report this and stop iterating
550                     more = YES;
551                     break;
552                 }
553             }
554         }
555     }
556     return more;
557 }
```

8.9.6.10 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

```
558  TPMI_YES_NO
559  SessionCapGetSaved(
560      TPMI_SH_HMAC     handle,         // IN: start handle
561      UINT32           count,          // IN: count of returned handles
562      TPML_HANDLE     *handleList      // OUT: list of handle
563      )
564  {
565      TPMI_YES_NO     more = NO;
566      UINT32          i;
567  #ifdef  TPM_HT_SAVED_SESSION
568      pAssert(HandleGetType(handle) == TPM_HT_SAVED_SESSION);
569  #else
570      pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);
571  #endif
572      // Initialize output handle list
573      handleList->count = 0;
574      // The maximum count of handles we may return is MAX_CAP_HANDLES
575      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
576      // Iterate session context ID slots to get loaded session handles
577      for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
578      {
579          // If session is active
580          if(gr.contextArray[i] != 0)
581          {
582              // If session is saved
583              if(gr.contextArray[i] > MAX_LOADED_SESSIONS)
584              {
585                  if(handleList->count < count)
586                  {
587                      // If we have not filled up the return list, add this
588                      // session handle to it
589                      handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
590                      handleList->count++;
591                  }
592                  else
593                  {
594                      // If the return list is full but we still have loaded object
595                      // available, report this and stop iterating
596                      more = YES;
597                      break;
598                  }
599              }
600          }
601      }
602      return more;
603  }
```

8.9.6.11 SessionCapGetLoadedNumber()

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

```
604  UINT32
605  SessionCapGetLoadedNumber(
606      void
607      )
608  {
609      return MAX_LOADED_SESSIONS - s_freeSessionSlots;
610  }
```


8.9.6.12 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.

```
611 UINT32
612 SessionCapGetLoadedAvail(
613     void
614     )
615 {
616     return s_freeSessionSlots;
617 }
```

8.9.6.13 SessionCapGetActiveNumber()

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

```
618 UINT32
619 SessionCapGetActiveNumber(
620     void
621     )
622 {
623     UINT32              i;
624     UINT32              num = 0;
625     // Iterate the context array to find the number of non-zero slots
626     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
627     {
628         if(gr.contextArray[i] != 0) num++;
629     }
630     return num;
631 }
```

8.9.6.14 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.

```
632 UINT32
633 SessionCapGetActiveAvail(
634     void
635     )
636 {
637     UINT32              i;
638     UINT32              num = 0;
639     // Iterate the context array to find the number of zero slots
640     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
641     {
642         if(gr.contextArray[i] == 0) num++;
643     }
644     return num;
645 }
```


8.10 Time.c

8.10.1 Introduction

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

8.10.2 Includes

```
#include "Tpm.h"
#include "PlatformData.h"
```

8.10.3 Functions

8.10.3.1 TimePowerOn()

This function initialize time info at _TPM_Init().

This function is called at _TPM_Init() so that the TPM time can start counting as soon as the TPM comes out of reset and doesn't have to wait until TPM2_Startup() in order to begin the new time epoch. This could be significant for systems that could get powered up but not run any TPM commands for some period of time.

```
void
TimePowerOn(
    void
    )
{
    // If the timer was reset or stopped, we need a new epoch
    if(_plat__TimerWasReset())
    {
        g_timeNewEpochNeeded = TRUE;
        // If the timer was reset, need to reset the base time of the TPM. By
        // resetting to zero here, the TPM can capture the time that passed between
        // when the system timer was reset and when the first call is made to
        // _plat__TimeRead().
        g_time = 0;
        // And reset the DA timers
        DAInit();
    }
}
```

8.10.3.2 TimeNewEpoch()

This function does the processing to generate a new time epoch nonce and set NV for update. This function is only called when NV is known to be available and the clock is running. The epoch is updated to persistent data.

```
static void
TimeNewEpoch(
    void
    )
{
#ifdef CLOCK_STOPS
    CryptRandomGenerate(sizeof(CLOCK_NONCE), (BYTE *)&g_timeEpoch);
#else
    // if the epoch is kept in NV, update it.
    gp.timeEpoch++;
```

```
31      NV_SYNC_PERSISTENT(timeEpoch);
32  #endif
33      g_timeNewEpochNeeded = FALSE;
34      // Clean out any lingering state
35      _plat__TimerWasStopped();
36  }
```

8.10.3.3 TimeStartup()

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

This function will deal with the deferred creation of a new epoch. TimeUpdateToCurrent() will not start a new epoch even if one is due when TPM_Startup() has not been run. This is because the state of NV is not known until startup completes. When Startup is done, then it will create the epoch nonce to complete the initializations by calling this function.

```
37  void
38  TimeStartup(
39      STARTUP_TYPE     type           // IN: start up type
40      )
41  {
42      NOT_REFERENCED(type);
43      // If the previous cycle is orderly shut down, the value of the safe bit
44      // the same as previously saved.  Otherwise, it is not safe.
45      if(!NV_IS_ORDERLY)
46          go.clockSafe = NO;
47      // Before Startup, the TPM will not do clock updates. At startup, need to
48      // do a time update.
49      TimeUpdate();
50      return;
51  }
```

8.10.3.4 TimeClockUpdate()

This function updates go.clock. If *newTime* requires an update of NV, then NV is checked for availability. If it is not available or is rate limiting, then go.clock is not updated and the function returns an error. If *newTime* would not cause an NV write, then go.clock is updated. If an NV write occurs, then go.safe is SET.

Error Returns	Meaning
TPM_RC_NV_RATE	NV cannot be written because it is rate limiting
TPM_RC_NV_UNAVAILABLE	NV cannot be accessed

```
52  TPM_RC
53  TimeClockUpdate(
54      UINT64           newTime
55      )
56  {
57  #define CLOCK_UPDATE_MASK  ((1ULL << NV_CLOCK_UPDATE_INTERVAL)- 1)
58      // Check to see if the update will cause a need for an nvClock update
59      if((newTime | CLOCK_UPDATE_MASK) > (go.clock | CLOCK_UPDATE_MASK))
60      {
61          RETURN_IF_NV_IS_NOT_AVAILABLE;
62          // Going to update the NV time state so SET the safe flag
63          go.clockSafe = YES;
64          // update the time
65          go.clock = newTime;
66          NvWrite(NV_ORDERLY_DATA, sizeof(go), &go);
67      }
```



```
68          else
69              // No NV update needed so just update
70              go.clock = newTime;
71          return TPM_RC_SUCCESS;
72      }
```

8.10.3.5 TimeUpdate()

This function is used to update the time and clock values. If the TPM has run TPM2_Startup(), this function is called at the start of each command. If the TPM has not run TPM2_Startup(), this is called from TPM2_Startup() to get the clock values initialized. It is not called on command entry because, in this implementation, the go structure is not read from NV until TPM2_Startup(). The reason for this is that the initialization code (_TPM_Init()) may run before NV is accessible.

```
73      void
74      TimeUpdate(
75          void
76          )
77      {
78              UINT64          elapsed;
79      //
80          if(g_timeNewEpochNeeded)
81              TimeNewEpoch();
82          // Get the difference between this call and the last time we updated the tick
83          // timer.
84          elapsed = _plat__TimerRead() - g_time;
85          g_time += elapsed;
86          // Don't need to check the result because it has to be success because have
87          // already checked that NV is available.
88          TimeClockUpdate(go.clock + elapsed);
89          // Call self healing logic for dictionary attack parameters
90          DASelfHeal();
91      }
```

8.10.3.6 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 implementation does not do rate limiting. If the implementation does do rate limiting, then the *Clock* update should not be inhibited even when doing rate limiting.

```
92      void
93      TimeUpdateToCurrent(
94          void
95          )
96      {
97          UINT64          elapsed;
98      //
99          // Can't update time during the dark interval or when rate limiting so don't
100         // make any modifications to the internal clock value
101         if(!NV_IS_AVAILABLE)
102             return;
103         // Make sure that we consume the current _plat__TimerWasStopped() state.
104         g_timeNewEpochNeeded |= _plat__TimerWasStopped();
105         // If we need a new epoch but the TPM has not started, don't generate the new
```

```
106         // epoch here because the crypto has not been initialized by TPM2_Startup().
107         // Instead, just continue and let TPM2_Startup() processing create the
108         // new epoch if needed.
109         if(g_timeNewEpochNeeded && TPMIsStarted())
110         {
111             TimeNewEpoch();
112         }
113         // Get the difference between this call and the last time we updated the tick
114         // timer.
115         elapsed = _plat__TimerRead() - g_time;
116         g_time += elapsed;
117         // Don't need to check the result because it has to be success because have
118         // already checked that NV is available.
119         TimeClockUpdate(go.clock + elapsed);
120         // Call self healing logic for dictionary attack parameters
121         DASelfHeal();
122         return;
123     }
```

8.10.3.7 TimeSetAdjustRate()

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

```
124     void
125     TimeSetAdjustRate(
126         TPM_CLOCK_ADJUST     adjust         // IN: adjust constant
127         )
128     {
129         switch(adjust)
130         {
131             case TPM_CLOCK_COARSE_SLOWER:
132                 _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
133                 break;
134             case TPM_CLOCK_COARSE_FASTER:
135                 _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
136                 break;
137             case TPM_CLOCK_MEDIUM_SLOWER:
138                 _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
139                 break;
140             case TPM_CLOCK_MEDIUM_FASTER:
141                 _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
142                 break;
143             case TPM_CLOCK_FINE_SLOWER:
144                 _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
145                 break;
146             case TPM_CLOCK_FINE_FASTER:
147                 _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
148                 break;
149             case TPM_CLOCK_NO_CHANGE:
150                 break;
151             default:
152                 FAIL(FATAL_ERROR_INTERNAL);
153                 break;
154         }
155         return;
156     }
```

8.10.3.8 TimeGetMarshaled()

This function is used to access TPMS_TIME_INFO in canonical form. The function collects the time information and marshals it into *dataBuffer* and returns the marshaled size

Return Value	Meaning

```
157   UINT16
158   TimeGetMarshaled(
159       TIME_INFO       *dataBuffer     // OUT: result buffer
160       )
161   {
162       TPMS_TIME_INFO      timeInfo;
163       // Fill TPMS_TIME_INFO structure
164       timeInfo.time = g_time;
165       TimeFillInfo(&timeInfo.clockInfo);
166       // Marshal TPMS_TIME_INFO to canonical form
167       return TPMS_TIME_INFO_Marshal(&timeInfo, (BYTE **)&dataBuffer, NULL);
168   }
```

8.10.3.9 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```
169   void
170   TimeFillInfo(
171       TPMS_CLOCK_INFO     *clockInfo
172       )
173   {
174       clockInfo->clock = go.clock;
175       clockInfo->resetCount = gp.resetCount;
176       clockInfo->restartCount = gr.restartCount;
177       // If NV is not available, clock stopped advancing and the value reported is
178       // not "safe".
179       if(NV_IS_AVAILABLE)
180           clockInfo->safe = go.clockSafe;
181       else
182           clockInfo->safe = NO;
183       return;
184   }
```


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 "Tpm.h"
typedef struct
{
    TPM_ALG_ID          algID;
    TPMA_ALGORITHM      attributes;
} ALGORITHM;
static const ALGORITHM    s_algorithms[] =
{
// The entries in this table need to be in ascending order but the table doesn't
// need to be full (gaps are allowed). One day, a tool might exist to fill in the
// table from the TPM_ALG description
#ifdef TPM_ALG_RSA
    {TPM_ALG_RSA,           {1, 0, 0, 1, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_TDES
    {TPM_ALG_TDES,          {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
{TPM_ALG_HMAC,          {0, 0, 1, 0, 0, 1, 0, 0, 0}},
#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_SM3_256
    {TPM_ALG_SM3_256,       {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SM4
    {TPM_ALG_SM4,          {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_RSASSA
    {TPM_ALG_RSASSA,        {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_RSAES
    {TPM_ALG_RSAES,         {1, 0, 0, 0, 0, 0, 1, 0, 0}},
```

```
52  #endif
53  #ifdef TPM_ALG_RSAPSS
54      {TPM_ALG_RSAPSS,        {1, 0, 0, 0, 0, 1, 0, 0, 0}},
55  #endif
56  #ifdef TPM_ALG_OAEP
57      {TPM_ALG_OAEP,          {1, 0, 0, 0, 0, 0, 1, 0, 0}},
58  #endif
59  #ifdef TPM_ALG_ECDSA
60      {TPM_ALG_ECDSA,         {1, 0, 0, 0, 0, 1, 0, 1, 0}},
61  #endif
62  #ifdef TPM_ALG_ECDH
63      {TPM_ALG_ECDH,          {1, 0, 0, 0, 0, 0, 0, 1, 0}},
64  #endif
65  #ifdef TPM_ALG_ECDAA
66      {TPM_ALG_ECDAA,         {1, 0, 0, 0, 0, 1, 0, 0, 0}},
67  #endif
68  #ifdef TPM_ALG_SM2
69      {TPM_ALG_SM2,           {1, 0, 0, 0, 0, 1, 0, 1, 0}},
70  #endif
71  #ifdef TPM_ALG_ECSCHNORR
72      {TPM_ALG_ECSCHNORR,     {1, 0, 0, 0, 0, 1, 0, 0, 0}},
73  #endif
74  #ifdef TPM_ALG_ECMQV
75      {TPM_ALG_ECMQV,         {1, 0, 0, 0, 0, 0, 0, 1, 0}},
76  #endif
77  #ifdef TPM_ALG_KDF1_SP800_56A
78      {TPM_ALG_KDF1_SP800_56A,{0, 0, 1, 0, 0, 0, 0, 1, 0}},
79  #endif
80  #ifdef TPM_ALG_KDF2
81      {TPM_ALG_KDF2,          {0, 0, 1, 0, 0, 0, 0, 1, 0}},
82  #endif
83  #ifdef TPM_ALG_KDF1_SP800_108
84      {TPM_ALG_KDF1_SP800_108,{0, 0, 1, 0, 0, 0, 0, 1, 0}},
85  #endif
86  #ifdef TPM_ALG_ECC
87      {TPM_ALG_ECC,           {1, 0, 0, 1, 0, 0, 0, 0, 0}},
88  #endif
89      {TPM_ALG_SYMCIPHER,     {0, 0, 0, 1, 0, 0, 0, 0, 0}},
90  #ifdef TPM_ALG_CAMELLIA
91      {TPM_ALG_CAMELLIA,      {0, 1, 0, 0, 0, 0, 0, 0, 0}},
92  #endif
93  #ifdef TPM_ALG_CTR
94      {TPM_ALG_CTR,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
95  #endif
96  #ifdef TPM_ALG_OFB
97      {TPM_ALG_OFB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
98  #endif
99  #ifdef TPM_ALG_CBC
100     {TPM_ALG_CBC,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
101 #endif
102 #ifdef TPM_ALG_CFB
103     {TPM_ALG_CFB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
104 #endif
105 #ifdef TPM_ALG_ECB
106     {TPM_ALG_ECB,           {0, 1, 0, 0, 0, 0, 1, 0, 0}},
107 #endif
108 };
```

9.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2_GetCapability() to return a list of the implemented algorithms.

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

```
109  TPMI_YES_NO
110  AlgorithmCapGetImplemented(
111      TPM_ALG_ID                   algID,     // IN: the starting algorithm ID
112      UINT32                       count,     // IN: count of returned algorithms
113      TPML_ALG_PROPERTY           *algList    // OUT: algorithm list
114      )
115  {
116      TPMI_YES_NO     more = NO;
117      UINT32          i;
118      UINT32          algNum;
119      // initialize output algorithm list
120      algList->count = 0;
121      // The maximum count of algorithms we may return is MAX_CAP_ALGS.
122      if(count > MAX_CAP_ALGS)
123          count = MAX_CAP_ALGS;
124      // Compute how many algorithms are defined in s_algorithms array.
125      algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
126      // Scan the implemented algorithm list to see if there is a match to 'algID'.
127      for(i = 0; i < algNum; i++)
128      {
129          // If algID is less than the starting algorithm ID, skip it
130          if(s_algorithms[i].algID < algID)
131              continue;
132          if(algList->count < count)
133          {
134              // If we have not filled up the return list, add more algorithms
135              // to it
136              algList->algProperties[algList->count].alg = s_algorithms[i].algID;
137              algList->algProperties[algList->count].algProperties =
138                  s_algorithms[i].attributes;
139              algList->count++;
140          }
141          else
142          {
143              // If the return list is full but we still have algorithms
144              // available, report this and stop scanning.
145              more = YES;
146              break;
147          }
148      }
149      return more;
150  }
151  LIB_EXPORT
152  void
153  AlgorithmGetImplementedVector(
154      ALGORITHM_VECTOR    *implemented    // OUT: the implemented bits are SET
155      )
156  {
157      int                      index;
158      // Nothing implemented until we say it is
159      MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));
160      for(index = (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
161      index >= 0;
162          index--)
163          SET_BIT(s_algorithms[index].algID, *implemented);
164      return;
165  }
```

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 "Tpm.h"
```

9.2.3 Functions

9.2.3.1 TestBit()

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

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

9.2.3.2 SetBit()

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

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


9.2.3.3 ClearBit()

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

```
26  #ifndef INLINE_FUNCTIONS
27  void
28  ClearBit(
29      unsigned int     bitNum,        // IN: number of the bit in 'bArray'.
30      BYTE            *bArray,        // IN: array containing the bits
31      unsigned int     bytesInArray   // IN: size in bytes of 'bArray'
32      )
33  {
34      pAssert(bytesInArray > (bitNum >> 3));
35      bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
36  }
37  #endif // INLINE_FUNCTIONS
```


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 "CommandCodeAttributes_fp.h"
```

Set the default value for CC_VEND if not already set

```
#ifndef CC_VEND
#define     CC_VEND     (TPM_CC)(0x20000000)
#endif
typedef UINT16          ATTRIBUTE_TYPE;
```

The following file is produced from the command tables in part 3 of the specification. It defines the attributes for each of the commands.

NOTE: This file is currently produced by an automated process. Files produced from Part 2 or Part 3 tables through automated processes are not included in the specification so that their is no ambiguity about the table containing the information being the normative definition.

```
#define _COMMAND_CODE_ATTRIBUTES_
#include    "CommandAttributeData.h"
```

9.3.3 Command Attribute Functions

9.3.3.1 NextImplementedIndex()

This function is used when the lists are not compressed. In a compressed list, only the implemented commands are present. So, a search might find a value but that value may not be implemented. This function checks to see if the input *commandIndex* points to an implemented command and, if not, it searches upwards until it finds one. When the list is compressed, this function gets defined as a no-op.

```
#ifndef COMPRESSED_LISTS
static COMMAND_INDEX
NextImplementedIndex(
    COMMAND_INDEX       commandIndex
    )
{
    for(;commandIndex < COMMAND_COUNT; commandIndex++)
    {
        if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
            return commandIndex;
    }
    return UNIMPLEMENTED_COMMAND_INDEX;
}
#else
#define NextImplementedIndex(x) (x)
#endif
```

9.3.3.2 GetClosestCommandIndex()

This function returns the command index for the command with a value that is equal to or greater than the input value

```
25  COMMAND_INDEX
26  GetClosestCommandIndex(
27      TPM_CC           commandCode    // IN: the command code to start at
28      )
29  {
30      BOOL                vendor = (commandCode & CC_VEND) != 0;
31      COMMAND_INDEX       searchIndex = (COMMAND_INDEX)commandCode;
32      // The commandCode is a UINT32 and the search index is UINT16. We are going to
33      // search for a match but need to make sure that the commandCode value is not
34      // out of range. To do this, need to clear the vendor bit of the commandCode
35      // (if set) and compare the result to the 16-bit searchIndex value. If it is
36      // out of range, indicate that the command is not implemented
37      if((commandCode & ~CC_VEND) != searchIndex)
38          return UNIMPLEMENTED_COMMAND_INDEX;
39      // if there is at least one vendor command, the last entry in the array will
40      // have the v bit set. If the input commandCode is larger than the last
41      // vendor-command, then it is out of range.
42      if(vendor)
43      {
44  #if VENDOR_COMMAND_ARRAY_SIZE > 0
45          COMMAND_INDEX       commandIndex;
46          COMMAND_INDEX       min;
47          COMMAND_INDEX       max;
48          int                 diff;
49  #if LIBRARY_COMMAND_ARRAY_SIZE == COMMAND_COUNT
50  #error "Constants are not consistent."
51  #endif
52          // Check to see if the value is equal to or below the minimum
53          // entry.
54          // Note: Put this check first so that the typical case of only one vendor-
55          // specific command doesn't waste any more time.
56          if(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE].commandIndex >= searchIndex)
57          {
58              // the vendor array is always assumed to be packed so there is
59              // no need to check to see if the command is implemented
60              return LIBRARY_COMMAND_ARRAY_SIZE;
61          }
62          // See if this is out of range on the top
63          if(s_ccAttr[COMMAND_COUNT - 1].commandIndex < searchIndex)
64          {
65              return UNIMPLEMENTED_COMMAND_INDEX;
66          }
67          commandIndex = UNIMPLEMENTED_COMMAND_INDEX; // Needs initialization to keep
68                                                      // compiler happy
69          min = LIBRARY_COMMAND_ARRAY_SIZE;       // first vendor command
70          max = COMMAND_COUNT - 1;                // last vendor command
71          diff = 1;                               // needs initialization to keep
72                                                  // compiler happy
73          while(min <= max)
74          {
75              commandIndex = (min + max + 1) / 2;
76              diff = s_ccAttr[commandIndex].commandIndex - searchIndex;
77              if(diff == 0)
78                  return commandIndex;
79              if(diff > 0)
80                  max = commandIndex - 1;
81              else
82                  min = commandIndex + 1;
83          }
84          // didn't find and exact match. commandIndex will be pointing at the last
```

```
85          // item tested. If diff is positive, then the last item tested was
86          // larger index of the command code so it is the smallest value
87          // larger than the requested value.
88          if(diff > 0)
89              return commandIndex;
90          // if diff is negative, then the value tested was smaller than
91          // the commandCode index and the next higher value is the correct one.
92          // Note: this will necessarily be in range because of the earlier check
93          // that the index was within range.
94          return commandIndex + 1;
95  #else
96      // If there are no vendor commands so anything with the vendor bit set is out
97      // of range
98          return UNIMPLEMENTED_COMMAND_INDEX;
99  #endif
100     }
101     // Get here if the V-Bit was not set in 'commandCode'
102     if(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE - 1].commandIndex < searchIndex)
103     {
104         // requested index is out of the range to the top
105 #if VENDOR_COMMAND_ARRAY_SIZE > 0
106         // If there are vendor commands, then the first vendor command
107         // is the next value greater than the commandCode.
108         // NOTE: we got here if the starting index did not have the V bit but we
109         // reached the end of the array of library commands (non-vendor). Since
110         // there is at least one vendor command, and vendor commands are always
111         // in a compressed list that starts after the library list, the next
112         // index value contains a valid vendor command.
113         return LIBRARY_COMMAND_ARRAY_SIZE;
114 #else
115         // if there are no vendor commands, then this is out of range
116         return UNIMPLEMENTED_COMMAND_INDEX;
117 #endif
118     }
119     // If the request is lower than any value in the array, then return
120     // the lowest value (needs to be an index for an implemented command
121     if(s_ccAttr[0].commandIndex >= searchIndex)
122     {
123         return NextImplementedIndex(0);
124     }
125     else
126     {
127 #ifdef COMPRESSED_LISTS
128         COMMAND_INDEX       commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
129         COMMAND_INDEX       min = 0;
130         COMMAND_INDEX       max = LIBRARY_COMMAND_ARRAY_SIZE - 1;
131         int                 diff = 1;
132 #if LIBRARY_COMMAND_ARRAY_SIZE == 0
133 #error  "Something is terribly wrong"
134 #endif
135         // The s_ccAttr array contains an extra entry at the end (a zero value).
136         // Don't count this as an array entry. This means that max should start
137         // out pointing to the last valid entry in the array which is - 2
138         pAssert(max == (sizeof(s_ccAttr) / sizeof(TPMA_CC)
139                         - VENDOR_COMMAND_ARRAY_SIZE - 2));
140         while(min <= max)
141         {
142             commandIndex = (min + max + 1) / 2;
143             diff = s_ccAttr[commandIndex].commandIndex - searchIndex;
144             if(diff == 0)
145                 return commandIndex;
146             if(diff > 0)
147                 max = commandIndex - 1;
148             else
149                 min = commandIndex + 1;
150         }
```



```
151             // didn't find and exact match. commandIndex will be pointing at the
152             // last item tested. If diff is positive, then the last item tested was
153             // larger index of the command code so it is the smallest value
154             // larger than the requested value.
155             if(diff > 0)
156                 return commandIndex;
157             // if diff is negative, then the value tested was smaller than
158             // the commandCode index and the next higher value is the correct one.
159             // Note: this will necessarily be in range because of the earlier check
160             // that the index was within range.
161             return commandIndex + 1;
162     #else
163             // The list is not compressed so offset into the array by the command
164             // code value of the first entry in the list. Then go find the first
165             // implemented command.
166             return NextImplementedIndex(searchIndex
167                                         - (COMMAND_INDEX)s_ccAttr[0].commandIndex);
168     #endif
169         }
170     }
```

9.3.3.3 CommandCodeToComandIndex()

This function returns the index in the various attributes arrays of the command.

Return Value	Meaning
UNIMPLEMNED_COMMAND_INDEX	command is not implemented
other	index of the command

```
171     COMMAND_INDEX
172     CommandCodeToCommandIndex(
173         TPM_CC           commandCode    // IN: the command code to look up
174         )
175     {
176         COMMAND_INDEX       searchIndex = (COMMAND_INDEX)commandCode;
177         BOOL                vendor = (commandCode & CC_VEND) != 0;
178         COMMAND_INDEX       commandIndex;
179     #if !defined COMPRESSED_LISTS
180         if(!vendor)
181         {
182             commandIndex = searchIndex - (COMMAND_INDEX)s_ccAttr[0].commandIndex;
183             // Check for out of range or unimplemented.
184             // Note, since a COMMAND_INDEX is unsigned, if searchIndex is smaller than
185             // the lowest value of command, it will become a 'negative' number making
186             // it look like a large unsigned number, this will cause it to fail
187             // the unsigned check below.
188             if(commandIndex >= LIBRARY_COMMAND_ARRAY_SIZE
189                || (s_commandAttributes[commandIndex] & IS_IMPLEMENTED) == 0)
190                 return UNIMPLEMENTED_COMMAND_INDEX;
191             return commandIndex;
192         }
193     #endif
194         // Need this code for any vendor code lookup or for compressed lists
195         commandIndex = GetClosestCommandIndex(commandCode);
196         // Look at the returned value from get closest. If it isn't the one that was
197         // requested, then the command is not implemented.
198         if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
199         {
200             if((s_ccAttr[commandIndex].commandIndex != searchIndex)
201                || ((s_ccAttr[commandIndex].V == SET) && !vendor)
202                || ((s_ccAttr[commandIndex].V == CLEAR) && vendor))
203                 commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
```

```
204         }
205         return commandIndex;
206     }
```

9.3.3.4 GetNextCommandIndex()

This function returns the index of the next implemented command.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	no more implemented commands
other	the index of the next implemented command

```
207     COMMAND_INDEX
208     GetNextCommandIndex(
209         COMMAND_INDEX    commandIndex   // IN: the starting index
210         )
211     {
212         while(++commandIndex < COMMAND_COUNT)
213         {
214     #if !defined COMPRESSED_LISTS
215             if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
216     #endif
217                 return commandIndex;
218         }
219         return UNIMPLEMENTED_COMMAND_INDEX;
220     }
```

9.3.3.5 GetCommandCode()

This function returns the *commandCode* associated with the command index

```
221     TPM_CC
222     GetCommandCode(
223         COMMAND_INDEX    commandIndex   // IN: the command index
224         )
225     {
226         TPM_CC              commandCode = s_ccAttr[commandIndex].commandIndex;
227         if(s_ccAttr[commandIndex].V)
228             commandCode += CC_VEND;
229         return commandCode;
230     }
```

9.3.3.6 CommandAuthRole()

This function returns the authorization role required of a handle.

Return Value	Meaning
AUTH_NONE	no authorization is required
AUTH_USER	user role authorization is required
AUTH_ADMIN	admin role authorization is required
AUTH_DUP	duplication role authorization is required

```
231     AUTH_ROLE
232     CommandAuthRole(
233         COMMAND_INDEX    commandIndex,  // IN: command index
234         UINT32           handleIndex    // IN: handle index (zero based)
```

```
235         )
236     {
237         if(0 == handleIndex)
238         {
239             // Any authorization role set?
240             COMMAND_ATTRIBUTES  properties = s_commandAttributes[commandIndex];
241             if(properties & HANDLE_1_USER)
242                 return AUTH_USER;
243             if(properties & HANDLE_1_ADMIN)
244                 return AUTH_ADMIN;
245             if(properties & HANDLE_1_DUP)
246                 return AUTH_DUP;
247         }
248         else if(1 == handleIndex)
249         {
250             if(s_commandAttributes[commandIndex] & HANDLE_2_USER)
251                 return AUTH_USER;
252         }
253         return AUTH_NONE;
254     }
```

9.3.3.7 EncryptSize()

This function returns the size of the decrypt size field. This function returns 0 if encryption is not allowed

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
255     #ifndef INLINE_FUNCTIONS
256     int
257     EncryptSize(
258         COMMAND_INDEX    commandIndex   // IN: command index
259         )
260     {
261         return ((s_commandAttributes[commandIndex] & ENCRYPT_2) ? 2 :
262                 (s_commandAttributes[commandIndex] & ENCRYPT_4) ? 4 : 0);
263     }
264     #endif // INLINE_FUNCTIONS
```

9.3.3.8 DecryptSize()

This function returns the size of the decrypt size field. This function returns 0 if decryption is not allowed

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
265     #ifndef INLINE_FUNCTIONS
266     int
267     DecryptSize(
268         COMMAND_INDEX    commandIndex   // IN: command index
269         )
270     {
271         return ((s_commandAttributes[commandIndex] & DECRYPT_2) ? 2 :
272                 (s_commandAttributes[commandIndex] & DECRYPT_4) ? 4 : 0);
```

```
}
#endif // INLINE_FUNCTIONS
```

9.3.3.9 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

```
#ifndef INLINE_FUNCTIONS
BOOL
IsSessionAllowed(
    COMMAND_INDEX    commandIndex   // IN: the command to be checked
    )
{
    return ((s_commandAttributes[commandIndex] & NO_SESSIONS) == 0);
}
#endif // INLINE_FUNCTIONS
```

9.3.3.10 IsHandleInResponse()

This function determines if a command has a handle in the response

```
#ifndef INLINE_FUNCTIONS
BOOL
IsHandleInResponse(
    COMMAND_INDEX    commandIndex
    )
{
    return ((s_commandAttributes[commandIndex] & R_HANDLE) != 0);
}
#endif // INLINE_FUNCTIONS
```

9.3.3.11 IsWriteOperation()

Checks to see if an operation will write to an NV Index and is subject to being blocked by read-lock

```
BOOL
IsWriteOperation(
    COMMAND_INDEX    commandIndex   // IN: Command to check
    )
{
#ifdef  WRITE_LOCK
    return ((s_commandAttributes[commandIndex] & WRITE_LOCK) != 0);
#else
    if(!s_ccAttr[commandIndex].V)
    {
        switch(s_ccAttr[commandIndex].commandIndex)
        {
            case TPM_CC_NV_Write:
            case TPM_CC_NV_Increment:
            case TPM_CC_NV_SetBits:
            case TPM_CC_NV_Extend:
            // NV write lock counts as a write operation for authorization purposes.
            // We check to see if the NV is write locked before we do the
            // authorization. If it is locked, we fail the command early.
```

```
            case TPM_CC_NV_WriteLock:
                return TRUE;
            default:
                break;
        }
    }
    return FALSE;
#endif
}
```

9.3.3.12 IsReadOperation()

Checks to see if an operation will write to an NV Index and is subject to being blocked by write-lock.

```
BOOL
IsReadOperation(
    COMMAND_INDEX    commandIndex   // IN: Command to check
    )
{
#ifdef  READ_LOCK
    return ((s_commandAttributes[commandIndex] & READ_LOCK) != 0);
#else
    if(!s_ccAttr[commandIndex].V)
    {
        switch(s_ccAttr[commandIndex].commandIndex)
        {
            case TPM_CC_NV_Read:
            case TPM_CC_PolicyNV:
            case TPM_CC_NV_Certify:
            // NV read lock counts as a read operation for authorization purposes.
            // We check to see if the NV is read locked before we do the
            // authorization. If it is locked, we fail the command early.
            case TPM_CC_NV_ReadLock:
                return TRUE;
            default:
                break;
        }
    }
    return FALSE;
#endif
}
```

9.3.3.13 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;
    COMMAND_INDEX    commandIndex;
```

```
358         // initialize output handle list count
359         commandList->count = 0;
360         for(commandIndex = GetClosestCommandIndex(commandCode);
361         commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
362             commandIndex = GetNextCommandIndex(commandIndex))
363         {
364     #ifndef COMPRESSED_LISTS
365             // this check isn't needed for compressed lists.
366             if(!(s_commandAttributes[commandIndex] & IS_IMPLEMENTED))
367                 continue;
368     #endif
369             if(commandList->count < count)
370             {
371                 // If the list is not full, add the attributes for this command.
372                 commandList->commandAttributes[commandList->count]
373                     = s_ccAttr[commandIndex];
374                 commandList->count++;
375             }
376             else
377             {
378                 // If the list is full but there are more commands to report,
379                 // indicate this and return.
380                 more = YES;
381                 break;
382             }
383         }
384         return more;
385     }
```

9.3.3.14 IsVendorCommand()

Function indicates if a command index references a vendor command.

Return Value	Meaning
TRUE	command is a vendor command
FALSE	command is not a vendor command

```
386     #ifndef INLINE_FUNCTIONS
387     BOOL
388     IsVendorCommand(
389         COMMAND_INDEX    commandIndex   // IN: command index to check
390         )
391     {
392         return (s_ccAttr[commandIndex].V == SET);
393     }
394     #endif // INLINE_FUNCTIONS
```

9.4 Entity.c

9.4.1 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.

9.4.2 Includes

```
#include "Tpm.h"
```

9.4.3 Functions

9.4.3.1 EntityGetLoadStatus()

This function will check that all the handles access loaded entities.

Error Returns	Meaning
TPM_RC_HANDLE	handle type does not match
TPM_RC_REFERENCE_Hx()	entity is not present
TPM_RC_HIERARCHY	entity belongs to a disabled hierarchy
TPM_RC_OBJECT_MEMORY	handle is an evict object but there is no space to load it to RAM

```
TPM_RC
EntityGetLoadStatus(
    COMMAND         *command        // IN/OUT: command parsing structure
    )
{
    UINT32              i;
    TPM_RC              result = TPM_RC_SUCCESS;
//
    for(i = 0; i < command->handleNum; i++)
    {
        TPM_HANDLE      handle = command->handles[i];
        switch(HandleGetType(handle))
        {
            // For handles associated with hierarchies, the entity is present
            // only if the associated enable is SET.
            case TPM_HT_PERMANENT:
                switch(handle)
                {
                    case TPM_RH_OWNER:
                        if(!gc.shEnable)
                            result = TPM_RC_HIERARCHY;
                        break;
#ifdef  VENDOR_PERMANENT
                    case VENDOR_PERMANENT:
#endif
                    case TPM_RH_ENDORSEMENT:
                        if(!gc.ehEnable)
                            result = TPM_RC_HIERARCHY;
                        break;
                    case TPM_RH_PLATFORM:
                        if(!g_phEnable)
                            result = TPM_RC_HIERARCHY;
                        break;
                    // null handle, PW session handle and lockout
```



```
                        // handle are always available
                    case TPM_RH_NULL:
                    case TPM_RS_PW:
                        // Need to be careful for lockout. Lockout is always available
                        // for policy checks but not always available when authValue
                        // is being checked.
                    case TPM_RH_LOCKOUT:
                        break;
                    default:
                        // handling of the manufacture_specific handles
                        if(((TPM_RH)handle >= TPM_RH_AUTH_00)
                           && ((TPM_RH)handle <= TPM_RH_AUTH_FF))
                           // use the value that would have been returned from
                           // unmarshaling if it did the handle filtering
                            result = TPM_RC_VALUE;
                        else
                            FAIL(FATAL_ERROR_INTERNAL);
                        break;
                }
                break;
            case TPM_HT_TRANSIENT:
                // For a transient object, check if the handle is associated
                // with a loaded object.
                if(!IsObjectPresent(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_REFERENCE_H0 error may be returned by
                // ObjectLoadEvict()
                result = ObjectLoadEvict(&command->handles[i], command->index);
                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.
                FAIL(FATAL_ERROR_INTERNAL);
                break;
        }
        if(result != TPM_RC_SUCCESS)
        {
            if(result == TPM_RC_REFERENCE_H0)
                result = result + i;
            else
                result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
            break;
        }
    }
    return result;
}
```

9.4.3.2 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	Meaning
count	number of bytes in the *authValue* with 0's stripped

```
UINT16
EntityGetAuthValue(
    TPMI_DH_ENTITY   handle,        // IN: handle of entity
    TPM2B_AUTH      *auth           // OUT: authValue of the entity
    )
{
    TPM2B_AUTH      *pAuth = NULL;
    auth->t.size = 0;
    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
        {
            switch(handle)
            {
                case TPM_RH_OWNER:
                    // ownerAuth for TPM_RH_OWNER
                    pAuth = &gp.ownerAuth;
                    break;
                case TPM_RH_ENDORSEMENT:
                    // endorsementAuth for TPM_RH_ENDORSEMENT
                    pAuth = &gp.endorsementAuth;
                    break;
                case TPM_RH_PLATFORM:
                    // platformAuth for TPM_RH_PLATFORM
                    pAuth = &gc.platformAuth;
                    break;
                case TPM_RH_LOCKOUT:
```



```
152                     // lockoutAuth for TPM_RH_LOCKOUT
153                     pAuth = &gp.lockoutAuth;
154                     break;
155                 case TPM_RH_NULL:
156                     // nullAuth for TPM_RH_NULL. Return 0 directly here
157                     return 0;
158                     break;
159 #ifdef  VENDOR_PERMANENT
160                 case VENDOR_PERMANENT:
161                     // vendor authorization value
162                     pAauth = &g_platformUniqueDetails;
163 #endif
164                 default:
165                     // If any other permanent handle is present it is
166                     // a code defect.
167                     FAIL(FATAL_ERROR_INTERNAL);
168                     break;
169             }
170             break;
171         }
172         case TPM_HT_TRANSIENT:
173             // authValue for an object
174             // A persistent object would have been copied into RAM
175             // and would have an transient object handle here.
176         {
177             OBJECT          *object;
178             object = HandleToObject(handle);
179             // special handling if this is a sequence object
180             if(ObjectIsSequence(object))
181             {
182                 pAuth = &((HASH_OBJECT *)object)->auth;
183             }
184             else
185             {
186                 // Authorization is available only when the private portion of
187                 // the object is loaded.  The check should be made before
188                 // this function is called
189                 pAssert(object->attributes.publicOnly == CLEAR);
190                 pAuth = &object->sensitive.authValue;
191             }
192         }
193         break;
194         case TPM_HT_NV_INDEX:
195             // authValue for an NV index
196         {
197             NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
198             pAssert(nvIndex != NULL);
199             pAuth = &nvIndex->authValue;
200         }
201         break;
202         case TPM_HT_PCR:
203             // authValue for PCR
204             pAuth = PCRGetAuthValue(handle);
205             break;
206         default:
207             // If any other handle type is present here, then there is a defect
208             // in the unmarshaling code.
209             FAIL(FATAL_ERROR_INTERNAL);
210             break;
211     }
212     // Copy the authValue
213     MemoryCopy2B(&auth->b, &pAuth->b, sizeof(auth->t.buffer));
214     MemoryRemoveTrailingZeros(auth);
215     return auth->t.size;
216 }
```


9.4.3.3 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.

```
217  TPMI_ALG_HASH
218  EntityGetAuthPolicy(
219      TPMI_DH_ENTITY   handle,        // IN: handle of entity
220      TPM2B_DIGEST    *authPolicy     // OUT: authPolicy of the entity
221      )
222  {
223      TPMI_ALG_HASH       hashAlg = TPM_ALG_NULL;
224      authPolicy->t.size = 0;
225      switch(HandleGetType(handle))
226      {
227          case TPM_HT_PERMANENT:
228              switch(handle)
229              {
230                  case TPM_RH_OWNER:
231                      // ownerPolicy for TPM_RH_OWNER
232                      *authPolicy = gp.ownerPolicy;
233                      hashAlg = gp.ownerAlg;
234                      break;
235                  case TPM_RH_ENDORSEMENT:
236                      // endorsementPolicy for TPM_RH_ENDORSEMENT
237                      *authPolicy = gp.endorsementPolicy;
238                      hashAlg = gp.endorsementAlg;
239                      break;
240                  case TPM_RH_PLATFORM:
241                      // platformPolicy for TPM_RH_PLATFORM
242                      *authPolicy = gc.platformPolicy;
243                      hashAlg = gc.platformAlg;
244                      break;
245                  case TPM_RH_LOCKOUT:
246                      // lockoutPolicy for TPM_RH_LOCKOUT
247                      *authPolicy = gp.lockoutPolicy;
248                      hashAlg = gp.lockoutAlg;
249                      break;
250                  default:
251                      return TPM_ALG_ERROR;
252                      break;
253              }
254              break;
255          case TPM_HT_TRANSIENT:
256              // authPolicy for an object
257              {
258                  OBJECT *object = HandleToObject(handle);
259                  *authPolicy = object->publicArea.authPolicy;
260                  hashAlg = object->publicArea.nameAlg;
261              }
262              break;
263              case TPM_HT_NV_INDEX:
264                  // authPolicy for a NV index
265                  {
266                      NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
267                      pAssert(nvIndex != 0);
268                      *authPolicy = nvIndex->publicArea.authPolicy;
269                      hashAlg = nvIndex->publicArea.nameAlg;
270                  }
```



```
271             break;
272             case TPM_HT_PCR:
273                 // authPolicy for a PCR
274                 hashAlg = PCRGetAuthPolicy(handle, authPolicy);
275                 break;
276             default:
277                 // If any other handle type is present it is a code defect.
278                 FAIL(FATAL_ERROR_INTERNAL);
279                 break;
280         }
281         return hashAlg;
282     }
```

9.4.3.4 EntityGetName()

This function returns the Name associated with a handle.

```
283     TPM2B_NAME *
284     EntityGetName(
285         TPMI_DH_ENTITY   handle,        // IN: handle of entity
286         TPM2B_NAME      *name           // OUT: name of entity
287         )
288     {
289         switch(HandleGetType(handle))
290         {
291             case TPM_HT_TRANSIENT:
292             {
293                 // Name for an object
294                 OBJECT      *object = HandleToObject(handle);
295                 // an object with no nameAlg has no name
296                 if(object->publicArea.nameAlg == TPM_ALG_NULL)
297                     name->b.size = 0;
298                 else
299                     *name = object->name;
300                 break;
301             }
302             case TPM_HT_NV_INDEX:
303                 // Name for a NV index
304                 NvGetNameByIndexHandle(handle, name);
305                 break;
306             default:
307                 // For all other types, the handle is the Name
308                 name->t.size = sizeof(TPM_HANDLE);
309                 UINT32_TO_BYTE_ARRAY(handle, name->t.name);
310                 break;
311         }
312         return name;
313     }
```

9.4.3.5 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.

```
314     TPMI_RH_HIERARCHY
315     EntityGetHierarchy(
```

```
316         TPMI_DH_ENTITY   handle         // IN :handle of entity
317         )
318     {
319         TPMI_RH_HIERARCHY       hierarcy = TPM_RH_NULL;
320         switch(HandleGetType(handle))
321         {
322             case TPM_HT_PERMANENT:
323                 // hierarchy for a permanent handle
324                 switch(handle)
325                 {
326                     case TPM_RH_PLATFORM:
327                     case TPM_RH_ENDORSEMENT:
328                     case TPM_RH_NULL:
329                         hierarcy = handle;
330                         break;
331                     // all other permanent handles are associated with the owner
332                     // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
333                     default:
334                         hierarcy = TPM_RH_OWNER;
335                         break;
336                 }
337                 break;
338             case TPM_HT_NV_INDEX:
339                 // hierarchy for NV index
340             {
341                 NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
342                 pAssert(nvIndex != NULL);
343                 // If only the platform can delete the index, then it is
344                 // considered to be in the platform hierarchy, otherwise it
345                 // is in the owner hierarchy.
346                 if(IsNv_TPMA_NV_PLATFORMCREATE(nvIndex->publicArea.attributes))
347                     hierarcy = TPM_RH_PLATFORM;
348                 else
349                     hierarcy = TPM_RH_OWNER;
350             }
351             break;
352             case TPM_HT_TRANSIENT:
353                 // hierarchy for an object
354             {
355                 OBJECT          *object;
356                 object = HandleToObject(handle);
357                 if(object->attributes.ppsHierarchy)
358                 {
359                     hierarcy = TPM_RH_PLATFORM;
360                 }
361                 else if(object->attributes.epsHierarchy)
362                 {
363                     hierarcy = TPM_RH_ENDORSEMENT;
364                 }
365                 else if(object->attributes.spsHierarchy)
366                 {
367                     hierarcy = TPM_RH_OWNER;
368                 }
369             }
370             break;
371             case TPM_HT_PCR:
372                 hierarcy = TPM_RH_OWNER;
373                 break;
374             default:
375                 FAIL(FATAL_ERROR_INTERNAL);
376                 break;
377         }
378         // this is unreachable but it provides a return value for the default
379         // case which makes the complier happy
380         return hierarcy;
381     }
```

9.5 Global.c

9.5.1 Description

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

9.5.2 Includes and Defines

```
#define GLOBAL_C
#include "Tpm.h"
```

9.5.3 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;
#ifdef CLOCK_STOPS
CLOCK_NONCE             g_timeEpoch;
#endif
BOOL                    g_timeNewEpochNeeded;
BOOL                    g_pcrReConfig;
TPMI_DH_OBJECT          g_DRTMHandle;
BOOL                    g_DrtmPreStartup;
BOOL                    g_StartupLocality3;
#ifdef USE_DA_USED
BOOL                   g_daUsed;
#endif
BOOL                    g_powerWasLost;
BOOL                    g_clearOrderly;
TPM_SU                  g_prevOrderlyState;
UPDATE_TYPE             g_updateNV;
BOOL                    g_nvOk;
TPM_RC                  g_NvStatus;
TPM2B_AUTH              g_platformUniqueDetails;
STATE_CLEAR_DATA        gc;
STATE_RESET_DATA        gr;
PERSISTENT_DATA         gp;
ORDERLY_DATA            go;
```

9.5.4 Private Values

9.5.4.1 SessionProcess.c

```
#ifndef __IGNORE_STATE__        // DO NOT DEFINE THIS VALUE
```

These values do not need to be retained between commands.

```
TPM_HANDLE              s_sessionHandles[MAX_SESSION_NUM];
TPMA_SESSION            s_attributes[MAX_SESSION_NUM];
TPM_HANDLE              s_associatedHandles[MAX_SESSION_NUM];
TPM2B_NONCE             s_nonceCaller[MAX_SESSION_NUM];
```

```
38   TPM2B_AUTH           s_inputAuthValues[MAX_SESSION_NUM];
39   SESSION             *s_usedSessions[MAX_SESSION_NUM];
40   UINT32               s_encryptSessionIndex;
41   UINT32               s_decryptSessionIndex;
42   UINT32               s_auditSessionIndex;
```

UINT32	*s_sessionNum*;

```
43   #endif  // __IGNORE_STATE__
44   BOOL                 s_DAPendingOnNV;
45   #ifdef TPM_CC_GetCommandAuditDigest
46   TPM2B_DIGEST         s_cpHashForCommandAudit;
47   #endif
```

9.5.4.2 DA.c

```
48   UINT64               s_selfHealTimer;
49   UINT64               s_lockoutTimer;
```

9.5.4.3 NV.c

```
50   UINT64               s_maxCounter;
51   NV_REF               s_evictNvEnd;
52   TPM_RC               g_NvStatus;
53   BYTE                 s_indexOrderlyRam[RAM_INDEX_SPACE];
54   NV_INDEX             s_cachedNvIndex;
55   NV_REF               s_cachedNvRef;
56   BYTE                *s_cachedNvRamRef;
```

9.5.4.4 Object.c

```
57   OBJECT              s_objects[MAX_LOADED_OBJECTS];
```

9.5.4.5 PCR.c

```
58   PCR                  s_pcrs[IMPLEMENTATION_PCR];
```

9.5.4.6 Session.c

```
59   SESSION_SLOT         s_sessions[MAX_LOADED_SESSIONS];
60   UINT32               s_oldestSavedSession;
61   int                  s_freeSessionSlots;
```

9.5.4.7 MemoryLib.c

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

```
62   #ifndef __IGNORE_STATE__        // DO NOT DEFINE THIS VALUE
63   UINT32   s_actionInputBuffer[1024];          // action input buffer
64   UINT32   s_actionOutputBuffer[1024];         // action output buffer
65   #endif
```

9.5.4.8 SelfTest.c

```
66   ALGORITHM_VECTOR         g_implementedAlgorithms;
67   ALGORITHM_VECTOR         g_toTest;
```

9.5.4.9 g_cryptoSelfTestState

This structure contains the cryptographic self-test state values.

```
68   CRYPTO_SELF_TEST_STATE    g_cryptoSelfTestState;
69   ALGORITHM_VECTOR          AlgToTest;
```

9.5.4.10 TpmFail.c

```
70   #ifdef SIMULATION
71   BOOL                 g_forceFailureMode;
72   #endif
73   BOOL                 g_inFailureMode;
74   UINT32               s_failFunction;
75   UINT32               s_failLine;
76   UINT32               s_failCode;
77   #if 0
78   #ifdef TPM_ALG_ECC
```

9.5.4.11 ECC Curves

```
79   ECC_CURVE   c_curves[ECC_CURVE_COUNT];
80   #endif
81   #endif // 0
```

This is the state used when the library uses a random number generator. A special function is installed for the library to call. That function picks up the state from this location and uses it for the generation of the random number.

```
82   RAND_STATE           *s_random;
```

9.5.4.12 Manufacture.c

The values is here rather than in the simulator or platform files in order to make it easier to find the TPM state. This is significant when trying to do TPM virtualization when the TPM state has to be moved along with virtual machine with which it is associated.

```
83   BOOL                 g_manufactured = FALSE;
```

9.5.4.13 Power.c

This is here for the same reason that g_manufactured is here. Both of these values can be provided by the actual platform-specific code or by hardware indications.

```
84   BOOL                 g_initialized;
```

9.5.4.14 Purpose String Constants

These string constants are shared across functions to make sure that they are all using consistent sting values.


```
85    TPM2B_STRING(PRIMARY_OBJECT_CREATION, "Primary Object Creation");
86    TPM2B_STRING(CFB_KEY, "CFB");
87    TPM2B_STRING(CONTEXT_KEY, "CONTEXT");
88    TPM2B_STRING(INTEGRITY_KEY, "INTEGRITY");
89    TPM2B_STRING(SECRET_KEY, "SECRET");
90    TPM2B_STRING(SESSION_KEY, "ATH");
91    TPM2B_STRING(STORAGE_KEY, "STORAGE");
92    TPM2B_STRING(XOR_KEY, "XOR");
93    TPM2B_STRING(COMMIT_STRING, "ECDAA Commit");
94    TPM2B_STRING(DUPLICATE_STRING, "DUPLICATE");
95    TPM2B_STRING(IDENTITY_STRING, "IDENTITY");
96    TPM2B_STRING(OBFUSCATE_STRING, "OBFUSCATE");
97    #ifdef SELF_TEST
98    TPM2B_STRING(OAEP_TEST_STRING, "OAEP Test Value");
99    #endif // SELF_TEST
```


9.6 Handle.c

9.6.1 Description

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

9.6.2 Includes

```
#include "Tpm.h"
```

9.6.3 Functions

9.6.3.1 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);
}
```

9.6.3.2 NextPermanentHandle()

This function returns the permanent handle that is equal to the input value or is the next higher value. If there is no handle with the input value and there is no next higher value, it returns 0:

Return Value	Meaning

```
TPM_HANDLE
NextPermanentHandle(
    TPM_HANDLE       inHandle       // IN: the handle to check
    )
{
    // If inHandle is below the start of the range of permanent handles
    // set it to the start and scan from there
    if(inHandle < TPM_RH_FIRST)
        inHandle = TPM_RH_FIRST;
    // scan from input value until we find an implemented permanent handle
    // or go out of range
    for(; inHandle <= TPM_RH_LAST; inHandle++)
    {
        switch(inHandle)
        {
            case TPM_RH_OWNER:
            case TPM_RH_NULL:
            case TPM_RS_PW:
            case TPM_RH_LOCKOUT:
            case TPM_RH_ENDORSEMENT:
            case TPM_RH_PLATFORM:
            case TPM_RH_PLATFORM_NV:
#ifdef  VENDOR_PERMANENT
            case VENDOR_PERMANENT:
#endif
                return inHandle;
```



```
36                  break;
37              default:
38                  break;
39          }
40      }
41      // Out of range on the top
42      return 0;
43  }
```

9.6.3.3 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

```
44  TPMI_YES_NO
45  PermanentCapGetHandles(
46      TPM_HANDLE       handle,        // IN: start handle
47      UINT32           count,         // IN: count of returned handles
48      TPML_HANDLE     *handleList     // OUT: list of handle
49      )
50  {
51      TPMI_YES_NO     more = NO;
52      UINT32          i;
53      pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
54      // Initialize output handle list
55      handleList->count = 0;
56      // The maximum count of handles we may return is MAX_CAP_HANDLES
57      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
58      // Iterate permanent handle range
59      for(i = NextPermanentHandle(handle);
60      i != 0; i = NextPermanentHandle(i + 1))
61      {
62          if(handleList->count < count)
63          {
64              // If we have not filled up the return list, add this permanent
65              // handle to it
66              handleList->handle[handleList->count] = i;
67              handleList->count++;
68          }
69          else
70          {
71              // If the return list is full but we still have permanent handle
72              // available, report this and stop iterating
73              more = YES;
74              break;
75          }
76      }
77      return more;
78  }
```

9.6.3.4 PermanentCapGetPolicy()

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
PermanentHandleGetPolicy(
    TPM_HANDLE           handle,        // IN: start handle
    UINT32               count,         // IN: max count of returned handles
    TPML_TAGGED_POLICY  *policyList     // OUT: list of handle
    )
{
    TPMI_YES_NO     more = NO;
    pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
    // Initialize output handle list
    policyList->count = 0;
    // The maximum count of policies we may return is MAX_TAGGED_POLICIES
    if(count > MAX_TAGGED_POLICIES)
        count = MAX_TAGGED_POLICIES;
    // Iterate permanent handle range
    for(handle = NextPermanentHandle(handle);
        handle != 0;
        handle = NextPermanentHandle(handle + 1))
    {
        TPM2B_DIGEST    policyDigest;
        TPM_ALG_ID      policyAlg;
        // Check to see if this permanent handle has a policy
        policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
        if(policyAlg == TPM_ALG_ERROR)
           continue;
        if(policyList->count < count)
        {
            // If we have not filled up the return list, add this
            // policy to the list;
            policyList->policies[policyList->count].handle = handle;
            policyList->policies[policyList->count].policyHash.hashAlg = policyAlg;
            MemoryCopy(&policyList->policies[policyList->count].policyHash.digest,
                       policyDigest.t.buffer, policyDigest.t.size);
            policyList->count++;
        }
        else
        {
            // If the return list is full but we still have permanent handle
            // available, report this and stop iterating
            more = YES;
            break;
        }
    }
    return more;
}
```


9.7 IoBuffers.c

9.7.1 Includes and Data Definitions

This definition allows this module to **see** the values that are private to this module but kept in Global.c for ease of state migration.

```
#define IO_BUFFER_C
#include "Tpm.h"
#include "IoBuffers_fp.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.

9.7.2 *Functions*

9.7.2.1 MemoryGetActionInputBuffer()

This function returns the address of the buffer into which the command parameters will be unmarshaled in preparation for calling the command actions.

```
BYTE *
MemoryGetActionInputBuffer(
    UINT32          size          // Size, in bytes, required for the input
                                  // unmarshaling
    )
{
    pAssert(size <= sizeof(s_actionInputBuffer));
   // In this implementation, a static buffer is set aside for the command action
   // input buffer.
    memset(s_actionInputBuffer, 0, size);
    return (BYTE *)&s_actionInputBuffer[0];
}
```

9.7.2.2 MemoryGetActionOutputBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```
void *
MemoryGetActionOutputBuffer(
    UINT32          size          // required size of the buffer
    )
{
    pAssert(size < sizeof(s_actionOutputBuffer));
    // In this implementation, a static buffer is set aside for the command action
    // output buffer.
    memset(s_actionOutputBuffer, 0, size);
    return s_actionOutputBuffer;
}
```

9.7.2.3 IsLabelProperlyFormatted()

This function checks that a label is a null-terminated string.

NOTE: this function is here because there was no better place for it.

Return Value	Meaning
FALSE	string is not null terminated
TRUE	string is null terminated

```
27  #ifndef INLINE_FUNCTIONS
28  BOOL
29  IsLabelProperlyFormatted(
30      TPM2B           *x
31      )
32  {
33      return (((x)->size == 0) || ((x)->buffer[(x)->size - 1] == 0));
34  }
35  #endif // INLINE_FUNCTIONS
```

9.8 Locality.c

9.8.1 Includes

```
#include "Tpm.h"
```

9.8.2 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;
}
```

9.9 Manufacture.c

9.9.1 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.

9.9.2 Includes and Data Definitions

```
#define MANUFACTURE_C
#include "Tpm.h"
#include "TpmSizeChecks_fp.h"
```

9.9.3 Functions

9.9.3.1 TPM_Manufacture()

This function initializes the TPM values in preparation for the TPM's first use. This function will fail if previously called. The TPM can be re-manufactured by calling TPM_Teardown() first and then calling this function again.

Return Value	Meaning
0	success
1	manufacturing process previously performed

```
LIB_EXPORT int
TPM_Manufacture(
    int             firstTime       // IN: indicates if this is the first call from
                                    //     main()
    )
{
    TPM_SU          orderlyShutdown;
#ifdef RUNTIME_SIZE_CHECKS
    // Call the function to verify the sizes of values that result from different
    // compile options.
    TpmSizeChecks();
#endif
    // If TPM has been manufactured, return indication.
    if(!firstTime && g_manufactured)
        return 1;
    s_DAPendingOnNV = FALSE;
    // initialize NV
    NvManufacture();
    // Clear the magic value in the DRBG state
    go.drbgState.magic = 0;
    CryptStartup(SU_RESET);
    // default configuration for PCR
    PCRSimStart();
    // initialize pre-installed hierarchy data
    // This should happen after NV is initialized because hierarchy data is
    // stored in NV.
    HierarchyPreInstall_Init();
    // initialize dictionary attack parameters
    DAPreInstall_Init();
    // initialize PP list
    PhysicalPresencePreInstall_Init();
    // initialize command audit list
    CommandAuditPreInstall_Init();
```



```
37      // first start up is required to be Startup(CLEAR)
38      orderlyShutdown = TPM_SU_CLEAR;
39      NV_WRITE_PERSISTENT(orderlyState, orderlyShutdown);
40      // initialize the firmware version
41      gp.firmwareV1 = FIRMWARE_V1;
42  #ifdef FIRMWARE_V2
43      gp.firmwareV2 = FIRMWARE_V2;
44  #else
45      gp.firmwareV2 = 0;
46  #endif
47      NV_SYNC_PERSISTENT(firmwareV1);
48      NV_SYNC_PERSISTENT(firmwareV2);
49      // initialize the total reset counter to 0
50      gp.totalResetCount = 0;
51      NV_SYNC_PERSISTENT(totalResetCount);
52      // initialize the clock stuff
53      go.clock = 0;
54      go.clockSafe = YES;
55      NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
56      // Commit NV writes.  Manufacture process is an artificial process existing
57      // only in simulator environment and it is not defined in the specification
58      // that what should be the expected behavior if the NV write fails at this
59      // point.  Therefore, it is assumed the NV write here is always success and
60      // no return code of this function is checked.
61      NvCommit();
62      g_manufactured = TRUE;
63      return 0;
64  }
```

9.9.3.2 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

```
65  LIB_EXPORT int
66  TPM_TearDown(
67      void
68      )
69  {
70      g_manufactured = FALSE;
71      return 0;
72  }
```

9.9.3.3 TpmEndSimulation()

This function is called at the end of the simulation run. It is used to provoke printing of any statistics that might be needed.

```
73  LIB_EXPORT void
74  TpmEndSimulation(
75      void
76      )
```

```
77  {
78  #ifdef SIMULATION
79      HashLibSimulationEnd();
80      SymLibSimulationEnd();
81      MathLibSimulationEnd();
82  #ifdef TPM_ALG_RSA
83      RsaSimulationEnd();
84  #endif
85  #ifdef TPM_ALG_ECC
86      EccSimulationEnd();
87  #endif
88  #endif // SIMULATION
89  }
```


9.10 Marshal.c

9.10.1 Introduction

This file contains the marshaling and unmarshaling code.

The marshaling and unmarshaling code and function prototypes are not listed, as the code is repetitive, long, and not very useful to read. Examples of a few unmarshaling routines are provided. Most of the others are similar.

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets ("<>") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

9.10.2 Unmarshal and Marshal a Value

In TPM 2.0 Part 2, a TPMI_DI_OBJECT is defined by this table:

Table xxx — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```
1   TPM_RC
2   TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
3                            BOOL flag)
4   {
5       TPM_RC    result;
6       result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       if(*target == TPM_RH_NULL)
10      {
11          if(flag)
12              return TPM_RC_SUCCESS;
13          else
14              return TPM_RC_VALUE;
15      }
16      if(((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
17        &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
18              return TPM_RC_VALUE;
19      return TPM_RC_SUCCESS;
20  }
```

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);
}
```

An additional script is used to do the work that might be done by a linker or globally optimizing compiler. It searches for functions like TPMI_DH_OBJECT_Marshal() that do nothing but call another function and replaces the function with a #define.

```
#define TPMI_DH_OBJECT_Marshal(source, buffer, size)    \
      UINT32_Marshal((UINT32 *)source, buffer, size)
```

When replacing the function with a #define, the #define is placed in marshal_fp.h and the function body is removed from marshal.c.

9.10.3 Unmarshal and Marshal a Union

In TPM 2.0 Part 2, a TPMU_PUBLIC_PARMS union is defined by:

Table xxx — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S>

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign | encrypt | neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys
NOTE The Description column indicates which of TPMA_OBJECT.*decrypt* or TPMA_OBJECT.*sign* may be set. "+" indicates that both may be set but one shall be set. "|" indicates the optional settings.			

From this table, the following unmarshaling code is generated.

```
TPM_RC
TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
                            UINT32 selector)
{
    switch(selector) {
#ifdef TPM_ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Unmarshal(
                        (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
#endif
#ifdef TPM_ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Unmarshal(
                    (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
#endif
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Unmarshal(
                                (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
#endif
#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Unmarshal(
                                (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
#endif
    }
```

```
    return TPM_RC_SELECTOR;
}
```

NOTE The `#ifdef/#endif` directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

```
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.

9.10.4 Unmarshal and Marshal a Structure

In TPM 2.0 Part 2, the TPMT_PUBLIC structure is defined by:

Table xxx — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	"algorithm" associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```
1   TPM_RC
2   TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
3   {
4       TPM_RC    result;
5       result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                          buffer, size);
7       if(result != TPM_RC_SUCCESS)
8           return result;
9       result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                       buffer, size, flag);
11      if(result != TPM_RC_SUCCESS)
12          return result;
13      result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                     buffer, size);
15      if(result != TPM_RC_SUCCESS)
16          return result;
17      result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                      buffer, size);
19      if(result != TPM_RC_SUCCESS)
20          return result;
21
22      result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                           buffer, size, (UINT32)target->type);
24      if(result != TPM_RC_SUCCESS)
25          return result;
26
27      result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                        buffer, size, (UINT32)target->type);
29      if(result != TPM_RC_SUCCESS)
30          return result;
31
32      return TPM_RC_SUCCESS;
33  }
```


The marshaling code for the TPMT_PUBLIC structure is:

```
UINT16
TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
                                   (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
    result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
                                  (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
;
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                           (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));

    result = (UINT16)(result + TPM2B_DIGEST_Marshal(
                                (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));

    result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
                           (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
                                                                  (UINT32)source->type));

    result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
                                  (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
                                                                  (UINT32)source->type));

    return result;
}
```

9.10.5 Unmarshal and Marshal an Array

In TPM 2.0 Part 2, the TPML_DIGEST is defined by:

Table xxx — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```
TPM_RC
TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    if( (target->count < 2))    // This check is triggered by the {2:} notation
                                // on 'count'
        return TPM_RC_SIZE;

    if((target->count) > 8)     // This check is triggered by the {:8} notation
```

```
                                    // on 'digests'.
        return TPM_RC_SIZE;

    result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
                                          buffer, size, (INT32)(target->count));
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}

```

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
UINT16
TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
                                                                        size));
    result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
                                    (TPM2B_DIGEST *)(source->digests), buffer, size,
                                    (INT32)(source->count)));

    return result;
}
```

The marshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}
```


9.10.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 9.10.5 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.

When marshaling a TPM2B where the second member is a BYTE array, the size parameter indicates the size of the array. The second member can also be a structure. In this case, the caller does not prefill the size member. The marshaling code must marshal the structure and then back fill the calculated size.

Table xxx — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

```
1   TPM_RC
2   TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3   {
4       TPM_RC    result;
5       result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6       if(result != TPM_RC_SUCCESS)
7           return result;
8       // if size equal to 0, the rest of the structure is a zero buffer
9       //  so stop processing
10      if(target->t.size == 0)
11          return TPM_RC_SUCCESS;
12      if((target->t.size) > 1024)    // This check is triggered by the {:1024}
13                                     // notation on 'buffer'
14          return TPM_RC_SIZE;
15      result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
16                                    (INT32)(target->t.size));
17      if(result != TPM_RC_SUCCESS)
18          return result;
19      return TPM_RC_SUCCESS;
20  }
```

using these structure definitions:

```
1   typedef union {
2       struct {
3           UINT16        size;
4           BYTE          buffer[1024];
5       }             t;
6       TPM2B         b;
7   } TPM2B_EVENT;
```


9.11 MathOnByteBuffers.c

9.11.1 Introduction

This file contains implementation of the math functions that are performed with canonical integers in byte buffers. The canonical integer is big-endian bytes.

```
1   #include "Tpm.h"
```

9.11.2 UnsignedCmpB

This function compare two unsigned values. The values are byte-aligned, big-endian numbers (e.g, a hash).

Return Value	Meaning
1	if (a > b)
0	if (a = b)
-1	if (a < b)

```
2   LIB_EXPORT int
3   UnsignedCompareB(
4       UINT32          aSize,          // IN: size of a
5       const BYTE      *a,             // IN: a
6       UINT32          bSize,          // IN: size of b
7       const BYTE      *b              // IN: b
8       )
9   {
10      UINT32             i;
11      if(aSize > bSize)
12          return 1;
13      else if(aSize < bSize)
14          return -1;
15      else
16      {
17          for(i = 0; i < aSize; i++)
18          {
19              if(a[i] != b[i])
20                  return (a[i] > b[i]) ? 1 : -1;
21          }
22      }
23      return 0;
24  }
```

9.11.3 SignedCompareB()

Compare two signed integers:

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
25  int
26  SignedCompareB(
27      const UINT32     aSize,         // IN: size of a
```

```
28      const BYTE      *a,             // IN: a buffer
29      const UINT32     bSize,         // IN: size of b
30      const BYTE      *b              // IN: b buffer
31      )
32  {
33      int      signA, signB;       // sign of a and b
34      // For positive or 0, sign_a is 1
35      // for negative, sign_a is 0
36      signA = ((a[0] & 0x80) == 0) ? 1 : 0;
37      // For positive or 0, sign_b is 1
38      // for negative, sign_b is 0
39      signB = ((b[0] & 0x80) == 0) ? 1 : 0;
40      if(signA != signB)
41      {
42          return signA - signB;
43      }
44      if(signA == 1)
45          // do unsigned compare function
46          return UnsignedCompareB(aSize, a, bSize, b);
47      else
48          // do unsigned compare the other way
49          return 0 - UnsignedCompareB(aSize, a, bSize, b);
50  }
```

9.11.4 ModExpB

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*.

Error Returns	Meaning
TPM_RC_SIZE	number to exponentiate is larger than the modulus
TPM_RC_NO_RESULT	result will not fit into the provided buffer

```
51  TPM_RC
52  ModExpB(
53      UINT32           cSize,         // IN: the size of the output buffer. It will
54                                      //     need to be the same size as the modulus
55      BYTE            *c,             // OUT: the buffer to receive the results
56                                      //     (c->size must be set to the maximum size
57                                      //     for the returned value)
58      const UINT32     mSize,
59      const BYTE      *m,             // IN: number to exponentiate
60      const UINT32     eSize,
61      const BYTE      *e,             // IN: power
62      const UINT32     nSize,
63      const BYTE      *n              // IN: modulus
64      )
65  {
66      BN_MAX(bnC);
67      BN_MAX(bnM);
68      BN_MAX(bnE);
69      BN_MAX(bnN);
70      NUMBYTES         tSize = (NUMBYTES)nSize;
71      TPM_RC           retVal = TPM_RC_SUCCESS;
72      // Convert input parameters
73      BnFromBytes(bnM, m, (NUMBYTES)mSize);
```

```
74      BnFromBytes(bnE, e, (NUMBYTES)eSize);
75      BnFromBytes(bnN, n, (NUMBYTES)nSize);
76      // Make sure that the output is big enough to hold the result
77      // and that 'm' is less than 'n' (the modulus)
78      if(cSize < nSize)
79          ERROR_RETURN(TPM_RC_NO_RESULT);
80      if(BnUnsignedCmp(bnM, bnN) >= 0)
81          ERROR_RETURN(TPM_RC_SIZE);
82      BnModExp(bnC, bnM, bnE, bnN);
83      BnToBytes(bnC, c, &tSize);
84  Exit:
85      return retVal;
86  }
```

9.11.5 DivideB()

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.

Error Returns	Meaning
TPM_RC_SUCCESS	operation complete
TPM_RC_NO_RESULT	*q* or *r* is too small to receive the result

```
87  LIB_EXPORT TPM_RC
88  DivideB(
89      const TPM2B     *n,             // IN: numerator
90      const TPM2B     *d,             // IN: denominator
91      TPM2B           *q,             // OUT: quotient
92      TPM2B           *r              // OUT: remainder
93      )
94  {
95      BN_MAX_INITIALIZED(bnN, n);
96      BN_MAX_INITIALIZED(bnD, d);
97      BN_MAX(bnQ);
98      BN_MAX(bnR);
99  //
100     // Do divide with converted values
101     BnDiv(bnQ, bnR, bnN, bnD);
102     // Convert the BIGNUM result back to 2B format using the size of the original
103     // number
104     if(q != NULL)
105         if(!BnTo2B(bnQ, q, q->size))
106             return TPM_RC_NO_RESULT;
107     if(r != NULL)
108         if(!BnTo2B(bnR, r, r->size))
109             return TPM_RC_NO_RESULT;
110     return TPM_RC_SUCCESS;
111 }
```

9.11.6 AdjustNumberB()

Remove/add leading zeros from a number in a TPM2B. Will try to make the number by adding or removing leading zeros. If the number is larger than the requested size, it will make the number as small as possible. Setting *requestedSize* to zero is equivalent to requesting that the number be normalized.

```
112 UINT16
113 AdjustNumberB(
114     TPM2B           *num,
115     UINT16           requestedSize
116     )
```

```
117 {
118     BYTE            *from;
119     UINT16           i;
120     // This is a request to shift the number to the left (remove leading zeros)
121     if(num->size == requestedSize)
122         return requestedSize;
123     from = num->buffer;
124     if (num->size > requestedSize)
125     {
126         // Find the first non-zero byte. Don't look past the point where removing
127         // more zeros would make the number smaller than requested.
128         for(i = num->size; *from == 0 && i > requestedSize; from++, i++);
129         if(i < num->size)
130         {
131             num->size = i;
132             MemoryCopy(num->buffer, from, i);
133         }
134     }
135     // This is a request to shift the number to the right (add leading zeros)
136     else
137     {
138         MemoryCopy(&num->buffer[requestedSize - num->size], num->buffer, num->size);
139         MemorySet(num->buffer, 0, requestedSize- num->size);
140         num->size = requestedSize;
141     }
142     return num->size;
143 }
```


9.12 Memory.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 directly in the TPM because they are not *safe*

This version uses string.h after adding guards. This is because the math libraries invariably use those functions so it is not practical to prevent those library functions from being pulled into the build.

9.12.2 Includes and Data Definitions

```
#include "Tpm.h"
#include "Memory_fp.h"
```

9.12.3 Functions

9.12.3.1 MemoryCopy()

This is an alias for memmove. This is used in place of memcpy because some of the moves may overlap and rather than try to make sure that memmove is used when necessary, it is always used. The #if 0 is used to prevent instantiation of the MemoryCopy() function so that the #define is always used

```
#ifndef INLINE_FUNCTIONS
void
MemoryCopy(
    void        *dest,
    const void  *src,
    int          sSize
    )
{
    memmove(dest, src, sSize);
}
#endif // INLINE_FUNCTIONS
```

9.12.3.2 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

```
BOOL
MemoryEqual(
    const void      *buffer1,       // IN: compare buffer1
    const void      *buffer2,       // IN: compare buffer2
    unsigned int     size           // IN: size of bytes being compared
    )
{
    BYTE         equal = 0;
    const BYTE  *b1 = (BYTE *)buffer1;
    const BYTE  *b2 = (BYTE *)buffer2;
//
    // Compare all bytes so that there is no leakage of information
```

```
26      // due to timing differences.
27      for(; size > 0; size--)
28          equal |= (*b1++ ^ *b2++);
29      return (equal == 0);
30  }
```

9.12.3.3 MemoryCopy2B()

This function copies a TPM2B. This can be used when the TPM2B types are the same or different.

This function returns the number of octets in the data buffer of the TPM2B.

```
31  LIB_EXPORT INT16
32  MemoryCopy2B(
33      TPM2B           *dest,          // OUT: receiving TPM2B
34      const TPM2B     *source,        // IN: source TPM2B
35      unsigned int     dSize          // IN: size of the receiving buffer
36      )
37  {
38      pAssert(dest != NULL);
39      if(source == NULL)
40          dest->size = 0;
41      else
42      {
43          pAssert(source->size <= dSize);
44          MemoryCopy(dest->buffer, source->buffer, source->size);
45          dest->size = source->size;
46      }
47      return dest->size;
48  }
```

9.12.3.4 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)*).

```
49  void
50  MemoryConcat2B(
51      TPM2B           *aInOut,        // IN/OUT: destination 2B
52      TPM2B           *bIn,           // IN: second 2B
53      unsigned int     aMaxSize       // IN: The size of aInOut.buffer (max values for
54                                      //     aInOut.size)
55      )
56  {
57      pAssert(bIn->size <= aMaxSize - aInOut->size);
58      MemoryCopy(&aInOut->buffer[aInOut->size], &bIn->buffer, bIn->size);
59      aInOut->size = aInOut->size + bIn->size;
60      return;
61  }
```

9.12.3.5 MemoryEqual2B()

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

```
62  BOOL
63  MemoryEqual2B(
64      const TPM2B     *aIn,           // IN: compare value
65      const TPM2B     *bIn            // IN: compare value
66      )
67  {
68      if(aIn->size != bIn->size)
69          return FALSE;
70      return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
71  }
```

9.12.3.6 MemorySet()

This function will set all the octets in the specified memory range to the specified octet value.

NOTE: A previous version had an additional parameter *(dSize)* that was intended to make sure that the destination would not be overrun. The problem is that, in use, all that was happening was that the value of size was used for *dSize* so there was no benefit in the extra parameter.

```
72  #ifndef INLINE_FUNCTIONS
73  void
74  MemorySet(
75      void            *dest,
76      int              value,
77      size_t           size
78      )
79  {
80      memset(dest, value, size);
81  }
82  #endif // INLINE_FUNCTIONS
```

9.12.3.7 MemoryPad2B()

Function to pad a TPM2B with zeros and adjust the size.

```
83  #ifndef INLINE_FUNCTIONS
84  void
85  MemoryPad2B(
86      TPM2B           *b,
87      UINT16           newSize
88      )
89  {
90      MemorySet(&b->buffer[b->size], 0, newSize - b->size);
91      b->size = newSize;
92  }
93  #endif // INLINE_FUNCTIONS
```

9.12.3.8 Uint16ToByteArray()

Function to write an integer to a byte array

```
94  #ifndef INLINE_FUNCTIONS
95  void
96  Uint16ToByteArray(
97      UINT16              i,
```

```
 98      BYTE                *a
 99      )
100  {
101      a[1] = (BYTE)(i); i >>= 8;
102      a[0] = (BYTE)(i);
103  }
104  #endif // INLINE_FUNCTIONS
```

9.12.3.9 Uint32ToByteArray()

Function to write an integer to a byte array

```
105  #ifndef INLINE_FUNCTIONS
106  void
107  Uint32ToByteArray(
108      UINT32              i,
109      BYTE                *a
110      )
111  {
112      a[3] = (BYTE)(i); i >>= 8;
113      a[2] = (BYTE)(i); i >>= 8;
114      a[1] = (BYTE)(i); i >>= 8;
115      a[0] = (BYTE)(i);
116  }
117  #endif // INLINE_FUNCTIONS
```

9.12.3.10 Uint64ToByteArray()

Function to write an integer to a byte array

```
118  #ifndef INLINE_FUNCTIONS
119  void
120  Uint64ToByteArray(
121      UINT64               i,
122      BYTE                *a
123      )
124  {
125      a[7] = (BYTE)(i); i >>= 8;
126      a[6] = (BYTE)(i); i >>= 8;
127      a[5] = (BYTE)(i); i >>= 8;
128      a[4] = (BYTE)(i); i >>= 8;
129      a[3] = (BYTE)(i); i >>= 8;
130      a[2] = (BYTE)(i); i >>= 8;
131      a[1] = (BYTE)(i); i >>= 8;
132      a[0] = (BYTE)(i);
133  }
134  #endif // INLINE_FUNCTIONS
```

9.12.3.11 ByteArrayToUint16()

Function to write an integer to a byte array

```
135  #ifndef INLINE_FUNCTIONS
136  UINT16
137  ByteArrayToUint16(
138      BYTE                *a
139      )
140  {
141      UINT16      retVal;
142      retVal  = a[0]; retVal <<= 8;
143      retVal += a[1];
```

```
144         return retVal;
145     }
146     #endif // INLINE_FUNCTIONS
```

9.12.3.12 ByteArrayToUint32()

Function to write an integer to a byte array

```
147     #ifndef INLINE_FUNCTIONS
148     UINT32
149     ByteArrayToUint32(
150         BYTE                *a
151         )
152     {
153         UINT32      retVal;
154         retVal  = a[0]; retVal <<= 8;
155         retVal += a[1]; retVal <<= 8;
156         retVal += a[2]; retVal <<= 8;
157         retVal += a[3];
158         return retVal;
159     }
160     #endif // INLINE_FUNCTIONS
```

9.12.3.13 ByteArrayToUint64()

Function to write an integer to a byte array

```
161     #ifndef INLINE_FUNCTIONS
162     UINT64
163     ByteArrayToUint64(
164         BYTE                *a
165         )
166     {
167         UINT64      retVal;
168         retVal  = a[0]; retVal <<= 8;
169         retVal += a[1]; retVal <<= 8;
170         retVal += a[2]; retVal <<= 8;
171         retVal += a[3]; retVal <<= 8;
172         retVal += a[4]; retVal <<= 8;
173         retVal += a[5]; retVal <<= 8;
174         retVal += a[6]; retVal <<= 8;
175         retVal += a[7];
176         return retVal;
177     }
178     #endif // INLINE_FUNCTIONS
```

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 "Tpm.h"
```

9.13.3 Functions

9.13.3.1 TPMInit()

This function is used to process a power on event.

```
#ifndef INLINE_FUNCTIONS
void
TPMInit(
    void
    )
{
    // Set state as not initialized. This means that Startup is required
    g_initialized = FALSE;
    return;
}
#endif  // INLINE_FUNCTIONS
```

9.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2_Startup() has completed successfully).

```
#ifndef INLINE_FUNCTIONS
void
TPMRegisterStartup(
    void
    )
{
    g_initialized = TRUE;
    return;
}
#endif  // INLINE_FUNCTIONS
```

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

```
#ifndef INLINE_FUNCTIONS
```

```
25  BOOL
26  TPMIsStarted(
27      void
28      )
29  {
30      return g_initialized;
31  }
32  #endif // INLINE_FUNCTIONS
```


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 "Tpm.h"
```

9.14.3 Functions

9.14.3.1 TPMPropertyIsDefined()

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 = ((UINT32)1 << (sizeof(CONTEXT_SLOT) * 8)) - 1;
112                 break;
113             case TPM_PT_NV_COUNTERS_MAX:
114                 // maximum number of NV indexes that are allowed to have the
115                 // TPMA_NV_COUNTER attribute SET
116                 // In this implementation, there is no limitation on the number
117                 // of counters, except for the size of the NV Index memory.
118                 *value = 0;
119                 break;
120             case TPM_PT_NV_INDEX_MAX:
121                 // maximum size of an NV index data area
122                 *value = MAX_NV_INDEX_SIZE;
123                 break;
124             case TPM_PT_MEMORY:
125                 // a TPMA_MEMORY indicating the memory management method for the TPM
126                 {
127                     union
128                     {
129                         TPMA_MEMORY     att;
130                         UINT32          u32;
131                     } attributes = {{0}};
132                     attributes.att.sharedNV = SET;
133                     attributes.att.objectCopiedToRam = SET;
134                     // Copy the bytes of the TPMA_MEMORY to the 32 bit integer assuming
135                     // that the structure is going to be packed and, because the union
136                     // contains a UINT32, it will be properly aligned. Note: this will
137                     // get and could get byte swapped if the CPU is little-endian.
138                     *value = attributes.u32;
139                     break;
140                 }
141             case TPM_PT_CLOCK_UPDATE:
142                 // interval, in seconds, between updates to the copy of
143                 // TPMS_TIME_INFO .clock in NV
144                 *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
145                 break;
146             case TPM_PT_CONTEXT_HASH:
147                 // algorithm used for the integrity hash on saved contexts and
148                 // for digesting the fuData of TPM2_FirmwareRead()
149                 *value = CONTEXT_INTEGRITY_HASH_ALG;
150                 break;
151             case TPM_PT_CONTEXT_SYM:
152                 // algorithm used for encryption of saved contexts
153                 *value = CONTEXT_ENCRYPT_ALG;
154                 break;
155             case TPM_PT_CONTEXT_SYM_SIZE:
156                 // size of the key used for encryption of saved contexts
157                 *value = CONTEXT_ENCRYPT_KEY_BITS;
158                 break;
159             case TPM_PT_ORDERLY_COUNT:
160                 // maximum difference between the volatile and non-volatile
161                 // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
162                 *value = MAX_ORDERLY_COUNT;
163                 break;
164             case TPM_PT_MAX_COMMAND_SIZE:
165                 // maximum value for 'commandSize'
166                 *value = MAX_COMMAND_SIZE;
167                 break;
168             case TPM_PT_MAX_RESPONSE_SIZE:
169                 // maximum value for 'responseSize'
```



```
170             *value = MAX_RESPONSE_SIZE;
171             break;
172         case TPM_PT_MAX_DIGEST:
173             // maximum size of a digest that can be produced by the TPM
174             *value = sizeof(TPMU_HA);
175             break;
176         case TPM_PT_MAX_OBJECT_CONTEXT:
177 // Header has 'sequence', 'handle' and 'hierarchy'
178 #define SIZE_OF_CONTEXT_HEADER                  \
179     sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) + sizeof(TPMI_RH_HIERARCHY)
180 #define SIZE_OF_CONTEXT_INTEGRITY (sizeof(UINT16) + CONTEXT_INTEGRITY_HASH_SIZE)
181 #define SIZE_OF_FINGERPRINT     sizeof(UINT64)
182 #define SIZE_OF_CONTEXT_BLOB_OVERHEAD                \
183         (sizeof(UINT16)  + SIZE_OF_CONTEXT_INTEGRITY + SIZE_OF_FINGERPRINT)
184 #define SIZE_OF_CONTEXT_OVERHEAD                     \
185         (SIZE_OF_CONTEXT_HEADER + SIZE_OF_CONTEXT_BLOB_OVERHEAD)
186 #if 0
187             // maximum size of a TPMS_CONTEXT that will be returned by
188             // TPM2_ContextSave for object context
189             *value = 0;
190             // adding sequence, saved handle and hierarchy
191             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
192                 sizeof(TPMI_RH_HIERARCHY);
193             // add size field in TPM2B_CONTEXT
194             *value += sizeof(UINT16);
195             // add integrity hash size
196             *value += sizeof(UINT16) +
197                 CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
198             // Add fingerprint size, which is the same as sequence size
199             *value += sizeof(UINT64);
200             // Add OBJECT structure size
201             *value += sizeof(OBJECT);
202 #else
203             // the maximum size of a TPMS_CONTEXT that will be returned by
204             // TPM2_ContextSave for object context
205             *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(OBJECT);
206 #endif
207             break;
208         case TPM_PT_MAX_SESSION_CONTEXT:
209 #if 0
210             // the maximum size of a TPMS_CONTEXT that will be returned by
211             // TPM2_ContextSave for object context
212             *value = 0;
213             // adding sequence, saved handle and hierarchy
214             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
215                 sizeof(TPMI_RH_HIERARCHY);
216             // Add size field in TPM2B_CONTEXT
217             *value += sizeof(UINT16);
218 // Add integrity hash size
219             *value += sizeof(UINT16) +
220                 CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
221       // Add fingerprint size, which is the same as sequence size
222             *value += sizeof(UINT64);
223             // Add SESSION structure size
224             *value += sizeof(SESSION);
225 #else
226              // the maximum size of a TPMS_CONTEXT that will be returned by
227             // TPM2_ContextSave for object context
228             *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(SESSION);
229 #endif
230             break;
231         case TPM_PT_PS_FAMILY_INDICATOR:
232             // platform specific values for the TPM_PT_PS parameters from
233             // the relevant platform-specific specification
234             // In this reference implementation, all of these values are 0.
235             *value = PLATFORM_FAMILY;
```

```
236             break;
237         case TPM_PT_PS_LEVEL:
238             // level of the platform-specific specification
239             *value = PLATFORM_LEVEL;
240             break;
241         case TPM_PT_PS_REVISION:
242             // specification Revision times 100 for the platform-specific
243             // specification
244             *value = PLATFORM_VERSION;
245             break;
246         case TPM_PT_PS_DAY_OF_YEAR:
247             // platform-specific specification day of year using TCG calendar
248             *value = PLATFORM_DAY_OF_YEAR;
249             break;
250         case TPM_PT_PS_YEAR:
251             // platform-specific specification year using the CE
252             *value = PLATFORM_YEAR;
253             break;
254         case TPM_PT_SPLIT_MAX:
255             // number of split signing operations supported by the TPM
256             *value = 0;
257 #ifdef TPM_ALG_ECC
258             *value = sizeof(gr.commitArray) * 8;
259 #endif
260             break;
261         case TPM_PT_TOTAL_COMMANDS:
262             // total number of commands implemented in the TPM
263             // Since the reference implementation does not have any
264             // vendor-defined commands, this will be the same as the
265             // number of library commands.
266         {
267 #ifdef COMPRESSED_LISTS
268             (*value) = COMMAND_COUNT;
269 #else
270             COMMAND_INDEX       commandIndex;
271             *value = 0;
272             // scan all implemented commands
273             for(commandIndex = GetClosestCommandIndex(0);
274             commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
275                 commandIndex = GetNextCommandIndex(commandIndex))
276             {
277                 (*value)++;     // count of all implemented
278             }
279 #endif
280             break;
281         }
282         case TPM_PT_LIBRARY_COMMANDS:
283             // number of commands from the TPM library that are implemented
284         {
285 #ifdef  COMPRESSED_LISTS
286             *value = LIBRARY_COMMAND_ARRAY_SIZE;
287 #else
288             COMMAND_INDEX       commandIndex;
289             *value = 0;
290             // scan all implemented commands
291             for(commandIndex = GetClosestCommandIndex(0);
292             commandIndex < LIBRARY_COMMAND_ARRAY_SIZE;
293                 commandIndex = GetNextCommandIndex(commandIndex))
294             {
295                 (*value)++;
296             }
297 #endif
298             break;
299         }
300         case TPM_PT_VENDOR_COMMANDS:
301             // number of vendor commands that are implemented
```

```
302             *value = VENDOR_COMMAND_ARRAY_SIZE;
303             break;
304         case TPM_PT_NV_BUFFER_MAX:
305             // Maximum data size in an NV write command
306             *value = MAX_NV_BUFFER_SIZE;
307             break;
308         case TPM_PT_MODES:
309 #ifdef FIPS_COMPLIANT
310             *value = 1;
311 #else
312             *value = 0;
313 #endif
314             break;
315         case TPM_PT_MAX_CAP_BUFFER:
316             *value = MAX_CAP_BUFFER;
317             break;
318         // Start of variable commands
319         case TPM_PT_PERMANENT:
320             // TPMA_PERMANENT
321         {
322             TPMA_PERMANENT          flags = {0};
323             if(gp.ownerAuth.t.size != 0)
324                 flags.ownerAuthSet = SET;
325             if(gp.endorsementAuth.t.size != 0)
326                 flags.endorsementAuthSet = SET;
327             if(gp.lockoutAuth.t.size != 0)
328                 flags.lockoutAuthSet = SET;
329             if(gp.disableClear)
330                 flags.disableClear = SET;
331             if(gp.failedTries >= gp.maxTries)
332                 flags.inLockout = SET;
333             // In this implementation, EPS is always generated by TPM
334             flags.tpmGeneratedEPS = SET;
335             // Note: Different compilers may require a different method to cast
336             // a bit field structure to a UINT32.
337             *value = *(UINT32 *)&flags;
338             break;
339         }
340         case TPM_PT_STARTUP_CLEAR:
341             // TPMA_STARTUP_CLEAR
342         {
343             TPMA_STARTUP_CLEAR      flags = {0};
344             if(g_phEnable)
345                 flags.phEnable = SET;
346             if(gc.shEnable)
347                 flags.shEnable = SET;
348             if(gc.ehEnable)
349                 flags.ehEnable = SET;
350             if(gc.phEnableNV)
351                 flags.phEnableNV = SET;
352             if(g_prevOrderlyState != SU_NONE_VALUE)
353                 flags.orderly = SET;
354             // Note: Different compilers may require a different method to cast
355             // a bit field structure to a UINT32.
356             *value = *(UINT32 *)&flags;
357             break;
358         }
359         case TPM_PT_HR_NV_INDEX:
360             // number of NV indexes currently defined
361             *value = NvCapGetIndexNumber();
362             break;
363         case TPM_PT_HR_LOADED:
364             // number of authorization sessions currently loaded into TPM
365             // RAM
366             *value = SessionCapGetLoadedNumber();
367             break;
```



```
368             case TPM_PT_HR_LOADED_AVAIL:
369                 // number of additional authorization sessions, of any type,
370                 // that could be loaded into TPM RAM
371                 *value = SessionCapGetLoadedAvail();
372                 break;
373             case TPM_PT_HR_ACTIVE:
374                 // number of active authorization sessions currently being
375                 // tracked by the TPM
376                 *value = SessionCapGetActiveNumber();
377                 break;
378             case TPM_PT_HR_ACTIVE_AVAIL:
379                 // number of additional authorization sessions, of any type,
380                 // that could be created
381                 *value = SessionCapGetActiveAvail();
382                 break;
383             case TPM_PT_HR_TRANSIENT_AVAIL:
384                 // estimate of the number of additional transient objects that
385                 // could be loaded into TPM RAM
386                 *value = ObjectCapGetTransientAvail();
387                 break;
388             case TPM_PT_HR_PERSISTENT:
389                 // number of persistent objects currently loaded into TPM
390                 // NV memory
391                 *value = NvCapGetPersistentNumber();
392                 break;
393             case TPM_PT_HR_PERSISTENT_AVAIL:
394                 // number of additional persistent objects that could be loaded
395                 // into NV memory
396                 *value = NvCapGetPersistentAvail();
397                 break;
398             case TPM_PT_NV_COUNTERS:
399                 // number of defined NV indexes that have NV TPMA_NV_COUNTER
400                 // attribute SET
401                 *value = NvCapGetCounterNumber();
402                 break;
403             case TPM_PT_NV_COUNTERS_AVAIL:
404                 // number of additional NV indexes that can be defined with their
405                 // TPMA_NV_COUNTER attribute SET
406                 *value = NvCapGetCounterAvail();
407                 break;
408             case TPM_PT_ALGORITHM_SET:
409                 // region code for the TPM
410                 *value = gp.algorithmSet;
411                 break;
412             case TPM_PT_LOADED_CURVES:
413 #ifdef TPM_ALG_ECC
414                 // number of loaded ECC curves
415                 *value = ECC_CURVE_COUNT;
416 #else // TPM_ALG_ECC
417                 *value = 0;
418 #endif // TPM_ALG_ECC
419                 break;
420             case TPM_PT_LOCKOUT_COUNTER:
421                 // current value of the lockout counter
422                 *value = gp.failedTries;
423                 break;
424             case TPM_PT_MAX_AUTH_FAIL:
425                 // number of authorization failures before DA lockout is invoked
426                 *value = gp.maxTries;
427                 break;
428             case TPM_PT_LOCKOUT_INTERVAL:
429                 // number of seconds before the value reported by
430                 // TPM_PT_LOCKOUT_COUNTER is decremented
431                 *value = gp.recoveryTime;
432                 break;
433             case TPM_PT_LOCKOUT_RECOVERY:
```



```
            // number of seconds after a lockoutAuth failure before use of
            // lockoutAuth may be attempted again
            *value = gp.lockoutRecovery;
            break;
        case TPM_PT_NV_WRITE_RECOVERY:
            // number of milliseconds before the TPM will accept another command
            // that will modify NV.
            // This should make a call to the platform code that is doing rate
            // limiting of NV. Rate limiting is not implemented in the reference
            // code so no call is made.
            *value = 0;
            break;
        case TPM_PT_AUDIT_COUNTER_0:
            // high-order 32 bits of the command audit counter
            *value = (UINT32)(gp.auditCounter >> 32);
            break;
        case TPM_PT_AUDIT_COUNTER_1:
            // low-order 32 bits of the command audit counter
            *value = (UINT32)(gp.auditCounter);
            break;
        default:
            // property is not defined
            return FALSE;
            break;
    }
    return TRUE;
}
```

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

```
TPMI_YES_NO
TPMCapGetProperties(
    TPM_PT                      property,       // IN: the starting TPM property
    UINT32                      count,          // IN: maximum number of returned
                                                //     properties
    TPML_TAGGED_TPM_PROPERTY    *propertyList   // OUT: property list
    )
{
    TPMI_YES_NO     more = NO;
    UINT32          i;
    UINT32          nextGroup;
    // initialize output property list
    propertyList->count = 0;
    // maximum count of properties we may return is MAX_PCR_PROPERTIES
    if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;
    // if property is less than PT_FIXED, start from PT_FIXED
    if(property < PT_FIXED)
        property = PT_FIXED;
    // There is only the fixed and variable groups with the variable group coming
    // last
    if(property >= (PT_VAR + PT_GROUP))
        return more;
    // Don't read past the end of the selected group
    nextGroup = ((property / PT_GROUP) * PT_GROUP) + PT_GROUP;
```



```
485         // Scan through the TPM properties of the requested group.
486         for(i = property; i < nextGroup; i++)
487         {
488             UINT32          value;
489             // if we have hit the end of the group, quit
490             if(i != property && ((i % PT_GROUP) == 0))
491                 break;
492             if(TPMPropertyIsDefined((TPM_PT)i, &value))
493             {
494                 if(propertyList->count < count)
495                 {
496                     // If the list is not full, add this property
497                     propertyList->tpmProperty[propertyList->count].property =
498                         (TPM_PT)i;
499                     propertyList->tpmProperty[propertyList->count].value = value;
500                     propertyList->count++;
501                 }
502                 else
503                 {
504                     // If the return list is full but there are more properties
505                     // available, set the indication and exit the loop.
506                     more = YES;
507                     break;
508                 }
509             }
510         }
511         return more;
512     }
```


9.15 Response.c

9.15.1 Description

This file contains the common code for building a response header, including setting the size of the structure. *command* may be NULL if result is not TPM_RC_SUCCESS.

9.15.2 Includes and Defines

```
#include "Tpm.h"
```

9.15.3 BuildResponseHeader()

Adds the reponse header to the response. It will update command->*parameterSize* to indicate the total size of the response.

```
void
BuildResponseHeader(
    COMMAND         *command,       // IN: main control structure
    BYTE            *buffer,        // OUT: the output buffer
    TPM_RC           result         // IN: the response code
    )
{
    TPM_ST              tag;
    UINT32              size;
    if(result != TPM_RC_SUCCESS)
    {
        tag = TPM_ST_NO_SESSIONS;
        size = 10;
    }
    else
    {
        tag = command->tag;
        // Compute the overall size of the response
        size = STD_RESPONSE_HEADER + command->handleNum * sizeof(TPM_HANDLE);
        size += command->parameterSize;
        size += (command->tag == TPM_ST_SESSIONS) ?
            command->authSize + sizeof(UINT32) : 0;
    }
    TPM_ST_Marshal(&tag, &buffer, NULL);
    UINT32_Marshal(&size, &buffer, NULL);
    TPM_RC_Marshal(&result, &buffer, NULL);
    if(result == TPM_RC_SUCCESS)
    {
        if(command->handleNum > 0)
            TPM_HANDLE_Marshal(&command->handles[0], &buffer, NULL);
        if(tag == TPM_ST_SESSIONS)
            UINT32_Marshal((UINT32 *)&command->parameterSize, &buffer, NULL);
    }
    command->parameterSize = size;
}
```


9.16 ResponseCodeProcessing.c

9.16.1 Description

This file contains the miscellaneous functions for processing response codes.

NOTE: Currently, there is only one.

9.16.2 Includes and Defines

```
1   #include "Tpm.h"
```

9.16.3 RcSafeAddToResult()

Adds a modifier to a response code as long as the response code allows a modifier and no modifier has already been added.

```
2   #ifndef INLINE_RcSafeAddToResult
3   TPM_RC
4   RcSafeAddToResult(
5       TPM_RC          responseCode,
6       TPM_RC          modifier
7       )
8   {
9       if((responseCode & RC_FMT1) && !(responseCode & 0xf40))
10          return responseCode + modifier;
11      else
12          return responseCode;
13  }
14  #endif // INLINE_RcSafeAddToResult
```

9.17 TpmFail.c

9.17.1 Includes, Defines, and Types

```
#define     TPM_FAIL_C
#include    "Tpm.h"
#include    <assert.h>
```

On MS C compiler, can save the alignment state and set the alignment to 1 for the duration of the TpmTypes.h include. This will avoid a lot of alignment warnings from the compiler for the unaligned structures. The alignment of the structures is not important as this function does not use any of the structures in TpmTypes.h and only include it for the #defines of the capabilities, properties, and command code values.

```
#include "TpmTypes.h"
```

9.17.2 Typedefs

These defines are used primarily for sizing of the local response buffer.

```
typedef struct
{
    TPM_ST          tag;
    UINT32          size;
    TPM_RC          code;
} HEADER;
typedef struct
{
    BYTE            tag[sizeof(TPM_ST)];
    BYTE            size[sizeof(UINT32)];
    BYTE            code[sizeof(TPM_RC)];
} PACKED_HEADER;
typedef struct
{
    BYTE             size[sizeof(UINT16)];
    struct
    {
        BYTE         function[sizeof(UINT32)];
        BYTE         line[sizeof(UINT32)];
        BYTE         code[sizeof(UINT32)];
    } values;
    BYTE             returnCode[sizeof(TPM_RC)];
} GET_TEST_RESULT_PARAMETERS;
typedef struct
{
    BYTE         moreData[sizeof(TPMI_YES_NO)];
    BYTE         capability[sizeof(TPM_CAP)]; // Always TPM_CAP_TPM_PROPERTIES
    BYTE         tpmProperty[sizeof(TPML_TAGGED_TPM_PROPERTY)];
} GET_CAPABILITY_PARAMETERS;
typedef struct
{
    BYTE         header[sizeof(PACKED_HEADER)];
    BYTE         getTestResult[sizeof(GET_TEST_RESULT_PARAMETERS)];
} TEST_RESPONSE;
typedef struct
{
    BYTE         header[sizeof(PACKED_HEADER)];
    BYTE         getCap[sizeof(GET_CAPABILITY_PARAMETERS)];
} CAPABILITY_RESPONSE;
typedef union
{
```



```
46      BYTE        test[sizeof(TEST_RESPONSE)];
47      BYTE        cap[sizeof(CAPABILITY_RESPONSE)];
48  } RESPONSES;
```

Buffer to hold the responses. This may be a little larger than required due to padding that a compiler might add.

NOTE: This is not in Global.c because of the specialized data definitions above. Since the data contained in this structure is not relevant outside of the execution of a single command (when the TPM is in failure mode. There is no compelling reason to move all the typedefs to Global.h and this structure to Global.c.

```
49  #ifndef __IGNORE_STATE__ // Don't define this value
50  static BYTE response[sizeof(RESPONSES)];
51  #endif
```

9.17.3 Local Functions

9.17.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```
52  static INT32
53  MarshalUint16(
54      UINT16          integer,
55      BYTE            **buffer
56      )
57  {
58      return UINT16_Marshal(&integer, buffer, NULL);
59  }
```

9.17.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```
60  static INT32
61  MarshalUint32(
62      UINT32           integer,
63      BYTE            **buffer
64      )
65  {
66      return UINT32_Marshal(&integer, buffer, NULL);
67  }
```

9.17.3.3 UnmarshalHeader()

function to unmarshal the 10-byte command header.

```
68  static BOOL
69  UnmarshalHeader(
70      HEADER          *header,
71      BYTE            **buffer,
72      INT32           *size
73      )
74  {
75      UINT32 usize;
76      TPM_RC ucode;
77      if(UINT16_Unmarshal(&header->tag, buffer, size) != TPM_RC_SUCCESS
78         || UINT32_Unmarshal(&usize, buffer, size) != TPM_RC_SUCCESS
79         || UINT32_Unmarshal(&ucode, buffer, size) != TPM_RC_SUCCESS)
80          return FALSE;
```

```
81      header->size = usize;
82      header->code = ucode;
83      return TRUE;
84  }
```

9.17.4 Public Functions

```
85  #ifdef SIMULATION
```

9.17.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```
86  LIB_EXPORT void
87  SetForceFailureMode(
88      void
89      )
90  {
91      g_forceFailureMode = TRUE;
92      return;
93  }
94  #endif
```

9.17.4.2 TpmFail()

This function is called by TPM.lib when a failure occurs. It will set up the failure values to be returned on TPM2_GetTestResult().

```
95  NORETURN void
96  TpmFail(
97      const char      *function,
98      int              line,
99      int              code
100     )
101 {
102     // Save the values that indicate where the error occurred.
103     // On a 64-bit machine, this may truncate the address of the string
104     // of the function name where the error occurred.
105     s_failFunction = *(UINT32*)&function;
106     s_failLine = line;
107     s_failCode = code;
108     // We are in failure mode
109     g_inFailureMode = TRUE;
110     // if asserts are enabled, then do an assert unless the failure mode code
111     // is being tested.
112 #ifdef SIMULATION
113 #   ifndef NDEBUG
114     assert(g_forceFailureMode);
115 #   endif
116     // Clear this flag
117     g_forceFailureMode = FALSE;
118 #endif
119     // Jump to the failure mode code.
120     // Note: only get here if asserts are off or if we are testing failure mode
121     _plat__Fail();
122 }
```

9.17.5 TpmFailureMode

This function is called by the interface code when the platform is in failure mode.

```
123  void
124  TpmFailureMode(
125      unsigned int     inRequestSize,     // IN: command buffer size
126      unsigned char   *inRequest,         // IN: command buffer
127      unsigned int    *outResponseSize,   // OUT: response buffer size
128      unsigned char   **outResponse       // OUT: response buffer
129      )
130  {
131      BYTE            *buffer;
132      UINT32           marshalSize;
133      UINT32           capability;
134      HEADER           header;    // unmarshaled command header
135      UINT32           pt;    // unmarshaled property type
136      UINT32           count; // unmarshaled property count
137      // If there is no command buffer, then just return TPM_RC_FAILURE
138      if(inRequestSize == 0 || inRequest == NULL)
139          goto FailureModeReturn;
140      // If the header is not correct for TPM2_GetCapability() or
141      // TPM2_GetTestResult() then just return the in failure mode response;
142      buffer = inRequest;
143      if(!UnmarshalHeader(&header, &inRequest, (INT32 *)&inRequestSize))
144          goto FailureModeReturn;
145      if(header.tag != TPM_ST_NO_SESSIONS
146         || header.size < 10)
147          goto FailureModeReturn;
148      switch(header.code)
149      {
150          case TPM_CC_GetTestResult:
151              // make sure that the command size is correct
152              if(header.size != 10)
153                  goto FailureModeReturn;
154              buffer = &response[10];
155              marshalSize = MarshalUint16(3 * sizeof(UINT32), &buffer);
156              marshalSize += MarshalUint32(s_failFunction, &buffer);
157              marshalSize += MarshalUint32(s_failLine, &buffer);
158              marshalSize += MarshalUint32(s_failCode, &buffer);
159              if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
160                  marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
161              else
162                  marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
163              break;
164          case TPM_CC_GetCapability:
165              // make sure that the size of the command is exactly the size
166              // returned for the capability, property, and count
167              if(header.size != (10 + (3 * sizeof(UINT32)))
168                      // also verify that this is requesting TPM properties
169                 || TPM_RC_SUCCESS != UINT32_Unmarshal(&capability, &inRequest,
170                                                       (INT32 *)&inRequestSize)
171                 || capability != TPM_CAP_TPM_PROPERTIES
172                 || TPM_RC_SUCCESS != UINT32_Unmarshal(&pt, &inRequest,
173                                                       (INT32 *)&inRequestSize)
174                 || TPM_RC_SUCCESS != UINT32_Unmarshal(&count, &inRequest,
175                                                       (INT32 *)&inRequestSize))
176                  goto FailureModeReturn;
177              // If in failure mode because of an unrecoverable read error, and the
178              // property is 0 and the count is 0, then this is an indication to
179              // re-manufacture the TPM. Do the re-manufacture but stay in failure
180              // mode until the TPM is reset.
181              // Note: this behavior is not required by the specification and it is
182              // OK to leave the TPM permanently bricked due to an unrecoverable NV
183              // error.
184              if(count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
185              {
186                  g_manufactured = FALSE;
187                  TPM_Manufacture(0);
188              }
```

```
189             if(count > 0)
190                 count = 1;
191             else if(pt > TPM_PT_FIRMWARE_VERSION_2)
192                 count = 0;
193             if(pt < TPM_PT_MANUFACTURER)
194                 pt = TPM_PT_MANUFACTURER;
195             // set up for return
196             buffer = &response[10];
197             // if the request was for a PT less than the last one
198             // then we indicate more, otherwise, not.
199             if(pt < TPM_PT_FIRMWARE_VERSION_2)
200                 *buffer++ = YES;
201             else
202                 *buffer++ = NO;
203             marshalSize = 1;
204             // indicate the capability type
205             marshalSize += MarshalUint32(capability, &buffer);
206             // indicate the number of values that are being returned (0 or 1)
207             marshalSize += MarshalUint32(count, &buffer);
208             // indicate the property
209             marshalSize += MarshalUint32(pt, &buffer);
210             if(count > 0)
211                 switch(pt)
212                 {
213                     case TPM_PT_MANUFACTURER:
214                     // the vendor ID unique to each TPM manufacturer
215 #ifdef  MANUFACTURER
216                         pt = *(UINT32*)MANUFACTURER;
217 #else
218                         pt = 0;
219 #endif
220                         break;
221                     case TPM_PT_VENDOR_STRING_1:
222                         // the first four characters of the vendor ID string
223 #ifdef  VENDOR_STRING_1
224                         pt = *(UINT32*)VENDOR_STRING_1;
225 #else
226                         pt = 0;
227 #endif
228                         break;
229                     case TPM_PT_VENDOR_STRING_2:
230                         // the second four characters of the vendor ID string
231 #ifdef  VENDOR_STRING_2
232                         pt = *(UINT32*)VENDOR_STRING_2;
233 #else
234                         pt = 0;
235 #endif
236                         break;
237                     case TPM_PT_VENDOR_STRING_3:
238                         // the third four characters of the vendor ID string
239 #ifdef  VENDOR_STRING_3
240                         pt = *(UINT32*)VENDOR_STRING_3;
241 #else
242                         pt = 0;
243 #endif
244                         break;
245                     case TPM_PT_VENDOR_STRING_4:
246                         // the fourth four characters of the vendor ID string
247 #ifdef  VENDOR_STRING_4
248                         pt = *(UINT32*)VENDOR_STRING_4;
249 #else
250                         pt = 0;
251 #endif
252                         break;
253                     case TPM_PT_VENDOR_TPM_TYPE:
254                         // vendor-defined value indicating the TPM model
```

```
255                         // We just make up a number here
256                         pt = 1;
257                         break;
258                     case TPM_PT_FIRMWARE_VERSION_1:
259                         // the more significant 32-bits of a vendor-specific value
260                         // indicating the version of the firmware
261 #ifdef  FIRMWARE_V1
262                         pt = FIRMWARE_V1;
263 #else
264                         pt = 0;
265 #endif
266                         break;
267                     default: // TPM_PT_FIRMWARE_VERSION_2:
268                         // the less significant 32-bits of a vendor-specific value
269                         // indicating the version of the firmware
270 #ifdef  FIRMWARE_V2
271                         pt = FIRMWARE_V2;
272 #else
273                         pt = 0;
274 #endif
275                         break;
276                 }
277             marshalSize += MarshalUint32(pt, &buffer);
278             break;
279         default: // default for switch (cc)
280             goto FailureModeReturn;
281     }
282     // Now do the header
283     buffer = response;
284     marshalSize = marshalSize + 10; // Add the header size to the
285                                     // stuff already marshaled
286     MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
287     MarshalUint32(marshalSize, &buffer);  // responseSize
288     MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
289     *outResponseSize = marshalSize;
290     *outResponse = (unsigned char *)&response;
291     return;
292 FailureModeReturn:
293     buffer = response;
294     marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
295     marshalSize += MarshalUint32(10, &buffer);
296     marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
297     *outResponseSize = marshalSize;
298     *outResponse = (unsigned char *)response;
299     return;
300 }
```

9.17.6 UnmarshalFail()

This is a stub that is used to catch an attempt to unmarshal an entry that is not defined. Don't ever expect this to be called but...

```
301 void
302 UnmarshalFail(
303     void            *type,
304     BYTE            **buffer,
305     INT32           *size
306     )
307 {
308     NOT_REFERENCED(type);
309     NOT_REFERENCED(buffer);
310     NOT_REFERENCED(size);
311     FAIL(FATAL_ERROR_INTERNAL);
312 }
```

10 Cryptographic Functions

10.1 Headers

10.1.1 BnValues.h

10.1.1.1 Introduction

This file contains the definitions needed for defining the internal BIGNUM structure. A BIGNUM is a pointer to a structure. The structure has three fields. The last field is and array (d) of crypt_uword_t. Each word is in machine format (big- or little-endian) with the words in ascending significance (i.e. words in little-endian order). This is the order that seems to be used in every big number library in the worlds, so...

The first field in the structure (allocated) is the number of words in *d*. This is the upper limit on the size of the number that can be held in the structure. This differs from libraries like OpenSSL() as this is not intended to deal with numbers of arbitrary size; just numbers that are needed to deal with the algorithms that are defined in the TPM implementation.

The second field in the structure (size) is the number of significant words in *n*. When this number is zero, the number is zero. The word at used-1 should never be zero. All words between d[size] and d[allocated-1] should be zero.

10.1.1.2 Defines

```
#ifndef _BN_NUMBERS_H
#define _BN_NUMBERS_H
#if RADIX_BITS == 64
# define RADIX_LOG2         6
#elif RADIX_BITS == 32
#define RADIX_LOG2          5
#else
# error "Unsupported radix"
#endif
#define RADIX_MOD(x)        ((x) & ((1 << RADIX_LOG2) - 1))
#define RADIX_DIV(x)        ((x) >> RADIX_LOG2)
#define RADIX_MASK  ((((crypt_uword_t)1) << RADIX_LOG2) - 1)
#define BITS_TO_CRYPT_WORDS(bits)       RADIX_DIV((bits) + (RADIX_BITS - 1))
#define BYTES_TO_CRYPT_WORDS(bytes)     BITS_TO_CRYPT_WORDS(bytes * 8)
#define SIZE_IN_CRYPT_WORDS(thing)      BYTES_TO_CRYPT_WORDS(sizeof(thing))
#if RADIX_BITS == 64
#define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_64(x)
#define CRYPT_TO_BYTE_ARRAY UINT64_TO_BYTE_ARRAY
typedef uint64_t    crypt_uword_t;
typedef int64_t     crypt_word_t;
#elif RADIX_BITS == 32
#define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_32((x))
#define CRYPT_TO_BYTE_ARRAY UINT32_TO_BYTE_ARRAY
typedef uint32_t    crypt_uword_t;
typedef int32_t     crypt_word_t;
#endif
#define MAX_CRYPT_UWORD (~((crypt_uword_t)0))
#define MAX_CRYPT_WORD  ((crypt_word_t)(MAX_CRYPT_UWORD >> 1))
#define MIN_CRYPT_WORD  (~MAX_CRYPT_WORD)
#define LARGEST_NUMBER (MAX((ALG_RSA * MAX_RSA_KEY_BYTES),                      \
                            MAX((ALG_ECC * MAX_ECC_KEY_BYTES), MAX_DIGEST_SIZE)))
#define LARGEST_NUMBER_BITS (LARGEST_NUMBER * 8)
#define MAX_ECC_PARAMETER_BYTES (MAX_ECC_KEY_BYTES * ALG_ECC)
```


These are the basic big number formats. This is convertible to the library- specific format without to much difficulty. For the math performed using these numbers, the value is always positive.

```
34  #define BN_STRUCT_DEF(count) struct {       \
35      crypt_uword_t       allocated;          \
36      crypt_uword_t       size;               \
37      crypt_uword_t       d[count];           \
38      }
39  typedef BN_STRUCT_DEF(1) bignum_t;
40  #ifndef bigNum
41  typedef bignum_t       *bigNum;
42  typedef const bignum_t *bigConst;
43  #endif
44  extern const bignum_t   BnConstZero;
```

The Functions to access the properties of a big number. Get number of allocated words

```
45  #define BnGetAllocated(x)   (unsigned)((x)->allocated)
```

Get number of words used

```
46  #define BnGetSize(x)        ((x)->size)
```

Get a pointer to the data array

```
47  #define BnGetArray(x)       ((crypt_uword_t *)&((x)->d[0]))
```

Get the nth word of a BIGNUM (zero-based)

```
48  #define BnGetWord(x, i)     (crypt_uword_t)((x)->d[i])
```

Some things that are done often. Test to see if a bignum_t is equal to zero

```
49  #define BnEqualZero(bn)   (BnGetSize(bn) == 0)
```

Test to see if a bignum_t is equal to a word type

```
50  #define BnEqualWord(bn, word)                           \
51              ((BnGetSize(bn) == 1) && (BnGetWord(bn, 0) == (crypt_uword_t)word))
```

Determine if a BIGNUM is even. A zero is even. Although the indication that a number is zero is that it's size is zero, all words of the number are 0 so this test works on zero.

```
52  #define BnIsEven(n)     ((BnGetWord(n, 0) & 1) == 0)
```

The macros below are used to define BIGNUM values of the required size. The values are allocated on the stack so they can be treated like simple local values. This will call the initialization function for a defined bignum_t. This sets the allocated and used fields and clears the words of *n*.

```
53  #define BN_INIT(name)                                           \
54      (bigNum)BnInit((bigNum)&(name),                             \
55                     BYTES_TO_CRYPT_WORDS(sizeof(name.d)))
```

In some cases, a function will need the address of the structure associated with a variable. The structure for a BIGNUM variable of *name* is *name_*. Generally, when the structure is created, it is initialized and a parameter is created with a pointer to the structure. The pointer has the *name* and the structure it points to is *name_*

```
56  #define BN_ADDRESS(name) (bigNum)&name##_
57  #define BN_STRUCT_ALLOCATION(bits) (BITS_TO_CRYPT_WORDS(bits) + 1)
```

Create a structure of the correct size.

```
58  #define BN_STRUCT(bits)                                         \
59      BN_STRUCT_DEF(BN_STRUCT_ALLOCATION(bits))
```

Define a BIGNUM type with a specific allocation

```
60  #define BN_TYPE(name, bits)                                     \
61      typedef BN_STRUCT(bits) bn_##name##_t
```

This creates a local BIGNUM variable of a specific size and initializes it from a TPM2B input parameter.

```
62  #define BN_INITIALIZED(name, bits, initializer)                 \
63      BN_STRUCT(bits)  name##_;                                   \
64      bigNum           name = BnFrom2B(BN_INIT(name##_),          \
65                                      (const TPM2B *)initializer)
```

Create a local variable that can hold a number with *bits*

```
66  #define BN_VAR(name, bits)                                      \
67      BN_STRUCT(bits)  _##name;                                    \
68      bigNum           name = BN_INIT(_##name)
```

Create a type that can hold the largest number defined by the implementation.

```
69  #define BN_MAX(name)   BN_VAR(name, LARGEST_NUMBER_BITS)
70  #define BN_MAX_INITIALIZED(name, initializer)                   \
71      BN_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)
```

A word size value is useful

```
72  #define BN_WORD(name)      BN_VAR(name, RADIX_BITS)
```

This is used to created a word-size BIGNUM and initialize it with an input parameter to a function.

```
73  #define BN_WORD_INITIALIZED(name, initial)                              \
74      BN_STRUCT(RADIX_BITS)  name##_;                                     \
75      bigNum                 name = BnInitializeWord((bigNum)&name##_,    \
76                                   BN_STRUCT_ALLOCATION(RADIX_BITS), initial)
```

ECC-Specific Values This is the format for a point. It is always in affine format. The Z value is carried as part of the point, primarily to simplify the interface to the support library. Rather than have the interface layer have to create space for the point each time it is used... The x, y, and z values are pointers to *bigNum* values and not in-line versions of the numbers. This is a relic of the days when there was no standard TPM format for the numbers

```
77  typedef struct _bn_point_t
78  {
79      bigNum          x;
80      bigNum          y;
81      bigNum          z;
82  } bn_point_t;
83  typedef bn_point_t          *bigPoint;
84  typedef const bn_point_t    *pointConst;
85  typedef struct constant_point_t
86  {
87      bigConst        x;
88      bigConst        y;
89      bigConst        z;
90  } constant_point_t;
91  #define ECC_BITS    (MAX_ECC_KEY_BYTES * 8)
```

```
BN_TYPE(ecc, ECC_BITS);
#define ECC_NUM(name)       BN_VAR(name, ECC_BITS)
#define ECC_INITIALIZED(name, initializer)                          \
    BN_INITIALIZED(name, ECC_BITS, initializer)
#define POINT_INSTANCE(name, bits)                                  \
    BN_STRUCT (bits)    name##_x =                                  \
                  {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}};            \
    BN_STRUCT ( bits )    name##_y =                                \
                  {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}};            \
    BN_STRUCT ( bits )    name##_z =                                \
                  {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}};            \
    bn_point_t name##_
#define POINT_INITIALIZER(name)                                      \
    BnInitializePoint(&name##_, (bigNum)&name##_x,                  \
                      (bigNum)&name##_y, (bigNum)&name##_z)
#define POINT_INITIALIZED(name, initValue)                          \
    POINT_INSTANCE(name, MAX_ECC_KEY_BITS);                         \
    bigPoint            name = BnPointFrom2B(                       \
                                    POINT_INITIALIZER(name),        \
                                    initValue)
#define POINT_VAR(name, bits)                                       \
    POINT_INSTANCE (name, bits);                                   \
    bigPoint           name = POINT_INITIALIZER(name)
#define POINT(name)     POINT_VAR(name, MAX_ECC_KEY_BITS)
```

Structure for the curve parameters. This is an analog to the TPMS_ALGORITHM_DETAIL_ECC

```
typedef struct
{
    bigConst            prime;     // a prime number
    bigConst            order;     // the order of the curve
    bigConst            h;         // cofactor
    bigConst            a;         // linear coefficient
    bigConst            b;         // constant term
    constant_point_t    base;      // base point
} ECC_CURVE_DATA;
```

Access macros for the ECC_CURVE structure. The parameter *C* is a pointer to an ECC_CURVE_DATA structure. In some libraries, the curve structure contains a pointer to an ECC_CURVE_DATA structure as well as some other bits. For those cases, the AccessCurveData() macro is used in the code to first get the pointer to the ECC_CURVE_DATA for access. In some cases, the macro does noting.

```
#define CurveGetPrime(C)    ((C)->prime)
#define CurveGetOrder(C)    ((C)->order)
#define CurveGetCofactor(C) ((C)->h)
#define CurveGet_a(C)       ((C)->a)
#define CurveGet_b(C)       ((C)->b)
#define CurveGetG(C)        ((pointConst)&((C)->base))
#define CurveGetGx(C)       ((C)->base.x)
#define CurveGetGy(C)       ((C)->base.y)
```

Convert bytes in initializers according to the endianess of the system This is used for BnEccData.c.

NOTE: There is no reason to try to optimize this by doing byte at a time shifts because this is handled at compile time.

```
#define     BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)                  \
            (    ((UINT32)(a) << 24)                                \
            +    ((UINT32)(b) << 16)                                \
            +    ((UINT32)(c) << 8)                                 \
            +    ((UINT32)(d))                                      \
            )
#define     BIG_ENDIAN_BYTES_TO_UINT64(a, b, c, d, e, f, g, h)      \
            (    ((UINT64)(a) << 56)                                \
            +    ((UINT64)(b) << 48)                                \
```

```
142             +    ((UINT64)(c) << 40)                                    \
143             +    ((UINT64)(d) << 32)                                    \
144             +    ((UINT64)(e) << 24)                                    \
145             +    ((UINT64)(f) << 16)                                    \
146             +    ((UINT64)(g) << 8)                                     \
147             +    ((UINT64)(h))                                          \
148             )
149   #if RADIX_BITS > 32
150   #  define TO_CRYPT_WORD_64(a, b, c, d, e, f, g, h)                     \
151              BIG_ENDIAN_BYTES_TO_UINT64(a, b, c, d, e, f, g, h)
152   #  define TO_CRYPT_WORD_32(a, b, c, d) TO_CRYPT_WORD_64(0, 0, 0, 0, a, b, c, d)
153   #else
154   #  define TO_CRYPT_WORD_64(a, b, c, d, e, f, g, h)                     \
155           BIG_ENDIAN_BYTES_TO_UINT32(e, f, g, h),                       \
156           BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)
157   #  define TO_CRYPT_WORD_32 BIG_ENDIAN_BYTES_TO_UINT32
158   #endif
```

Add implementation dependent definitions for other ECC Values and for linkages. MATH_LIB_H is defined in LibSupport.h

```
159   #include MATHLIB_H
160   #ifndef RADIX_BYTES
161   #   if RADIX_BITS == 32
162   #       define RADIX_BYTES 4
163   #   elif RADIX_BITS == 64
164   #       define RADIX_BYTES 8
165   #   else
166   #       error "RADIX_BITS must either be 32 or 64"
167   #   endif
168   #endif
169   #endif // _BN_NUMBERS_H
```

10.1.2 CryptEcc.h

10.1.2.1 Introduction

This file contains structure definitions used for ECC. The structures in this file are only used internally. The ECC-related structures that cross the TPM interface are defined in TpmTypes.h

```
1   #ifndef _CRYPT_ECC_H
2   #define _CRYPT_ECC_H
```

10.1.2.2 ECC-related Structures

This is used to define the macro that may or may not be in the data set for the curve (BnEccData.c). If there is a mismatch, the compiler will warn that there is to much/not enough initialization data in the curve. The macro is used because not all versions of the CryptEccData.c need the curve name.

```
3   #ifdef NAMED_CURVES
4   #define CURVE_NAME(a) , a
5   #define CURVE_NAME_DEF const char *name;
6   #else
7   #  define CURVE_NAME(a)
8   #  define CURVE_NAME_DEF
9   #endif
10  typedef struct ECC_CURVE
11  {
12      const TPM_ECC_CURVE         curveId;
13      const UINT16                keySizeBits;
14      const TPMT_KDF_SCHEME       kdf;
```

```
15      const TPMT_ECC_SCHEME        sign;
16      const ECC_CURVE_DATA        *curveData; // the address of the curve data
17      CURVE_NAME_DEF
18  } ECC_CURVE;
19  extern const ECC_CURVE eccCurves[ECC_CURVE_COUNT];
20  #endif
```

10.1.3 CryptHash.h

10.1.3.1 Hash-related Structures

```
 1  typedef struct
 2  {
 3      const TPM_ALG_ID     alg;
 4      const UINT16         digestSize;
 5      const UINT16         blockSize;
 6      const UINT16         derSize;
 7      const BYTE           der[20];
 8  } HASH_INFO;
 9  typedef union
10  {
11  #ifdef TPM_ALG_SHA1
12      tpmHashStateSHA1_t         Sha1;
13  #endif
14  #ifdef TPM_ALG_SHA256
15      tpmHashStateSHA256_t       Sha256;
16  #endif
17  #ifdef TPM_ALG_SHA384
18      tpmHashStateSHA384_t       Sha384;
19  #endif
20  #ifdef TPM_ALG_SHA512
21      tpmHashStateSHA512_t       Sha512;
22  #endif
23      // to force structure alignment to be no worse than HASH_ALIGNMENT
24  #if HASH_ALIGNMENT == 4
25      uint32_t             align;
26  #else
27      uint64_t             align;
28  #endif
29  } ANY_HASH_STATE;
30  typedef ANY_HASH_STATE *PANY_HASH_STATE;
31  typedef const ANY_HASH_STATE    *PCANY_HASH_STATE;
32  #define ALIGNED_SIZE(x, b) ((((x) + (b) - 1) / (b)) * (b))
```

MAX_HASH_STATE_SIZE will change with each implementation. It is assumed that a hash state will not be larger than twice the block size plus some overhead (in this case, 16 bytes). The overall size needs to be as large as any of the hash contexts. The structure needs to start on an alignment boundary and be an even multiple of the alignment

```
33  #define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
34  #define MAX_HASH_STATE_SIZE_ALIGNED                                    \
35                      ALIGNED_SIZE(MAX_HASH_STATE_SIZE, HASH_ALIGNMENT)
```

This is an aligned byte array that will hold any of the hash contexts.

```
36  typedef  ANY_HASH_STATE ALIGNED_HASH_STATE;
```

The header associated with the hash library is expected to define the methods which include the calling sequence. When not compiling CryptHash.c, the methods are not defined so we need placeholder functions for the structures

```
37  #ifndef HASH_START_METHOD_DEF
38  #   define HASH_START_METHOD_DEF    void (HASH_START_METHOD)(void)
39  #endif
40  #ifndef HASH_DATA_METHOD_DEF
41  #   define HASH_DATA_METHOD_DEF     void (HASH_DATA_METHOD)(void)
42  #endif
43  #ifndef HASH_END_METHOD_DEF
44  #   define HASH_END_METHOD_DEF      void (HASH_END_METHOD)(void)
45  #endif
46  #ifndef HASH_STATE_COPY_METHOD_DEF
47  #   define HASH_STATE_COPY_METHOD_DEF     void (HASH_STATE_COPY_METHOD)(void)
48  #endif
49  #ifndef  HASH_STATE_EXPORT_METHOD_DEF
50  #   define  HASH_STATE_EXPORT_METHOD_DEF   void (HASH_STATE_EXPORT_METHOD)(void)
51  #endif
52  #ifndef  HASH_STATE_IMPORT_METHOD_DEF
53  #   define  HASH_STATE_IMPORT_METHOD_DEF   void ( HASH_STATE_IMPORT_METHOD)(void)
54  #endif
```

Define the prototypical function call for each of the methods. This defines the order in which the parameters are passed to the underlying function.

```
55  typedef HASH_START_METHOD_DEF;
56  typedef HASH_DATA_METHOD_DEF;
57  typedef HASH_END_METHOD_DEF;
58  typedef HASH_STATE_COPY_METHOD_DEF;
59  typedef HASH_STATE_EXPORT_METHOD_DEF;
60  typedef HASH_STATE_IMPORT_METHOD_DEF;
61  typedef struct _HASH_METHODS
62  {
63      HASH_START_METHOD           *start;
64      HASH_DATA_METHOD            *data;
65      HASH_END_METHOD             *end;
66      HASH_STATE_COPY_METHOD      *copy;          // Copy a hash block
67      HASH_STATE_EXPORT_METHOD    *copyOut;       // Copy a hash block from a hash
68                                                  // context
69      HASH_STATE_IMPORT_METHOD    *copyIn;        // Copy a hash block to a proper hash
70                                                  // context
71  } HASH_METHODS, *PHASH_METHODS;
72  #if ALG_SHA1
73      TPM2B_TYPE(SHA1_DIGEST, SHA1_DIGEST_SIZE);
74  #endif
75  #if ALG_SHA256
76      TPM2B_TYPE(SHA256_DIGEST, SHA256_DIGEST_SIZE);
77  #endif
78  #if ALG_SHA384
79      TPM2B_TYPE(SHA384_DIGEST, SHA384_DIGEST_SIZE);
80  #endif
81  #if ALG_SHA512
82      TPM2B_TYPE(SHA512_DIGEST, SHA512_DIGEST_SIZE);
83  #endif
84  #if ALG_SM3_256
85      TPM2B_TYPE(SM3_256_DIGEST, SM3_256_DIGEST_SIZE);
86  #endif
87  typedef const struct
88  {
89      HASH_METHODS         method;
90      uint16_t             blockSize;
91      uint16_t             digestSize;
92      uint16_t             contextSize;
93      uint16_t             hashAlg;
94  } HASH_DEF, *PHASH_DEF;
```

Macro to fill in the HASH_DEF for an algorithm. For SHA1, the instance would be: HASH_DEF_TEMPLATE(Sha1, SHA1) This handles the difference in capitalization for the various pieces.

```
#define HASH_DEF_TEMPLATE(HASH)                                                 \
    HASH_DEF    HASH##_Def= {                                                   \
                        {(HASH_START_METHOD *)&tpmHashStart_##HASH,             \
                         (HASH_DATA_METHOD *)&tpmHashData_##HASH,               \
                         (HASH_END_METHOD *)&tpmHashEnd_##HASH,                 \
                         (HASH_STATE_COPY_METHOD *)&tpmHashStateCopy_##HASH,    \
                         (HASH_STATE_EXPORT_METHOD *)&tpmHashStateExport_##HASH,\
                         (HASH_STATE_IMPORT_METHOD *)&tpmHashStateImport_##HASH,\
                        },                                                      \
                        HASH##_BLOCK_SIZE,      /*block size */                \
                        HASH##_DIGEST_SIZE,     /*data size */                 \
                        sizeof(tpmHashState##HASH##_t),                         \
                        TPM_ALG_##HASH}
```

These definitions are for the types that can be in a hash state structure. These types are used in the crypto utilities. This is a define rather than an enum so that the size of this field can be explicit.

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

This is the structure that is used for passing a context into the hashing functions. It should be the same size as the function context used within the hashing functions. This is checked when the hash function is initialized. This version uses a new layout for the contexts and a different definition. The state buffer is an array of HASH_UNIT values so that a decent compiler will put the structure on a HASH_UNIT boundary. If the structure is not properly aligned, the code that manipulates the structure will copy to a properly aligned structure before it is used and copy the result back. This just makes things slower.

```
typedef struct _HASH_STATE
{
    PHASH_DEF               def;
    TPM_ALG_ID              hashAlg;
    HASH_STATE_TYPE         type;               // type of the context
    ANY_HASH_STATE          state;
} HASH_STATE, *PHASH_STATE;
typedef const HASH_STATE *PCHASH_STATE;
```

10.1.3.2 HMAC State Structures

This header contains the hash structure definitions used in the TPM code to define the amount of space to be reserved for the hash state. This allows the TPM code to not have to import all of the symbols used by the hash computations. This lets the build environment of the TPM code not to have include the header files associated with the CryptoEngine() code.

```
#ifndef _CRYPT_HASH_H
#define _CRYPT_HASH_H
```

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.

```
typedef struct
{
    HASH_STATE          hashState;          // the hash state
    TPM2B_HASH_BLOCK    hmacKey;            // the HMAC key
} HMAC_STATE, *PHMAC_STATE;
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 run time check that makes sure that this i.

```
128  typedef struct
129  {
130      BYTE                    buffer[sizeof(HASH_STATE)];
131  } EXPORT_HASH_STATE, *PEXPORT_HASH_STATE;
132  typedef const EXPORT_HASH_STATE *PCEXPORT_HASH_STATE;
133  #endif // _ALL_HASH_H
```

10.1.4 CryptHashData.h

```
  1  #ifdef GLOBAL_C
  2  const HASH_INFO   g_hashData[HASH_COUNT + 1] = {
  3  #ifdef  TPM_ALG_SHA1
  4      {TPM_ALG_SHA1,    SHA1_DIGEST_SIZE,   SHA1_BLOCK_SIZE,
  5      SHA1_DER_SIZE,   {SHA1_DER}},
  6  #endif
  7  #ifdef  TPM_ALG_SHA256
  8      {TPM_ALG_SHA256,    SHA256_DIGEST_SIZE,   SHA256_BLOCK_SIZE,
  9      SHA256_DER_SIZE,   {SHA256_DER}},
 10  #endif
 11  #ifdef  TPM_ALG_SHA512
 12      {TPM_ALG_SHA512,    SHA512_DIGEST_SIZE,   SHA512_BLOCK_SIZE,
 13      SHA512_DER_SIZE,   {SHA512_DER}},
 14  #endif
 15  #ifdef  TPM_ALG_SHA384
 16      {TPM_ALG_SHA384,    SHA384_DIGEST_SIZE,   SHA384_BLOCK_SIZE,
 17      SHA384_DER_SIZE,   {SHA384_DER}},
 18  #endif
 19  #ifdef  TPM_ALG_SM3_256
 20      {TPM_ALG_SM3_256,    SM3_256_DIGEST_SIZE,   SM3_256_BLOCK_SIZE,
 21      SM3_256_DER_SIZE,   {SM3_256_DER}},
 22  #endif
 23          {TPM_ALG_NULL,0,0,0,{0}}
 24      };
 25  #endif // GLOBAL_C
```

10.1.5 CryptRand.h

This file contains constant definition shared by CryptUtil() and the parts of the Crypto Engine.

```
1  #ifndef _CRYPT_RAND_H
2  #define _CRYPT_RAND_H
```

DRBG Structures and Defines 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 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

```
3  #define DRBG_KEY_SIZE_BITS      MAX_AES_KEY_BITS
4  #define DRBG_IV_SIZE_BITS       (MAX_AES_BLOCK_SIZE_BYTES * 8)
5  #define DRBG_ALGORITHM          TPM_ALG_AES
6  typedef tpmKeyScheduleAES     DRBG_KEY_SCHEDULE;
7  #define DRBG_ENCRYPT_SETUP(key, keySizeInBits, schedule)        \
8              TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)
```

```
#define DRBG_ENCRYPT(keySchedule, in, out)                      \
            TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out))
#if     ((DRBG_KEY_SIZE_BITS % RADIX_BITS) != 0) \
    || ((DRBG_IV_SIZE_BITS % RADIX_BITS) != 0)
#error "Key size and IV for DRBG must be even multiples of the radix"
#endif
#if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
#error "Key size for DRBG must be even multiple of the cypher block size"
#endif
```

Derived values

```
#define DRBG_MAX_REQUESTS_PER_RESEED (1 << 48)
#define DRBG_MAX_REQEST_SIZE (1 << 32)
#define pDRBG_KEY(seed)    ((DRBG_KEY *)&(((BYTE *)(seed))[0]))
#define pDRBG_IV(seed)     ((DRBG_IV *)&(((BYTE *)(seed))[DRBG_KEY_SIZE_BYTES]))
#define DRBG_KEY_SIZE_WORDS     (BITS_TO_CRYPT_WORDS(DRBG_KEY_SIZE_BITS))
#define DRBG_KEY_SIZE_BYTES     (DRBG_KEY_SIZE_WORDS * RADIX_BYTES)
#define DRBG_IV_SIZE_WORDS      (BITS_TO_CRYPT_WORDS(DRBG_IV_SIZE_BITS))
#define DRBG_IV_SIZE_BYTES      (DRBG_IV_SIZE_WORDS * RADIX_BYTES)
#define DRBG_SEED_SIZE_WORDS    (DRBG_KEY_SIZE_WORDS + DRBG_IV_SIZE_WORDS)
#define DRBG_SEED_SIZE_BYTES    (DRBG_KEY_SIZE_BYTES + DRBG_IV_SIZE_BYTES)
typedef union
{
    BYTE            bytes[DRBG_KEY_SIZE_BYTES];
    crypt_uword_t   words[1];
} DRBG_KEY;
typedef union
{
    BYTE        bytes[DRBG_IV_SIZE_BYTES];
    crypt_uword_t   words[1];
} DRBG_IV;
typedef union
{
    BYTE        bytes[DRBG_SEED_SIZE_BYTES];
    crypt_uword_t   words[1];
} DRBG_SEED;
#define CTR_DRBG_MAX_REQUESTS_PER_RESEED        ((UINT64)1 << 20)
#define CTR_DRBG_MAX_BYTES_PER_REQUEST          (1 << 16)
#   define CTR_DRBG_MIN_ENTROPY_INPUT_LENGTH    DRBG_SEED_SIZE_BYTES
#   define CTR_DRBG_MAX_ENTROPY_INPUT_LENGTH    DRBG_SEED_SIZE_BYTES
#   define CTR_DRBG_MAX_ADDITIONAL_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
#define     TESTING         (1 << 0)
#define     ENTROPY         (1 << 1)
#define     TESTED          (1 << 2)
#define     IsTestStateSet(BIT)    ((g_cryptoSelfTestState.rng & BIT) != 0)
#define     SetTestStateBit(BIT)   (g_cryptoSelfTestState.rng |= BIT)
#define     ClearTestStateBit(BIT) (g_cryptoSelfTestState.rng &= ~BIT)
#define     IsSelfTest()    IsTestStateSet(TESTING)
#define     SetSelfTest()   SetTestStateBit(TESTING)
#define     ClearSelfTest() ClearTestStateBit(TESTING)
#define     IsEntropyBad()      IsTestStateSet(ENTROPY)
#define     SetEntropyBad()     SetTestStateBit(ENTROPY)
#define     ClearEntropyBad()   ClearTestStateBit(ENTROPY)
#define     IsDrbgTested()      IsTestStateSet(TESTED)
#define     SetDrbgTested()     SetTestStateBit(TESTED)
#define     ClearDrbgTested()   ClearTestStateBit(TESTED)
typedef struct
{
    UINT64      reseedCounter;
    UINT32      magic;
    DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
    UINT32      lastValue[4];   // used when the TPM does continuous self-test
                                // for FIPS compliance of DRBG
} DRBG_STATE, *pDRBG_STATE;
```

```
71  #define DRBG_MAGIC   ((UINT32) 0x47425244) // "DRBG" backwards so that it displays
72  typedef struct
73  {
74      UINT64               counter;
75      UINT32               magic;
76      TPM2B               *seed;
77      const TPM2B         *label;
78      TPM2B               *context;
79      TPM_ALG_ID           hash;
80      TPM_ALG_ID           kdf;
81  } KDF_STATE, *pKDR_STATE;
82  #define KDF_MAGIC    ((UINT32) 0x4048444a) // "KDF " backwards
```

Make sure that any other structures added to this union start with a 64-bit counter and a 32-bit magic number

```
83  typedef union
84  {
85      DRBG_STATE      drbg;
86      KDF_STATE       kdf;
87  } RAND_STATE;
```

This is the state used when the library uses a random number generator. A special function is installed for the library to call. That function picks up the state from this location and uses it for the generation of the random number.

```
88  extern RAND_STATE           *s_random;
```

When instrumenting RSA key sieve

```
89  #ifdef  RSA_INSTRUMENT
90  #define PRIME_INDEX(x)  ((x) == 512 ? 0 : (x) == 1024 ? 1 : 2)
91  #   define INSTRUMENT_SET(a, b) ((a) = (b))
92  #   define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
93  #   define INSTRUMENT_INC(a)    (a) = (a) + 1
94  extern UINT32  PrimeIndex;
95  extern UINT32  failedAtIteration[10];
96  extern UINT32  PrimeCounts[3];
97  extern UINT32  MillerRabinTrials[3];
98  extern UINT32  totalFieldsSieved[3];
99  extern UINT32  bitsInFieldAfterSieve[3];
100 extern UINT32  emptyFieldsSieved[3];
101 extern UINT32  noPrimeFields[3];
102 extern UINT32  primesChecked[3];
103 extern UINT16  lastSievePrime;
104 #else
105 #   define INSTRUMENT_SET(a, b)
106 #   define INSTRUMENT_ADD(a, b)
107 #   define INSTRUMENT_INC(a)
108 #endif
109 #endif // _CRYPT_RAND_H
```

10.1.6 CryptRsa.h

```
1   #ifndef _CRYPT_RSA_H
2   #define _CRYPT_RSA_H
```

This structure is a succinct representation of the cryptographic components of an RSA key. It is used in testing

```
3   typedef struct
4   {
```

```
    UINT32        exponent;      // The public exponent pointer
    TPM2B        *publicKey;     // Pointer to the public modulus
    TPM2B        *privateKey;    // The private prime
} RSA_KEY;
```

These values are used in the *bigNum* representation of various RSA values.

```
#define RSA_BITS            (MAX_RSA_KEY_BYTES * 8)
BN_TYPE(rsa, RSA_BITS);
#define BN_RSA(name)       BN_VAR(name, RSA_BITS)
#define BN_RSA_INITIALIZED(name, initializer)                    \
    BN_INITIALIZED(name, RSA_BITS, initializer)
#define BN_PRIME(name)     BN_VAR(name, (RSA_BITS / 2))
BN_TYPE(prime, (RSA_BITS / 2));
#define BN_PRIME_INITIALIZED(name, initializer)                  \
    BN_INITIALIZED(name, RSA_BITS / 2, initializer)
typedef struct privateExponent
{
#if CRT_FORMAT_RSA == NO
    bn_rsa_t            D;
#else
    bn_prime_t          Q;
    bn_prime_t          dP;
    bn_prime_t          dQ;
    bn_prime_t          qInv;
#endif // CRT_FORMAT_RSA
} privateExponent_t;
#endif      // _CRYPT_RSA_H
```

10.1.7 CryptTest.h

10.1.7.1 Introduction

This file contains constant definition shared by CryptUtil() and the parts of the Crypto Engine.

```
#ifndef _CRYPT_TEST_H
#define _CRYPT_TEST_H
```

This is the definition of a bit array with one bit per algorithm

```
typedef BYTE    ALGORITHM_VECTOR[(ALG_LAST_VALUE + 7) / 8];
#ifdef  TEST_SELF_TEST
LIB_EXPORT    extern  ALGORITHM_VECTOR    LibToTest;
#endif
```

10.1.7.2 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;
```

10.1.7.3 g_cryptoSelfTestState

This structure contains the crypto self-test state values.

```
extern CRYPTO_SELF_TEST_STATE   g_cryptoSelfTestState;
#endif // _CRYPT_TEST_H
```

10.1.8 HashTestData.h

Hash Test Vectors

```
TPM2B_TYPE(HASH_TEST_KEY, 96); // Twice the largest digest size
TPM2B_HASH_TEST_KEY        c_hashTestKey = {{96, {
  0xa0,0xed,0x5c,0x9a,0xd2,0x4a,0x21,0x40,0x1a,0xd0,0x81,0x47,0x39,0x63,0xf9,0x50,
  0xdc,0x59,0x47,0x11,0x40,0x13,0x99,0x92,0xc0,0x72,0xa4,0x0f,0xe2,0x33,0xe4,0x63,
  0x9b,0xb6,0x76,0xc3,0x1e,0x6f,0x13,0xee,0xcc,0x99,0x71,0xa5,0xc0,0xcf,0x9a,0x40,
  0xcf,0xdb,0x66,0x70,0x05,0x63,0x54,0x12,0x25,0xf4,0xe0,0x1b,0x23,0x35,0xe3,0x70,
  0x7d,0x19,0x5f,0x00,0xe4,0xf1,0x61,0x73,0x05,0xd8,0x58,0x7f,0x60,0x61,0x84,0x36,
  0xec,0xbe,0x96,0x1b,0x69,0x00,0xf0,0x9a,0x6e,0xe3,0x26,0x73,0x0d,0x17,0x5b,0x33
  }}};
TPM2B_TYPE(HASH_TEST_DATA, 256); // Twice the largest block size
TPM2B_HASH_TEST_DATA       c_hashTestData = {{256, {
  0x88,0xac,0xc3,0xe5,0x5f,0x66,0x9d,0x18,0x80,0xc9,0x7a,0x9c,0xa4,0x08,0x90,0x98,
  0x0f,0x3a,0x53,0x92,0x4c,0x67,0x4e,0xb7,0x37,0xec,0x67,0x87,0xb6,0xbe,0x10,0xca,
  0x11,0x5b,0x4a,0x0b,0x45,0xc3,0x32,0x68,0x48,0x69,0xce,0x25,0x1b,0xc8,0xaf,0x44,
  0x79,0x22,0x83,0xc8,0xfb,0xe2,0x63,0x94,0xa2,0x3c,0x59,0x3e,0x3e,0xc6,0x64,0x2c,
  0x1f,0x8c,0x11,0x93,0x24,0xa3,0x17,0xc5,0x2f,0x37,0xcf,0x95,0x97,0x8e,0x63,0x39,
  0x68,0xd5,0xca,0xba,0x18,0x37,0x69,0x6e,0x4f,0x19,0xfd,0x8a,0xc0,0x8d,0x87,0x3a,
  0xbc,0x31,0x42,0x04,0x05,0xef,0xb5,0x02,0xef,0x1e,0x92,0x4b,0xb7,0x73,0x2c,0x8c,
  0xeb,0x23,0x13,0x81,0x34,0xb9,0xb5,0xc1,0x17,0x37,0x39,0xf8,0x3e,0xe4,0x4c,0x06,
  0xa8,0x81,0x52,0x2f,0xef,0xc9,0x9c,0x69,0x89,0xbc,0x85,0x9c,0x30,0x16,0x02,0xca,
  0xe3,0x61,0xd4,0x0f,0xed,0x34,0x1b,0xca,0xc1,0x1b,0xd1,0xfa,0xc1,0xa2,0xe0,0xdf,
  0x52,0x2f,0x0b,0x4b,0x9f,0x0e,0x45,0x54,0xb9,0x17,0xb6,0xaf,0xd6,0xd5,0xca,0x90,
  0x29,0x57,0x7b,0x70,0x50,0x94,0x5c,0x8e,0xf6,0x4e,0x21,0x8b,0xc6,0x8b,0xa6,0xbc,
  0xb9,0x64,0xd4,0x4d,0xf3,0x68,0xd8,0xac,0xde,0xd8,0xd8,0xb5,0x6d,0xcd,0x93,0xeb,
  0x28,0xa4,0xe2,0x5c,0x44,0xef,0xf0,0xe1,0x6f,0x38,0x1a,0x3c,0xe6,0xef,0xa2,0x9d,
  0xb9,0xa8,0x05,0x2a,0x95,0xec,0x5f,0xdb,0xb0,0x25,0x67,0x9c,0x86,0x7a,0x8e,0xea,
  0x51,0xcc,0xc3,0xd3,0xff,0x6e,0xf0,0xed,0xa3,0xae,0xf9,0x5d,0x33,0x70,0xf2,0x11
  }}};
#if ALG_SHA1 == YES
TPM2B_TYPE(SHA1, 20);
TPM2B_SHA1    c_SHA1_digest = {{20, {
  0xee,0x2c,0xef,0x93,0x76,0xbd,0xf8,0x91,0xbc,0xe6,0xe5,0x57,0x53,0x77,0x01,0xb5,
  0x70,0x95,0xe5,0x40
  }}};
#endif
#if ALG_SHA384 == YES
TPM2B_TYPE(SHA384, 48);
TPM2B_SHA384    c_SHA384_digest = {{48, {
  0x37,0x75,0x29,0xb5,0x20,0x15,0x6e,0xa3,0x7e,0xa3,0x0d,0xcd,0x80,0xa8,0xa3,0x3d,
  0xeb,0xe8,0xad,0x4e,0x1c,0x77,0x94,0x5a,0xaf,0x6c,0xd0,0xc1,0xfa,0x43,0x3f,0xc7,
  0xb8,0xf1,0x01,0xc0,0x60,0xbf,0xf2,0x87,0xe8,0x71,0x9e,0x51,0x97,0xa0,0x09,0x8d
  }}};
#endif
#if ALG_SHA256 == YES
TPM2B_TYPE(SHA256, 32);
TPM2B_SHA256    c_SHA256_digest = {{32, {
  0x64,0xe8,0xe0,0xc3,0xa9,0xa4,0x51,0x49,0x10,0x55,0x8d,0x31,0x71,0xe5,0x2f,0x69,
  0x3a,0xdc,0xc7,0x11,0x32,0x44,0x61,0xbd,0x34,0x39,0x57,0xb0,0xa8,0x75,0x86,0x1b
  }}};
```



```
50  #endif
```

10.1.9 RsaTestData.h

RSA Test Vectors

```
 1  #define RSA_TEST_KEY_SIZE   256
 2  typedef struct
 3  {
 4      UINT16      size;
 5      BYTE        buffer[RSA_TEST_KEY_SIZE];
 6  } TPM2B_RSA_TEST_KEY;
 7  typedef TPM2B_RSA_TEST_KEY  TPM2B_RSA_TEST_VALUE;
 8  typedef struct
 9  {
10      UINT16       size;
11      BYTE        buffer[RSA_TEST_KEY_SIZE / 2];
12  } TPM2B_RSA_TEST_PRIME;
13  const TPM2B_RSA_TEST_KEY    c_rsaPublicModulus = {256, {
14    0x91,0x12,0xf5,0x07,0x9d,0x5f,0x6b,0x1c,0x90,0xf6,0xcc,0x87,0xde,0x3a,0x7a,0x15,
15    0xdc,0x54,0x07,0x6c,0x26,0x8f,0x25,0xef,0x7e,0x66,0xc0,0xe3,0x82,0x12,0x2f,0xab,
16    0x52,0x82,0x1e,0x85,0xbc,0x53,0xba,0x2b,0x01,0xad,0x01,0xc7,0x8d,0x46,0x4f,0x7d,
17    0xdd,0x7e,0xdc,0xb0,0xad,0xf6,0x0c,0xa1,0x62,0x92,0x97,0x8a,0x3e,0x6f,0x7e,0x3e,
18    0xf6,0x9a,0xcc,0xf9,0xa9,0x86,0x77,0xb6,0x85,0x43,0x42,0x04,0x13,0x65,0xe2,0xad,
19    0x36,0xc9,0xbf,0xc1,0x97,0x84,0x6f,0xee,0x7c,0xda,0x58,0xd2,0xae,0x07,0x00,0xaf,
20    0xc5,0x5f,0x4d,0x3a,0x98,0xb0,0xed,0x27,0x7c,0xc2,0xce,0x26,0x5d,0x87,0xe1,0xe3,
21    0xa9,0x69,0x88,0x4f,0x8c,0x08,0x31,0x18,0xae,0x93,0x16,0xe3,0x74,0xde,0xd3,0xf6,
22    0x16,0xaf,0xa3,0xac,0x37,0x91,0x8d,0x10,0xc6,0x6b,0x64,0x14,0x3a,0xd9,0xfc,0xe4,
23    0xa0,0xf2,0xd1,0x01,0x37,0x4f,0x4a,0xeb,0xe5,0xec,0x98,0xc5,0xd9,0x4b,0x30,0xd2,
24    0x80,0x2a,0x5a,0x18,0x5a,0x7d,0xd4,0x3d,0xb7,0x62,0x98,0xce,0x6d,0xa2,0x02,0x6e,
25    0x45,0xaa,0x95,0x73,0xe0,0xaa,0x75,0x57,0xb1,0x3d,0x1b,0x05,0x75,0x23,0x6b,0x20,
26    0x69,0x9e,0x14,0xb0,0x7f,0xac,0xae,0xd2,0xc7,0x48,0x3b,0xe4,0x56,0x11,0x34,0x1e,
27    0x05,0x1a,0x30,0x20,0xef,0x68,0x93,0x6b,0x9d,0x7e,0xdd,0xba,0x96,0x50,0xcc,0x1c,
28    0x81,0xb4,0x59,0xb9,0x74,0x36,0xd9,0x97,0xdc,0x8f,0x17,0x82,0x72,0xb3,0x59,0xf6,
29    0x23,0xfa,0x84,0xf7,0x6d,0xf2,0x05,0xff,0xf1,0xb9,0xcc,0xe9,0xa2,0x82,0x01,0xfb}};
30  const TPM2B_RSA_TEST_PRIME    c_rsaPrivatePrime = {RSA_TEST_KEY_SIZE / 2, {
31    0xb7,0xa0,0x90,0xc7,0x92,0x09,0xde,0x71,0x03,0x37,0x4a,0xb5,0x2f,0xda,0x61,0xb8,
32    0x09,0x1b,0xba,0x99,0x70,0x45,0xc1,0x0b,0x15,0x12,0x71,0x8a,0xb3,0x2a,0x4d,0x5a,
33    0x41,0x9b,0x73,0x89,0x80,0x0a,0x8f,0x18,0x4c,0x8b,0xa2,0x5b,0xda,0xbd,0x43,0xbe,
34    0xdc,0x76,0x4d,0x71,0x0f,0xb9,0xfc,0x7a,0x09,0xfe,0x4f,0xac,0x63,0xd9,0x2e,0x50,
35    0x3a,0xa1,0x37,0xc6,0xf2,0xa1,0x89,0x12,0xe7,0x72,0x64,0x2b,0xba,0xc1,0x1f,0xca,
36    0x9d,0xb7,0xaa,0x3a,0xa9,0xd3,0xa6,0x6f,0x73,0x02,0xbb,0x85,0x5d,0x9a,0xb9,0x5c,
37    0x08,0x83,0x22,0x20,0x49,0x91,0x5f,0x4b,0x86,0xbc,0x3f,0x76,0x43,0x08,0x97,0xbf,
38    0x82,0x55,0x36,0x2d,0x8b,0x6e,0x9e,0xfb,0xc1,0x67,0x6a,0x43,0xa2,0x46,0x81,0x71}};
39  const BYTE        c_RsaTestValue[RSA_TEST_KEY_SIZE] = {
40    0x2a,0x24,0x3a,0xbb,0x50,0x1d,0xd4,0x2a,0xf9,0x18,0x32,0x34,0xa2,0x0f,0xea,0x5c,
41    0x91,0x77,0xe9,0xe1,0x09,0x83,0xdc,0x5f,0x71,0x64,0x5b,0xeb,0x57,0x79,0xa0,0x41,
42    0xc9,0xe4,0x5a,0x0b,0xf4,0x9f,0xdb,0x84,0x04,0xa6,0x48,0x24,0xf6,0x3f,0x66,0x1f,
43    0xa8,0x04,0x5c,0xf0,0x7a,0x6b,0x4a,0x9c,0x7e,0x21,0xb6,0xda,0x6b,0x65,0x9c,0x3a,
44    0x68,0x50,0x13,0x1e,0xa4,0xb7,0xca,0xec,0xd3,0xcc,0xb2,0x9b,0x8c,0x87,0xa4,0x6a,
45    0xba,0xc2,0x06,0x3f,0x40,0x48,0x7b,0xa8,0xb8,0x2c,0x03,0x14,0x33,0xf3,0x1d,0xe9,
46    0xbd,0x6f,0x54,0x66,0xb4,0x69,0x5e,0xbc,0x80,0x7c,0xe9,0x6a,0x43,0x7f,0xb8,0x6a,
47    0xa0,0x5f,0x5d,0x7a,0x20,0xfd,0x7a,0x39,0xe1,0xea,0x0e,0x94,0x91,0x28,0x63,0x7a,
48    0xac,0xc9,0xa5,0x3a,0x6d,0x31,0x7b,0x7c,0x54,0x56,0x99,0x56,0xbb,0xb7,0xa1,0x2d,
49    0xd2,0x5c,0x91,0x5f,0x1c,0xd3,0x06,0x7f,0x34,0x53,0x2f,0x4c,0xd1,0x8b,0xd2,0x9e,
50    0xdc,0xc3,0x94,0x0a,0xe1,0x0f,0xa5,0x15,0x46,0x2a,0x8e,0x10,0xc2,0xfe,0xb7,0x5e,
51    0x2d,0x0d,0xd1,0x25,0xfc,0xe4,0xf7,0x02,0x19,0xfe,0xb6,0xe4,0x95,0x9c,0x17,0x4a,
52    0x9b,0xdb,0xab,0xc7,0x79,0xe3,0x5e,0x40,0xd0,0x56,0x6d,0x25,0x0a,0x72,0x65,0x80,
53    0x92,0x9a,0xa8,0x07,0x70,0x32,0x14,0xfb,0xfe,0x08,0xeb,0x13,0xb4,0x07,0x68,0xb4,
54    0x58,0x39,0xbe,0x8e,0x78,0x3a,0x59,0x3f,0x9c,0x4c,0xe9,0xa8,0x64,0x68,0xf7,0xb9,
55    0x6e,0x20,0xf5,0xcb,0xca,0x47,0xf2,0x17,0xaa,0x8b,0xbc,0x13,0x14,0x84,0xf6,0xab};
56  #define    OAEP_TEST_LABEL     "OAEP Test Value"
57  #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
58  const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
```

```
59      0x32,0x68,0x84,0x0b,0x9c,0xc9,0x25,0x26,0xd9,0xc0,0xd0,0xb1,0xde,0x60,0x55,0xae,
60      0x33,0xe5,0xcf,0x6c,0x85,0xbe,0x0d,0x71,0x11,0xe1,0x45,0x60,0xbb,0x42,0x3d,0xf3,
61      0xb1,0x18,0x84,0x7b,0xc6,0x5d,0xce,0x1d,0x5f,0x9a,0x97,0xcf,0xb1,0x97,0x9a,0x85,
62      0x7c,0xa7,0xa1,0x63,0x23,0xb6,0x74,0x0f,0x1a,0xee,0x29,0x51,0xeb,0x50,0x8f,0x3c,
63      0x8e,0x4e,0x31,0x38,0xdc,0x11,0xfc,0x9a,0x4e,0xaf,0x93,0xc9,0x7f,0x6e,0x35,0xf3,
64      0xc9,0xe4,0x89,0x14,0x53,0xe2,0xc2,0x1a,0xf7,0x6b,0x9b,0xf0,0x7a,0xa4,0x69,0x52,
65      0xe0,0x24,0x8f,0xea,0x31,0xa7,0x5c,0x43,0xb0,0x65,0xc9,0xfe,0xba,0xfe,0x80,0x9e,
66      0xa5,0xc0,0xf5,0x8d,0xce,0x41,0xf9,0x83,0x0d,0x8e,0x0f,0xef,0x3d,0x1f,0x6a,0xcc,
67      0x8a,0x3d,0x3b,0xdf,0x22,0x38,0xd7,0x34,0x58,0x7b,0x55,0xc9,0xf6,0xbc,0x7c,0x4c,
68      0x3f,0xd7,0xde,0x4e,0x30,0xa9,0x69,0xf3,0x5f,0x56,0x8f,0xc2,0xe7,0x75,0x79,0xb8,
69      0xa5,0xc8,0x0d,0xc0,0xcd,0xb6,0xc9,0x63,0xad,0x7c,0xe4,0x8f,0x39,0x60,0x4d,0x7d,
70      0xdb,0x34,0x49,0x2a,0x47,0xde,0xc0,0x42,0x4a,0x19,0x94,0x2e,0x50,0x21,0x03,0x47,
71      0xff,0x73,0xb3,0xb7,0x89,0xcc,0x7b,0x2c,0xeb,0x03,0xa7,0x9a,0x06,0xfd,0xed,0x19,
72      0xbb,0x82,0xa0,0x13,0xe9,0xfa,0xac,0x06,0x5f,0xc5,0xa9,0x2b,0xda,0x88,0x23,0xa2,
73      0x5d,0xc2,0x7f,0xda,0xc8,0x5a,0x94,0x31,0xc1,0x21,0xd7,0x1e,0x6b,0xd7,0x89,0xb1,
74      0x93,0x80,0xab,0xd1,0x37,0xf2,0x6f,0x50,0xcd,0x2a,0xea,0xb1,0xc4,0xcd,0xcb,0xb5}};
75    const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
76      0x29,0xa4,0x2f,0xbb,0x8a,0x14,0x05,0x1e,0x3c,0x72,0x76,0x77,0x38,0xe7,0x73,0xe3,
77      0x6e,0x24,0x4b,0x38,0xd2,0x1a,0xcf,0x23,0x58,0x78,0x36,0x82,0x23,0x6e,0x6b,0xef,
78      0x2c,0x3d,0xf2,0xe8,0xd6,0xc6,0x87,0x8e,0x78,0x9b,0x27,0x39,0xc0,0xd6,0xef,0x4d,
79      0x0b,0xfc,0x51,0x27,0x18,0xf3,0x51,0x5e,0x4d,0x96,0x3a,0xe2,0x15,0xe2,0x7e,0x42,
80      0xf4,0x16,0xd5,0xc6,0x52,0x5d,0x17,0x44,0x76,0x09,0x7a,0xcf,0xe3,0x30,0xe3,0x84,
81      0xf6,0x6f,0x3a,0x33,0xfb,0x32,0x0d,0x1d,0xe7,0x7c,0x80,0x82,0x4f,0xed,0xda,0x87,
82      0x11,0x9c,0xc3,0x7e,0x85,0xbd,0x18,0x58,0x08,0x2b,0x23,0x37,0xe7,0x9d,0xd0,0xd1,
83      0x79,0xe2,0x05,0xbd,0xf5,0x4f,0x0e,0x0f,0xdb,0x4a,0x74,0xeb,0x09,0x01,0xb3,0xca,
84      0xbd,0xa6,0x7b,0x09,0xb1,0x13,0x77,0x30,0x4d,0x87,0x41,0x06,0x57,0x2e,0x5f,0x36,
85      0x6e,0xfc,0x35,0x69,0xfe,0x0a,0x24,0x6c,0x98,0x8c,0xda,0x97,0xf4,0xfb,0xc7,0x83,
86      0x2d,0x3e,0x7d,0xc0,0x5c,0x34,0xfd,0x11,0x2a,0x12,0xa7,0xae,0x4a,0xde,0xc8,0x4e,
87      0xcf,0xf4,0x85,0x63,0x77,0xc6,0x33,0x34,0xe0,0x27,0xe4,0x9e,0x91,0x0b,0x4b,0x85,
88      0xf0,0xb0,0x79,0xaa,0x7c,0xc6,0xff,0x3b,0xbc,0x04,0x73,0xb8,0x95,0xd7,0x31,0x54,
89      0x3b,0x56,0xec,0x52,0x15,0xd7,0x3e,0x62,0xf5,0x82,0x99,0x3e,0x2a,0xc0,0x4b,0x2e,
90      0x06,0x57,0x6d,0x3f,0x3e,0x77,0x1f,0x2b,0x2d,0xc5,0xb9,0x3b,0x68,0x56,0x73,0x70,
91      0x32,0x6b,0x6b,0x65,0x25,0x76,0x45,0x6c,0x45,0xf1,0x6c,0x59,0xfc,0x94,0xa7,0x15}};
92    const TPM2B_RSA_TEST_VALUE    c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
93      0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x9a,0x07,0xb6,
94      0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
95      0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
96      0x93,0xb8,0x07,0x53,0xcc,0xdb,0x7f,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
97      0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,
98      0x9a,0x76,0x71,0x2a,0x35,0x2a,0xa1,0x08,0xbb,0x95,0xe3,0x64,0xdc,0xdb,0xc2,0x33,
99      0xa9,0x5f,0xbe,0x4c,0xc4,0xcc,0x28,0xc9,0x25,0xff,0xee,0x17,0x15,0x9a,0x50,0x90,
100     0x0e,0x15,0xb4,0xea,0x6a,0x09,0xe6,0xff,0xa4,0xee,0xc7,0x7e,0xce,0xa9,0x73,0xe4,
101     0xa0,0x56,0xbd,0x53,0x2a,0xe4,0xc0,0x2b,0xa8,0x9b,0x09,0x30,0x72,0x62,0x0f,0xf9,
102     0xf6,0xa1,0x52,0xd2,0x8a,0x37,0xee,0xa5,0xc8,0x47,0xe1,0x99,0x21,0x47,0xeb,0xdd,
103     0x37,0xaa,0xe4,0xbd,0x55,0x46,0x5a,0x5a,0x5d,0xfb,0x7b,0xfc,0xff,0xbf,0x26,0x71,
104     0xf6,0x1e,0xad,0xbc,0xbf,0x33,0xca,0xe1,0x92,0x8f,0x2a,0x89,0x6c,0x45,0x24,0xd1,
105     0xa6,0x52,0x56,0x24,0x5e,0x90,0x47,0xe5,0xcb,0x12,0xb0,0x32,0xf9,0xa6,0xbb,0xea,
106     0x37,0xa9,0xbd,0xef,0x23,0xef,0x63,0x07,0x6c,0xc4,0x4e,0x64,0x3c,0xc6,0x11,0x84,
107     0x7d,0x65,0xd6,0x5d,0x7a,0x17,0x58,0xa5,0xf7,0x74,0x3b,0x42,0xe3,0xd2,0xda,0x5f,
108     0x6f,0xe0,0x1e,0x4b,0xcf,0x46,0xe2,0xdf,0x3e,0x41,0x8e,0x0e,0xb0,0x3f,0x8b,0x65}};
109   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
110     0x01,0xfe,0xd5,0x83,0x0b,0x15,0xba,0x90,0x2c,0xdf,0xf7,0x26,0xb7,0x8f,0xb1,0xd7,
111     0x0b,0xfd,0x83,0xf9,0x95,0xd5,0xd7,0xb5,0xc5,0xc5,0x4a,0xde,0xd5,0xe6,0x20,0x78,
112     0xca,0x73,0x77,0x3d,0x61,0x36,0x48,0xae,0x3e,0x8f,0xee,0x43,0x29,0x96,0xdf,0x3f,
113     0x1c,0x97,0x5a,0xbe,0xe5,0xa2,0x7e,0x5b,0xd0,0xc0,0x29,0x39,0x83,0x81,0x77,0x24,
114     0x43,0xdb,0x3c,0x64,0x4d,0xf0,0x23,0xe4,0xae,0x0f,0x78,0x31,0x8c,0xda,0x0c,0xec,
115     0xf1,0xdf,0x09,0xf2,0x14,0x6a,0x4d,0xaf,0x36,0x81,0x6e,0xbd,0xbe,0x36,0x79,0x88,
116     0x98,0xb6,0x6f,0x5a,0xad,0xcf,0x7c,0xee,0xe0,0xdd,0x00,0xbe,0x59,0x97,0x88,0x00,
117     0x34,0xc0,0x8b,0x48,0x42,0x05,0x04,0x5a,0xb7,0x85,0x38,0xa0,0x35,0xd7,0x3b,0x51,
118     0xb8,0x7b,0x81,0x83,0xee,0xff,0x76,0x6f,0x50,0x39,0x4d,0xab,0x89,0x63,0x07,0x6d,
119     0xf5,0xe5,0x01,0x10,0x56,0xfe,0x93,0x06,0x8f,0xd3,0xc9,0x41,0xab,0xc9,0xdf,0x6e,
120     0x59,0xa8,0xc3,0x1d,0xbf,0x96,0x4a,0x59,0x80,0x3c,0x90,0x3a,0x59,0x56,0x4c,0x6d,
121     0x44,0x6d,0xeb,0xdc,0x73,0xcd,0xc1,0xec,0xb8,0x41,0xbf,0x89,0x8c,0x03,0x69,0x4c,
122     0xaf,0x3f,0xc1,0xc5,0xc7,0xe7,0x7d,0xa7,0x83,0x39,0x70,0xa2,0x6b,0x83,0xbc,0xbe,
123     0xf5,0xbf,0x1c,0xee,0x6e,0xa3,0x22,0x1e,0x25,0x2f,0x16,0x68,0x69,0x5a,0x1d,0xfa,
124     0x2c,0x3a,0x0f,0x67,0xe1,0x77,0x12,0xe8,0x3d,0xba,0xaa,0xef,0x96,0x9c,0x1f,0x64,
```

```
125       0x32,0xf4,0xa7,0xb3,0x3f,0x7d,0x61,0xbb,0x9a,0x27,0xad,0xfb,0x2f,0x33,0xc4,0x70}};
126     const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
127       0x67,0x4e,0xdd,0xc2,0xd2,0x6d,0xe0,0x03,0xc4,0xc2,0x41,0xd3,0xd4,0x61,0x30,0xd0,
128       0xe1,0x68,0x31,0x4a,0xda,0xd9,0xc2,0x5d,0xaa,0xa2,0x7b,0xfb,0x44,0x02,0xf5,0xd6,
129       0xd8,0x2e,0xcd,0x13,0x36,0xc9,0x4b,0xdb,0x1a,0x4b,0x66,0x1b,0x4f,0x9c,0xb7,0x17,
130       0xac,0x53,0x37,0x4f,0x21,0xbd,0x0c,0x66,0xac,0x06,0x65,0x52,0x9f,0x04,0xf6,0xa5,
131       0x22,0x5b,0xf7,0xe6,0x0d,0x3c,0x9f,0x41,0x19,0x09,0x88,0x7c,0x41,0x4c,0x2f,0x9c,
132       0x8b,0x3c,0xdd,0x7c,0x28,0x78,0x24,0xd2,0x09,0xa6,0x5b,0xf7,0x3c,0x88,0x7e,0x73,
133       0x5a,0x2d,0x36,0x02,0x4f,0x65,0xb0,0xcb,0xc8,0xdc,0xac,0xa2,0xda,0x8b,0x84,0x91,
134       0x71,0xe4,0x30,0x8b,0xb6,0x12,0xf2,0xf0,0xd0,0xa0,0x38,0xcf,0x75,0xb7,0x20,0xcb,
135       0x35,0x51,0x52,0x6b,0xc4,0xf4,0x21,0x95,0xc2,0xf7,0x9a,0x13,0xc1,0x1a,0x7b,0x8f,
136       0x77,0xda,0x19,0x48,0xbb,0x6d,0x14,0x5d,0xba,0x65,0xb4,0x9e,0x43,0x42,0x58,0x98,
137       0x0b,0x91,0x46,0xd8,0x4c,0xf3,0x4c,0xaf,0x2e,0x02,0xa6,0xb2,0x49,0x12,0x62,0x43,
138       0x4e,0xa8,0xac,0xbf,0xfd,0xfa,0x37,0x24,0xea,0x69,0x1c,0xf5,0xae,0xfa,0x08,0x82,
139       0x30,0xc3,0xc0,0xf8,0x9a,0x89,0x33,0xe1,0x40,0x6d,0x18,0x5c,0x7b,0x90,0x48,0xbf,
140       0x37,0xdb,0xea,0xfb,0x0e,0xd4,0x2e,0x11,0xfa,0xa9,0x86,0xff,0x00,0x0b,0x7b,0xca,
141       0x09,0x64,0x6a,0x8f,0x0c,0x0e,0x09,0x14,0x36,0x4a,0x74,0x31,0x18,0x5b,0x18,0xeb,
142       0xea,0x83,0xc3,0x66,0x68,0xa6,0x7d,0x43,0x06,0x0f,0x99,0x60,0xce,0x65,0x08,0xf6}};
143   #endif // SHA1
144   #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
145   const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
146     0x0f,0x3c,0x42,0x4d,0x8c,0x91,0x96,0x05,0x3c,0xfd,0x59,0x3b,0x7f,0x29,0xbc,0x03,
147     0x67,0xc1,0xff,0x74,0xe7,0x09,0xf4,0x13,0x45,0xbe,0x13,0x1d,0xc9,0x86,0x94,0xfe,
148     0xed,0xa6,0xe8,0x3a,0xcb,0x89,0x4d,0xec,0x86,0x63,0x4c,0xdb,0xf1,0x95,0xee,0xc1,
149     0x46,0xc5,0x3b,0xd8,0xf8,0xa2,0x41,0x6a,0x60,0x8b,0x9e,0x5e,0x7f,0x20,0x16,0xe3,
150     0x69,0xb6,0x2d,0x92,0xfc,0x60,0xa2,0x74,0x88,0xd5,0xc7,0xa6,0xd1,0xff,0xe3,0x45,
151     0x02,0x51,0x39,0xd9,0xf3,0x56,0x0b,0x91,0x80,0xe0,0x6c,0xa8,0xc3,0x78,0xef,0x34,
152     0x22,0x8c,0xf5,0xfb,0x47,0x98,0x5d,0x57,0x8e,0x3a,0xb9,0xff,0x92,0x04,0xc7,0xc2,
153     0x6e,0xfa,0x14,0xc1,0xb9,0x68,0x15,0x5c,0x12,0xe8,0xa8,0xbe,0xea,0xe8,0x8d,0x9b,
154     0x48,0x28,0x35,0xdb,0x4b,0x52,0xc1,0x2d,0x85,0x47,0x83,0xd0,0xe9,0xae,0x90,0x6e,
155     0x65,0xd4,0x34,0x7f,0x81,0xce,0x69,0xf0,0x96,0x62,0xf7,0xec,0x41,0xd5,0xc2,0xe3,
156     0x4b,0xba,0x9c,0x8a,0x02,0xce,0xf0,0x5d,0x14,0xf7,0x09,0x42,0x8e,0x4a,0x27,0xfe,
157     0x3e,0x66,0x42,0x99,0x03,0xe1,0x69,0xbd,0xdb,0x7f,0x9b,0x70,0xeb,0x4e,0x9c,0xac,
158     0x45,0x67,0x91,0x9f,0x75,0x10,0xc6,0xfc,0x14,0xe1,0x28,0xc1,0x0e,0xe0,0x7e,0xc0,
159     0x5c,0x1d,0xee,0xe8,0xff,0x45,0x79,0x51,0x86,0x08,0xe6,0x39,0xac,0xb5,0xfd,0xb8,
160     0xf1,0xdd,0x2e,0xf4,0xb2,0x1a,0x69,0x0d,0xd9,0x98,0x8e,0xdb,0x85,0x61,0x70,0x20,
161     0x82,0x91,0x26,0x87,0x80,0xc4,0x6a,0xd8,0x3b,0x91,0x4d,0xd3,0x33,0x84,0xad,0xb7}};
162   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
163     0x44,0xd5,0x9f,0xbc,0x48,0x03,0x3d,0x9f,0x22,0x91,0x2a,0xab,0x3c,0x31,0x71,0xab,
164     0x86,0x3f,0x0f,0x6f,0x59,0x5b,0x93,0x27,0xbc,0xbc,0xcd,0x29,0x38,0x43,0x2a,0x3b,
165     0x3b,0xd2,0xb3,0x45,0x40,0xba,0x15,0xb4,0x45,0xe3,0x56,0xab,0xff,0xb3,0x20,0x26,
166     0x39,0xcc,0x48,0xc5,0x5d,0x41,0x0d,0x2f,0x57,0x7f,0x9d,0x16,0x2e,0x26,0x57,0xc7,
167     0x6b,0xf3,0x36,0x54,0xbd,0xb6,0x1d,0x46,0x4e,0x13,0x50,0xd7,0x61,0x9d,0x8d,0x7b,
168     0xeb,0x21,0x9f,0x79,0xf3,0xfd,0xe0,0x1b,0xa8,0xed,0x6d,0x29,0x33,0x0d,0x65,0x94,
169     0x24,0x1e,0x62,0x88,0x6b,0x2b,0x4e,0x39,0xf5,0x80,0x39,0xca,0x76,0x95,0xbc,0x7c,
170     0x27,0x1d,0xdd,0x3a,0x11,0xf1,0x3e,0x54,0x03,0xb7,0x43,0x91,0x99,0x33,0xfe,0x9d,
171     0x14,0x2c,0x87,0x9a,0x95,0x18,0x1f,0x02,0x04,0x6a,0xe2,0xb7,0x81,0x14,0x13,0x45,
172     0x16,0xfb,0xe4,0xb7,0x8f,0xab,0x2b,0xd7,0x60,0x34,0x8a,0x55,0xbc,0x01,0x8c,0x49,
173     0x02,0x29,0xf1,0x9c,0x94,0x98,0x44,0xd0,0x94,0xcb,0xd4,0x85,0x4c,0x3b,0x77,0x72,
174     0x99,0xd5,0x4b,0xc6,0x3b,0xe4,0xd2,0xc8,0xe9,0x6a,0x23,0x18,0x3b,0x3b,0x5e,0x32,
175     0xec,0x70,0x84,0x5d,0xbb,0x6a,0x8f,0x0c,0x5f,0x55,0xa5,0x30,0x34,0x48,0xbb,0xc2,
176     0xdf,0x12,0xb9,0x81,0xad,0x36,0x3f,0xf0,0x24,0x16,0x48,0x04,0x4a,0x7f,0xfd,0x9f,
177     0x4c,0xea,0xfe,0x1d,0x83,0xd0,0x81,0xad,0x25,0x6c,0x5f,0x45,0x36,0x91,0xf0,0xd5,
178     0x8b,0x53,0x0a,0xdf,0xec,0x9f,0x04,0x58,0xc4,0x35,0xa0,0x78,0x1f,0x68,0xe0,0x22}};
179   const TPM2B_RSA_TEST_VALUE    c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
180     0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x9a,0x07,0xb6,
181     0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
182     0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
183     0x93,0xb8,0x07,0x53,0xcc,0xdb,0x7f,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
184     0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,
185     0x9a,0x76,0x71,0x2a,0x35,0x2a,0xa1,0x08,0xbb,0x95,0xe3,0x64,0xdc,0xdb,0xc2,0x33,
186     0xa9,0x5f,0xbe,0x4c,0xc4,0xcc,0x28,0xc9,0x25,0xff,0xee,0x17,0x15,0x9a,0x50,0x90,
187     0x0e,0x15,0xb4,0xea,0x6a,0x09,0xe6,0xff,0xa4,0xee,0xc7,0x7e,0xce,0xa9,0x73,0xe4,
188     0xa0,0x56,0xbd,0x53,0x2a,0xe4,0xc0,0x2b,0xa8,0x9b,0x09,0x30,0x72,0x62,0x0f,0xf9,
189     0xf6,0xa1,0x52,0xd2,0x8a,0x37,0xee,0xa5,0xc8,0x47,0xe1,0x99,0x21,0x47,0xeb,0xdd,
190     0x37,0xaa,0xe4,0xbd,0x55,0x46,0x5a,0x5a,0x5d,0xfb,0x7b,0xfc,0xff,0xbf,0x26,0x71,
```

```
191     0xf6,0x1e,0xad,0xbc,0xbf,0x33,0xca,0xe1,0x92,0x8f,0x2a,0x89,0x6c,0x45,0x24,0xd1,
192     0xa6,0x52,0x56,0x24,0x5e,0x90,0x47,0xe5,0xcb,0x12,0xb0,0x32,0xf9,0xa6,0xbb,0xea,
193     0x37,0xa9,0xbd,0xef,0x23,0xef,0x63,0x07,0x6c,0xc4,0x4e,0x64,0x3c,0xc6,0x11,0x84,
194     0x7d,0x65,0xd6,0x5d,0x7a,0x17,0x58,0xa5,0xf7,0x74,0x3b,0x42,0xe3,0xd2,0xda,0x5f,
195     0x6f,0xe0,0x1e,0x4b,0xcf,0x46,0xe2,0xdf,0x3e,0x41,0x8e,0x0e,0xb0,0x3f,0x8b,0x65}};
196   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
197     0x3f,0x3a,0x82,0x6d,0x42,0xe3,0x8b,0x4f,0x45,0x9c,0xda,0x6c,0xbe,0xbe,0xcd,0x00,
198     0x98,0xfb,0xbe,0x59,0x30,0xc6,0x3c,0xaa,0xb3,0x06,0x27,0xb5,0xda,0xfa,0xb2,0xc3,
199     0x43,0xb7,0xbd,0xe9,0xd3,0x23,0xed,0x80,0xce,0x74,0xb3,0xb8,0x77,0x8d,0xe6,0x8d,
200     0x3c,0xe5,0xf5,0xd7,0x80,0xcf,0x38,0x55,0x76,0xd7,0x87,0xa8,0xd6,0x3a,0xcf,0xfd,
201     0xd8,0x91,0x65,0xab,0x43,0x66,0x50,0xb7,0x9a,0x13,0x6b,0x45,0x80,0x76,0x86,0x22,
202     0x27,0x72,0xf7,0xbb,0x65,0x22,0x5c,0x55,0x60,0xd8,0x84,0x9f,0xf2,0x61,0x52,0xac,
203     0xf2,0x4f,0x5b,0x7b,0x21,0xe1,0xf5,0x4b,0x8f,0x01,0xf2,0x4b,0xcf,0xd3,0xfb,0x74,
204     0x5e,0x6e,0x96,0xb4,0xa8,0x0f,0x01,0x9b,0x26,0x54,0x0a,0x70,0x55,0x26,0xb7,0x0b,
205     0xe8,0x01,0x68,0x66,0x0d,0x6f,0xb5,0xfc,0x66,0xbd,0x9e,0x44,0xed,0x6a,0x1e,0x3c,
206     0x3b,0x61,0x5d,0xe8,0xdb,0x99,0x5b,0x67,0xbf,0x94,0xfb,0xe6,0x8c,0x4b,0x07,0xcb,
207     0x43,0x3a,0x0d,0xb1,0x1b,0x10,0x66,0x81,0xe2,0x0d,0xe7,0xd1,0xca,0x85,0xa7,0x50,
208     0x82,0x2d,0xbf,0xed,0xcf,0x43,0x6d,0xdb,0x2c,0x7b,0x73,0x20,0xfe,0x73,0x3f,0x19,
209     0xc6,0xdb,0x69,0xb8,0xc3,0xd3,0xf4,0xe5,0x64,0xf8,0x36,0x8e,0xd5,0xd8,0x09,0x2a,
210     0x5f,0x26,0x70,0xa1,0xd9,0x5b,0x14,0xf8,0x22,0xe9,0x9d,0x22,0x51,0xf4,0x52,0xc1,
211     0x6f,0x53,0xf5,0xca,0x0d,0xda,0x39,0x8c,0x29,0x42,0xe8,0x58,0x89,0xbb,0xd1,0x2e,
212     0xc5,0xdb,0x86,0x8d,0xaf,0xec,0x58,0x36,0x8d,0x8d,0x57,0x23,0xd5,0xdd,0xb9,0x24}};
213   const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
214     0x39,0x10,0x58,0x7d,0x6d,0xa8,0xd5,0x90,0x07,0xd6,0x2b,0x13,0xe9,0xd8,0x93,0x7e,
215     0xf3,0x5d,0x71,0xe0,0xf0,0x33,0x3a,0x4a,0x22,0xf3,0xe6,0x95,0xd3,0x8e,0x8c,0x41,
216     0xe7,0xb3,0x13,0xde,0x4a,0x45,0xd3,0xd1,0xfb,0xb1,0x3f,0x9b,0x39,0xa5,0x50,0x58,
217     0xef,0xb6,0x3a,0x43,0xdd,0x54,0xab,0xda,0x9d,0x32,0x49,0xe4,0x57,0x96,0xe5,0x1b,
218     0x1d,0x8f,0x33,0x8e,0x07,0x67,0x56,0x14,0xc1,0x18,0x78,0xa2,0x52,0xe6,0x2e,0x07,
219     0x81,0xbe,0xd8,0xca,0x76,0x63,0x68,0xc5,0x47,0xa2,0x92,0x5e,0x4c,0xfd,0x14,0xc7,
220     0x46,0x14,0xbe,0xc7,0x85,0xef,0xe6,0xb8,0x46,0xcb,0x3a,0x67,0x66,0x89,0xc6,0xee,
221     0x9d,0x64,0xf5,0x0d,0x09,0x80,0x9a,0x6f,0x0e,0xeb,0xe4,0xb9,0xe9,0xab,0x90,0x4f,
222     0xe7,0x5a,0xc8,0xca,0xf6,0x16,0x0a,0x82,0xbd,0xb7,0x76,0x59,0x08,0x2d,0xd9,0x40,
223     0x5d,0xaa,0xa5,0xef,0xfb,0xe3,0x81,0x2c,0x2c,0x5c,0xa8,0x16,0xbd,0x63,0x20,0xc2,
224     0x4d,0x3b,0x51,0xaa,0x62,0x1f,0x06,0xe5,0xbb,0x78,0x44,0x04,0x0c,0x5c,0xe1,0x1b,
225     0x6b,0x9d,0x21,0x10,0xaf,0x48,0x48,0x98,0x97,0x77,0xc2,0x73,0xb4,0x98,0x64,0xcc,
226     0x94,0x2c,0x29,0x28,0x45,0x36,0xd1,0xc5,0xd0,0x2f,0x97,0x27,0x92,0x65,0x22,0xbb,
227     0x63,0x79,0xea,0xf5,0xff,0x77,0x0f,0x4b,0x56,0x8a,0x9f,0xad,0x1a,0x97,0x67,0x39,
228     0x69,0xb8,0x4c,0x6c,0xc2,0x56,0xc5,0x7a,0xa8,0x14,0x5a,0x24,0x7a,0xa4,0x6e,0x55,
229     0xb2,0x86,0x1d,0xf4,0x62,0x5a,0x2d,0x87,0x6d,0xde,0x99,0x78,0x2d,0xef,0xd7,0xdc}};
230   #endif // SHA384
231   #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
232   const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
233     0x33,0x20,0x6e,0x21,0xc3,0xf6,0xcd,0xf8,0xd7,0x5d,0x9f,0xe9,0x05,0x14,0x8c,0x7c,
234     0xbb,0x69,0x24,0x9e,0x52,0x8f,0xaf,0x84,0x73,0x21,0x2c,0x85,0xa5,0x30,0x4d,0xb6,
235     0xb8,0xfa,0x15,0x9b,0xc7,0x8f,0xc9,0x7a,0x72,0x4b,0x85,0xa4,0x1c,0xc5,0xd8,0xe4,
236     0x92,0xb3,0xec,0xd9,0xa8,0xca,0x5e,0x74,0x73,0x89,0x7f,0xb4,0xac,0x7e,0x68,0x12,
237     0xb2,0x53,0x27,0x4b,0xbf,0xd0,0x71,0x69,0x46,0x9f,0xef,0xf4,0x70,0x60,0xf8,0xd7,
238     0xae,0xc7,0x5a,0x27,0x38,0x25,0x2d,0x25,0xab,0x96,0x56,0x66,0x3a,0x23,0x40,0xa8,
239     0xdb,0xbc,0x86,0xe8,0xf3,0xd2,0x58,0x0b,0x44,0xfc,0x94,0x1e,0xb7,0x5d,0xb4,0x57,
240     0xb5,0xf3,0x56,0xee,0x9b,0xcf,0x97,0x91,0x29,0x36,0xe3,0x06,0x13,0xa2,0xea,0xd6,
241     0xd6,0x0b,0x86,0x0b,0x1a,0x27,0xe6,0x22,0xc4,0x7b,0xff,0xde,0x0f,0xbf,0x79,0xc8,
242     0x1b,0xed,0xf1,0x27,0x62,0xb5,0x8b,0xf9,0xd9,0x76,0x90,0xf6,0xcc,0x83,0x0f,0xce,
243     0xce,0x2e,0x63,0x7a,0x9b,0xf4,0x48,0x5b,0xd7,0x81,0x2c,0x3a,0xdb,0x59,0x0d,0x4d,
244     0x9e,0x46,0xe9,0x9e,0x92,0x22,0x27,0x1c,0xb0,0x67,0x8a,0xe6,0x8a,0x16,0x8a,0xdf,
245     0x95,0x76,0x24,0x82,0xad,0xf1,0xbc,0x97,0xbf,0xd3,0x5e,0x6e,0x14,0x0c,0x5b,0x25,
246     0xfe,0x58,0xfa,0x64,0xe5,0x14,0x46,0xb7,0x58,0xc6,0x3f,0x7f,0x42,0xd2,0x8e,0x45,
247     0x13,0x41,0x85,0x12,0x2e,0x96,0x19,0xd0,0x5e,0x7d,0x34,0x06,0x32,0x2b,0xc8,0xd9,
248     0x0d,0x6c,0x06,0x36,0xa0,0xff,0x47,0x57,0x2c,0x25,0xbc,0x8a,0xa5,0xe2,0xc7,0xe3}};
249   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
250     0x39,0xfc,0x10,0x5d,0xf4,0x45,0x3d,0x94,0x53,0x06,0x89,0x24,0xe7,0xe8,0xfd,0x03,
251     0xac,0xfd,0xbd,0xb2,0x28,0xd3,0x4a,0x52,0xc5,0xd4,0xdb,0x17,0xd4,0x24,0x05,0xc4,
252     0xeb,0x6a,0xce,0x1d,0xbb,0x37,0xcb,0x09,0xd8,0x6c,0x83,0x19,0x93,0xd4,0xe2,0x88,
253     0x88,0x9b,0xaf,0x92,0x16,0xc4,0x15,0xbd,0x49,0x13,0x22,0xb7,0x84,0xcf,0x23,0xf2,
254     0x6f,0x0c,0x3e,0x8f,0xde,0x04,0x09,0x31,0x2d,0x99,0xdf,0xe6,0x74,0x70,0x30,0xde,
255     0x8c,0xad,0x32,0x86,0xe2,0x7c,0x12,0x90,0x21,0xf3,0x86,0xb7,0xe2,0x64,0xca,0x98,
256     0xcc,0x64,0x4b,0xef,0x57,0x4f,0x5a,0x16,0x6e,0xd7,0x2f,0x5b,0xf6,0x07,0xad,0x33,
```

```
257     0xb4,0x8f,0x3b,0x3a,0x8b,0xd9,0x06,0x2b,0xed,0x3c,0x3c,0x76,0xf6,0x21,0x31,0xe3,
258     0xfb,0x2c,0x45,0x61,0x42,0xba,0xe0,0xc3,0x72,0x63,0xd0,0x6b,0x8f,0x36,0x26,0xfb,
259     0x9e,0x89,0x0e,0x44,0x9a,0xc1,0x84,0x5e,0x84,0x8d,0xb6,0xea,0xf1,0x0d,0x66,0xc7,
260     0xdb,0x44,0xbd,0x19,0x7c,0x05,0xbe,0xc4,0xab,0x88,0x32,0xbe,0xc7,0x63,0x31,0xe6,
261     0x38,0xd4,0xe5,0xb8,0x4b,0xf5,0x0e,0x55,0x9a,0x3a,0xe6,0x0a,0xec,0xee,0xe2,0xa8,
262     0x88,0x04,0xf2,0xb8,0xaa,0x5a,0xd8,0x97,0x5d,0xa0,0xa8,0x42,0xfb,0xd9,0xde,0x80,
263     0xae,0x4c,0xb3,0xa1,0x90,0x47,0x57,0x03,0x10,0x78,0xa6,0x8f,0x11,0xba,0x4b,0xce,
264     0x2d,0x56,0xa4,0xe1,0xbd,0xf8,0xa0,0xa4,0xd5,0x48,0x3c,0x63,0x20,0x00,0x38,0xa0,
265     0xd1,0xe6,0x12,0xe9,0x1d,0xd8,0x49,0xe3,0xd5,0x24,0xb5,0xc5,0x3a,0x1f,0xb0,0xd4}};
266   const TPM2B_RSA_TEST_VALUE    c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
267     0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x9a,0x07,0xb6,
268     0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
269     0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
270     0x93,0xb8,0x07,0x53,0xcc,0xdb,0x7f,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
271     0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,
272     0x9a,0x76,0x71,0x2a,0x35,0x2a,0xa1,0x08,0xbb,0x95,0xe3,0x64,0xdc,0xdb,0xc2,0x33,
273     0xa9,0x5f,0xbe,0x4c,0xc4,0xcc,0x28,0xc9,0x25,0xff,0xee,0x17,0x15,0x9a,0x50,0x90,
274     0x0e,0x15,0xb4,0xea,0x6a,0x09,0xe6,0xff,0xa4,0xee,0xc7,0x7e,0xce,0xa9,0x73,0xe4,
275     0xa0,0x56,0xbd,0x53,0x2a,0xe4,0xc0,0x2b,0xa8,0x9b,0x09,0x30,0x72,0x62,0x0f,0xf9,
276     0xf6,0xa1,0x52,0xd2,0x8a,0x37,0xee,0xa5,0xc8,0x47,0xe1,0x99,0x21,0x47,0xeb,0xdd,
277     0x37,0xaa,0xe4,0xbd,0x55,0x46,0x5a,0x5a,0x5d,0xfb,0x7b,0xfc,0xff,0xbf,0x26,0x71,
278     0xf6,0x1e,0xad,0xbc,0xbf,0x33,0xca,0xe1,0x92,0x8f,0x2a,0x89,0x6c,0x45,0x24,0xd1,
279     0xa6,0x52,0x56,0x24,0x5e,0x90,0x47,0xe5,0xcb,0x12,0xb0,0x32,0xf9,0xa6,0xbb,0xea,
280     0x37,0xa9,0xbd,0xef,0x23,0xef,0x63,0x07,0x6c,0xc4,0x4e,0x64,0x3c,0xc6,0x11,0x84,
281     0x7d,0x65,0xd6,0x5d,0x7a,0x17,0x58,0xa5,0xf7,0x74,0x3b,0x42,0xe3,0xd2,0xda,0x5f,
282     0x6f,0xe0,0x1e,0x4b,0xcf,0x46,0xe2,0xdf,0x3e,0x41,0x8e,0x0e,0xb0,0x3f,0x8b,0x65}};
283   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
284     0x74,0x89,0x29,0x3e,0x1b,0xac,0xc6,0x85,0xca,0xf0,0x63,0x43,0x30,0x7d,0x1c,0x9b,
285     0x2f,0xbd,0x4d,0x69,0x39,0x5e,0x85,0xe2,0xef,0x86,0x0a,0xc6,0x6b,0xa6,0x08,0x19,
286     0x6c,0x56,0x38,0x24,0x55,0x92,0x84,0x9b,0x1b,0x8b,0x04,0xcf,0x24,0x14,0x24,0x13,
287     0x0e,0x8b,0x82,0x6f,0x96,0xc8,0x9a,0x68,0xfc,0x4c,0x02,0xf0,0xdc,0xcd,0x36,0x25,
288     0x31,0xd5,0x82,0xcf,0xc9,0x69,0x72,0xf6,0x1d,0xab,0x68,0x20,0x2e,0x2d,0x19,0x49,
289     0xf0,0x2e,0xad,0xd2,0xda,0xaf,0xff,0xb6,0x92,0x83,0x5b,0x8a,0x06,0x2d,0x0c,0x32,
290     0x11,0x32,0x3b,0x77,0x17,0xf6,0x50,0xfb,0xf8,0x57,0xc9,0xc7,0x9b,0x9e,0xc6,0xd1,
291     0xa9,0x55,0xf0,0x22,0x35,0xda,0xca,0x3c,0x8e,0xc6,0x9a,0xd8,0x25,0xc8,0x5e,0x93,
292     0x0d,0xaa,0xa7,0x06,0xaf,0x11,0x29,0x99,0xe7,0x7c,0xee,0x49,0x82,0x30,0xba,0x2c,
293     0xe2,0x40,0x8f,0x0a,0xa6,0x7b,0x24,0x75,0xc5,0xcd,0x03,0x12,0xf4,0xb2,0x4b,0x3a,
294     0xd1,0x91,0x3c,0x20,0x0e,0x58,0x2b,0x31,0xf8,0x8b,0xee,0xbc,0x1f,0x95,0x35,0x58,
295     0x6a,0x73,0xee,0x99,0xb0,0x01,0x42,0x4f,0x66,0xc0,0x66,0xbb,0x35,0x86,0xeb,0xd9,
296     0x7b,0x55,0x77,0x2d,0x54,0x78,0x19,0x49,0xe8,0xcc,0xfd,0xb1,0xcb,0x49,0xc9,0xea,
297     0x20,0xab,0xed,0xb5,0xed,0xfe,0xb2,0xb5,0xa8,0xcf,0x05,0x06,0xd5,0x7d,0x2b,0xbb,
298     0x0b,0x65,0x6b,0x2b,0x6d,0x55,0x95,0x85,0x44,0x8b,0x12,0x05,0xf3,0x4b,0xd4,0x8e,
299     0x3d,0x68,0x2d,0x29,0x9c,0x05,0x79,0xd6,0xfc,0x72,0x90,0x6a,0xab,0x46,0x38,0x81}};
300   const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
301     0x8a,0xb1,0x0a,0xb5,0xe4,0x02,0xf7,0xdd,0x45,0x2a,0xcc,0x2b,0x6b,0x8c,0x0e,0x9a,
302     0x92,0x4f,0x9b,0xc5,0xe4,0x8b,0x82,0xb9,0xb0,0xd9,0x87,0x8c,0xcb,0xf0,0xb0,0x59,
303     0xa5,0x92,0x21,0xa0,0xa7,0x61,0x5c,0xed,0xa8,0x6e,0x22,0x29,0x46,0xc7,0x86,0x37,
304     0x4b,0x1b,0x1e,0x94,0x93,0xc8,0x4c,0x17,0x7a,0xae,0x59,0x91,0xf8,0x83,0x84,0xc4,
305     0x8c,0x38,0xc2,0x35,0x0e,0x7e,0x50,0x67,0x76,0xe7,0xd3,0xec,0x6f,0x0d,0xa0,0x5c,
306     0x2f,0x0a,0x80,0x28,0xd3,0xc5,0x7d,0x2d,0x1a,0x0b,0x96,0xd6,0xe5,0x98,0x05,0x8c,
307     0x4d,0xa0,0x1f,0x8c,0xb6,0xfb,0xb1,0xcf,0xe9,0xcb,0x38,0x27,0x60,0x64,0x17,0xca,
308     0xf4,0x8b,0x61,0xb7,0x1d,0xb6,0x20,0x9d,0x40,0x2a,0x1c,0xfd,0x55,0x40,0x4b,0x95,
309     0x39,0x52,0x18,0x3b,0xab,0x44,0xe8,0x83,0x4b,0x7c,0x47,0xfb,0xed,0x06,0x9c,0xcd,
310     0x4f,0xba,0x81,0xd6,0xb7,0x31,0xcf,0x5c,0x23,0xf8,0x25,0xab,0x95,0x77,0x0a,0x8f,
311     0x46,0xef,0xfb,0x59,0xb8,0x04,0xd7,0x1e,0xf5,0xaf,0x6a,0x1a,0x26,0x9b,0xae,0xf4,
312     0xf5,0x7f,0x84,0x6f,0x3c,0xed,0xf8,0x24,0x0b,0x43,0xd1,0xba,0x74,0x89,0x4e,0x39,
313     0xfe,0xab,0xa5,0x16,0xa5,0x28,0xee,0x96,0x84,0x3e,0x16,0x6d,0x5f,0x4e,0x0b,0x7d,
314     0x94,0x16,0x1b,0x8c,0xf9,0xaa,0x9b,0xc0,0x49,0x02,0x4c,0x3e,0x62,0xff,0xfe,0xa2,
315     0x20,0x33,0x5e,0xa6,0xdd,0xda,0x15,0x2d,0xb7,0xcd,0xda,0xff,0xb1,0x0b,0x45,0x7b,
316     0xd3,0xa0,0x42,0x29,0xab,0xa9,0x73,0xe9,0xa4,0xd9,0x8d,0xac,0xa1,0x88,0x2c,0x2d}};
317   #endif // SHA256
```

10.1.10 SymmetricTestData.h

This is a vector for testing either encrypt or decrypt. The premise for decrypt is that the IV for decryption is the same as the IV for encryption. However, the *ivOut* value may be different for encryption and decryption. We will encrypt at least two blocks. This means that the chaining value will be used for each of the schemes (if any) and that implicitly checks that the chaining value is handled properly.

```
#if AES_128
const BYTE  key_AES128 [] = {
                0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
                0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
const BYTE  dataIn_AES128 [] = {
                0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
                0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
                0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
const BYTE  dataOut_AES128_ECB [] = {
                0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
                0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97,
                0xf5, 0xd3, 0xd5, 0x85, 0x03, 0xb9, 0x69, 0x9d,
                0xe7, 0x85, 0x89, 0x5a, 0x96, 0xfd, 0xba, 0xaf};
const BYTE  dataOut_AES128_CBC [] = {
                0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46,
                0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
                0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee,
                0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2};
const BYTE  dataOut_AES128_CFB [] = {
                0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
                0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
                0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
                0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b};
const BYTE  dataOut_AES128_OFB [] = {
                0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
                0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
                0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
                0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25};
const BYTE  dataOut_AES128_CTR [] = {
                0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
                0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
                0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
                0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff};
#endif
#if AES_192
const BYTE  key_AES192 [] = {
                0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
                0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
                0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b};
const BYTE  dataIn_AES192 [] = {
                0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
                0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
                0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
const BYTE  dataOut_AES192_ECB [] = {
                0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f,
                0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc,
                0x97, 0x41, 0x04, 0x84, 0x6d, 0x0a, 0xd3, 0xad,
                0x77, 0x34, 0xec, 0xb3, 0xec, 0xee, 0x4e, 0xef};
const BYTE  dataOut_AES192_CBC [] = {
                0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d,
                0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
                0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4,
                0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a};
const BYTE  dataOut_AES192_CFB [] = {
                0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
                0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
```



```
59                  0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
60                  0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a};
61      const BYTE  dataOut_AES192_OFB [] = {
62                  0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
63                  0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
64                  0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
65                  0x09, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01};
66      const BYTE  dataOut_AES192_CTR [] = {
67                  0x1a, 0xbc, 0x93, 0x24, 0x17, 0x52, 0x1c, 0xa2,
68                  0x4f, 0x2b, 0x04, 0x59, 0xfe, 0x7e, 0x6e, 0x0b,
69                  0x09, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef,
70                  0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e, 0x94};
71      #endif
72      #if AES_256
73      const BYTE  key_AES256 [] = {
74                  0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
75                  0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
76                  0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
77                  0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4};
78      const BYTE  dataIn_AES256 [] = {
79                  0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
80                  0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
81                  0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
82                  0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
83      const BYTE  dataOut_AES256_ECB [] = {
84                  0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c,
85                  0x06, 0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8,
86                  0x59, 0x1c, 0xcb, 0x10, 0xd4, 0x10, 0xed, 0x26,
87                  0xdc, 0x5b, 0xa7, 0x4a, 0x31, 0x36, 0x28, 0x70};
88      const BYTE  dataOut_AES256_CBC [] = {
89                  0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba,
90                  0x77, 0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
91                  0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d,
92                  0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d};
93      const BYTE  dataOut_AES256_CFB [] = {
94                  0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
95                  0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
96                  0x39, 0xff, 0xed, 0x14, 0x3b, 0x28, 0xb1, 0xc8,
97                  0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b};
98      const BYTE  dataOut_AES256_OFB [] = {
99                  0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
100                 0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
101                 0x4f, 0xeb, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,
102                 0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d};
103     const BYTE  dataOut_AES256_CTR [] = {
104                 0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
105                 0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
106                 0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
107                 0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5};
108     #endif
```

10.1.11 SymmetricTest.h

```
1   #ifndef     SELF_TEST_DATA
2   #error  "This file many only be included in AlgorithmTests.c"
3   #endif
4   #ifndef     _SYMMETRIC_TEST_H
5   #define     _SYMMETRIC_TEST_H
```

This file contains the structures and data definitions for the symmetric tests. This file references the header file that contains the actual test vectors. This organization was chosen so that the program that is used to generate the test vector values does not have to also re-generate this data.

```
6   #include    "SymmetricTestData.h"
```

```
const SYMMETRIC_TEST_VECTOR   c_symTestValues[NUM_SYMS] = {
#undef  COMMA
#if AES_128
    {ALG_AES_VALUE, 128, key_AES128, 16, sizeof(dataIn_AES128), dataIn_AES128,
    {dataOut_AES128_CTR, dataOut_AES128_OFB, dataOut_AES128_CBC,
     dataOut_AES128_CFB, dataOut_AES128_ECB}}
#   define COMMA ,
#endif
#if AES_192
    COMMA
    {ALG_AES_VALUE, 192, key_AES192, 16, sizeof(dataIn_AES192), dataIn_AES192,
    {dataOut_AES192_CTR, dataOut_AES192_OFB, dataOut_AES192_CBC,
     dataOut_AES192_CFB, dataOut_AES192_ECB}}
#   undef   COMMA
#   define COMMA ,
#endif
#if AES_256
    COMMA
    {ALG_AES_VALUE, 256, key_AES256, 16, sizeof(dataIn_AES256), dataIn_AES256,
    {dataOut_AES256_CTR, dataOut_AES256_OFB, dataOut_AES256_CBC,
    dataOut_AES256_CFB, dataOut_AES256_ECB}}
#   undef  COMMA
#   define COMMA ,
#endif
#if SM4_128
    COMMA
    {ALG_SM4_VALUE, 128, key_SM4128, 16, sizeof(dataIn_SM4128), dataIn_SM4128,
    {dataOut_SM4128_CTR, dataOut_SM4128_OFB, dataOut_SM4128_CBC,
     dataOut_SM4128_CFB, dataOut_AES128_ECB}}
#endif
};
#undef COMMA
#endif  // _SYMMETRIC_TEST_H
```

10.1.12 EccTestData.h

```
#ifdef  SELF_TEST_DATA
TPM2B_TYPE(EC_TEST, 32);
const  TPM_ECC_CURVE        c_testCurve = 00003;
```

The **static** key

```
const  TPM2B_EC_TEST    c_ecTestKey_ds = {{32, {
 0xdf,0x8d,0xa4,0xa3,0x88,0xf6,0x76,0x96,0x89,0xfc,0x2f,0x2d,0xa1,0xb4,0x39,0x7a,
 0x78,0xc4,0x7f,0x71,0x8c,0xa6,0x91,0x85,0xc0,0xbf,0xf3,0x54,0x20,0x91,0x2f,0x73}}};
const  TPM2B_EC_TEST    c_ecTestKey_QsX = {{32, {
 0x17,0xad,0x2f,0xcb,0x18,0xd4,0xdb,0x3f,0x2c,0x53,0x13,0x82,0x42,0x97,0xff,0x8d,
 0x99,0x50,0x16,0x02,0x35,0xa7,0x06,0xae,0x1f,0xda,0xe2,0x9c,0x12,0x77,0xc0,0xf9}}};
const  TPM2B_EC_TEST    c_ecTestKey_QsY = {{32, {
 0xa6,0xca,0xf2,0x18,0x45,0x96,0x6e,0x58,0xe6,0x72,0x34,0x12,0x89,0xcd,0xaa,0xad,
 0xcb,0x68,0xb2,0x51,0xdc,0x5e,0xd1,0x6d,0x38,0x20,0x35,0x57,0xb2,0xfd,0xc7,0x52}}};
```

The **ephemeral** key

```
const  TPM2B_EC_TEST    c_ecTestKey_de = {{32, {
 0xb6,0xb5,0x33,0x5c,0xd1,0xee,0x52,0x07,0x99,0xea,0x2e,0x8f,0x8b,0x19,0x18,0x07,
 0xc1,0xf8,0xdf,0xdd,0xb8,0x77,0x00,0xc7,0xd6,0x53,0x21,0xed,0x02,0x53,0xee,0xac}}};
const  TPM2B_EC_TEST    c_ecTestKey_QeX = {{32, {
 0xa5,0x1e,0x80,0xd1,0x76,0x3e,0x8b,0x96,0xce,0xcc,0x21,0x82,0xc9,0xa2,0xa2,0xed,
 0x47,0x21,0x89,0x53,0x44,0xe9,0xc7,0x92,0xe7,0x31,0x48,0x38,0xe6,0xea,0x93,0x47}}};
const  TPM2B_EC_TEST    c_ecTestKey_QeY = {{32, {
 0x30,0xe6,0x4f,0x97,0x03,0xa1,0xcb,0x3b,0x32,0x2a,0x70,0x39,0x94,0xeb,0x4e,0xea,
 0x55,0x88,0x81,0x3f,0xb5,0x00,0xb8,0x54,0x25,0xab,0xd4,0xda,0xfd,0x53,0x7a,0x18}}};
```

ECDH test results

```
22  const  TPM2B_EC_TEST    c_ecTestEcdh_X = {{32, {
23   0x64,0x02,0x68,0x92,0x78,0xdb,0x33,0x52,0xed,0x3b,0xfa,0x3b,0x74,0xa3,0x3d,0x2c,
24   0x2f,0x9c,0x59,0x03,0x07,0xf8,0x22,0x90,0xed,0xe3,0x45,0xf8,0x2a,0x0a,0xd8,0x1d}}};
25  const  TPM2B_EC_TEST    c_ecTestEcdh_Y = {{32, {
26   0x58,0x94,0x05,0x82,0xbe,0x5f,0x33,0x02,0x25,0x90,0x3a,0x33,0x90,0x89,0xe3,0xe5,
27   0x10,0x4a,0xbc,0x78,0xa5,0xc5,0x07,0x64,0xaf,0x91,0xbc,0xe6,0xff,0x85,0x11,0x40}}};
28  #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
29  TPM2B_TYPE(TEST_VALUE, 64);
30  const TPM2B_TEST_VALUE        c_ecTestValue = {{64, {
31   0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
32   0xf4,0x45,0x7f,0x13,0xd8,0x13,0x81,0x0b,0xe5,0x76,0xbe,0xaa,0xb6,0x3f,0x8d,0x4d,
33   0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xcb,0x0c,0xc7,0x11,0x8d,0xc3,0xff,
34   0x62,0x69,0xa2,0xbe,0x46,0x90,0xe7,0x7d,0x81,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}}};
35  const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
36   0x57,0xf3,0x36,0xb7,0xec,0xc2,0xdd,0x76,0x0e,0xe2,0x81,0x21,0x49,0xc5,0x66,0x11,
37   0x4b,0x8a,0x4f,0x17,0x62,0x82,0xcc,0x06,0xf6,0x64,0x78,0xef,0x6b,0x7c,0xf2,0x6c}}};
38  const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
39   0x1b,0xed,0x23,0x72,0x8f,0x17,0x5f,0x47,0x2e,0xa7,0x97,0x2c,0x51,0x57,0x20,0x70,
40   0x6f,0x89,0x74,0x8a,0xa8,0xf4,0x26,0xf4,0x96,0xa1,0xb8,0x3e,0xe5,0x35,0xc5,0x94}}};
41  const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32, {
42   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x42,0xc6,0x0d,0x2f,
43   0x86,0xa0,0x1c,0x93,0x5e,0x4a,0xad,0x0b,0x67,0xde,0x5d,0x2a,0xb1,0x08,0x4d,0xae}}};
44  const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32, {
45   0xb1,0x37,0x7b,0xff,0xf8,0xf7,0xcf,0x2f,0xa6,0xa9,0x5a,0xb1,0x03,0xc1,0x1e,0xa7,
46   0xf8,0x05,0x33,0xc0,0x0a,0x7b,0xda,0x7b,0x1a,0x00,0x47,0xe1,0x9e,0xbe,0x50,0xe4}}};
47  #endif // SHA1
48  #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
49  TPM2B_TYPE(TEST_VALUE, 64);
50  const TPM2B_TEST_VALUE        c_ecTestValue = {{64, {
51   0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
52   0xf4,0x45,0x7f,0x13,0xd8,0x13,0x81,0x0b,0xe5,0x76,0xbe,0xaa,0xb6,0x3f,0x8d,0x4d,
53   0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xcb,0x0c,0xc7,0x11,0x8d,0xc3,0xff,
54   0x62,0x69,0xa2,0xbe,0x46,0x90,0xe7,0x7d,0x81,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}}};
55  const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
56   0xf5,0x74,0x6d,0xd6,0xc6,0x56,0x86,0xbb,0xba,0x1c,0xba,0x75,0x65,0xee,0x64,0x31,
57   0xce,0x04,0xe3,0x9f,0x24,0x3f,0xbd,0xfe,0x04,0xcd,0xab,0x7e,0xfe,0xad,0xcb,0x82}}};
58  const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
59   0xc2,0x4f,0x32,0xa1,0x06,0xc0,0x85,0x4f,0xc6,0xd8,0x31,0x66,0x91,0x9f,0x79,0xcd,
60   0x5b,0xe5,0x7b,0x94,0xa1,0x91,0x38,0xac,0xd4,0x20,0xa2,0x10,0xf0,0xd5,0x9d,0xbf}}};
61  const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32, {
62   0xad,0xe8,0xd2,0xda,0x57,0x6d,0x4a,0xe5,0xcc,0xcd,0x7c,0x14,0x1a,0x71,0x93,0xab,
63   0x6b,0x9d,0x0f,0x6a,0x97,0xb2,0x32,0x75,0x6c,0xb7,0x82,0x17,0x6b,0x17,0xca,0x1b}}};
64  const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32, {
65   0xd6,0xf1,0xe3,0x05,0xe5,0xdb,0x7c,0x8f,0x67,0x46,0x24,0x0f,0x5a,0x67,0xf8,0x2d,
66   0x5e,0xcb,0xa6,0x8b,0xd7,0x80,0x66,0x83,0x36,0x50,0xd7,0x73,0x7c,0xb8,0x63,0x41}}};
67  #endif // SHA384
68  #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
69  TPM2B_TYPE(TEST_VALUE, 64);
70  const TPM2B_TEST_VALUE        c_ecTestValue = {{64, {
71   0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
72   0xf4,0x45,0x7f,0x13,0xd8,0x13,0x81,0x0b,0xe5,0x76,0xbe,0xaa,0xb6,0x3f,0x8d,0x4d,
73   0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xcb,0x0c,0xc7,0x11,0x8d,0xc3,0xff,
74   0x62,0x69,0xa2,0xbe,0x46,0x90,0xe7,0x7d,0x81,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}}};
75  const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
76   0x04,0x7d,0x54,0xeb,0x04,0x6f,0x56,0xec,0xa2,0x6c,0x38,0x8c,0xeb,0x43,0x0b,0x71,
77   0xf8,0xf2,0xf4,0xa5,0xe0,0x1d,0x3c,0xa2,0x39,0x31,0xe4,0xe7,0x36,0x3b,0xb5,0x5f}}};
78  const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
79   0x8f,0xd0,0x12,0xd9,0x24,0x75,0xf6,0xc4,0x3b,0xb5,0x46,0x75,0x3a,0x41,0x8d,0x80,
80   0x23,0x99,0x38,0xd7,0xe2,0x40,0xca,0x9a,0x19,0x2a,0xfc,0x54,0x75,0xd3,0x4a,0x6e}}};
81  const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32, {
82   0x80,0xd5,0x77,0x52,0xf4,0x51,0x4a,0xe8,0xd0,0x28,0x91,0x98,0xbd,0x36,0x5d,0x89,
83   0x24,0x0b,0xcf,0x08,0x30,0x35,0xc2,0x19,0x9f,0xf9,0x49,0x4f,0x53,0xe5,0x1f,0xe8}}};
84  const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32, {
85   0xf7,0x47,0xda,0x09,0xc6,0xc3,0x28,0x28,0xc2,0xbd,0xbf,0xe6,0xa2,0xec,0x3f,0x44,
```

```
86     0xcf,0xee,0xa2,0x30,0x96,0xe7,0xa4,0xaf,0x1f,0x37,0xac,0x1f,0xc6,0xc1,0xb4,0x25}}};
87   #endif // SHA256
88   #endif // ELF_TEST_DATA
```

10.2 Source

10.2.1 AlgorithmTests.c

10.2.1.1 Introduction

This file contains the code to perform the various self-test functions.

10.2.1.2 Includes and Defines

```
#include    "Tpm.h"
#define     SELF_TEST_DATA
#ifdef SELF_TEST
```

These includes pull in the data structures. They contain data definitions for the various tests.

```
#include    "SelfTest.h"
#include    "SymmetricTest.h"
#include    "RsaTestData.h"
#include    "EccTestData.h"
#include    "HashTestData.h"
#define TEST_DEFAULT_TEST_HASH(vector)                                      \
            if(TEST_BIT(DEFAULT_TEST_HASH, g_toTest))                       \
                TestHash(DEFAULT_TEST_HASH, vector);
```

Make sure that the algorithm has been tested

```
#define CLEAR_BOTH(alg)     {   CLEAR_BIT(alg, *toTest);                    \
                                if(toTest != &g_toTest)                     \
                                    CLEAR_BIT(alg, g_toTest); }
#define SET_BOTH(alg)     {   SET_BIT(alg, *toTest);                        \
                                if(toTest != &g_toTest)                     \
                                    SET_BIT(alg, g_toTest); }
#define TEST_BOTH(alg)       ((toTest != &g_toTest)                          \
                            ? TEST_BIT(alg, *toTest) || TEST_BIT(alg, g_toTest)  \
                            : TEST_BIT(alg, *toTest))
```

Can only cancel if doing a list.

```
#define CHECK_CANCELED                                                  \
    if(_plat__IsCanceled() && toTest != &g_toTest)                      \
        return TPM_RC_CANCELED;
```

10.2.1.3 Hash Tests

10.2.1.3.1 Description

The hash test does a known-value HMAC using the specified hash algorithm.

10.2.1.3.2 TestHash()

The hash test function.

```
static TPM_RC
TestHash(
    TPM_ALG_ID          hashAlg,
    ALGORITHM_VECTOR    *toTest
```

```
28      )
29  {
30      TPM2B_DIGEST            computed;  // value computed
31      UINT16                  digestSize;
32      const TPM2B            *testDigest = NULL;
33      HMAC_STATE              state;
34  //    TPM2B_TYPE(HMAC_BLOCK, DEFAULT_TEST_HASH_BLOCK_SIZE);
35      pAssert(hashAlg != TPM_ALG_NULL);
36      switch(hashAlg)
37      {
38  #if ALG_SHA1
39          case ALG_SHA1_VALUE:
40              testDigest = &c_SHA1_digest.b;
41              break;
42  #endif
43  #if ALG_SHA256
44          case ALG_SHA256_VALUE:
45              testDigest = &c_SHA256_digest.b;
46              break;
47  #endif
48  #if ALG_SHA384
49          case ALG_SHA384_VALUE:
50              testDigest = &c_SHA384_digest.b;
51              break;
52  #endif
53  #if ALG_SHA512
54          case ALG_SHA512_VALUE:
55              testDigest = &c_SHA512_digest.b;
56              break;
57  #endif
58  #if ALG_SM3_256
59          case ALG_SM3_256_VALUE:
60              testDigest = &c_SM3_256_digest.b;
61              break;
62  #endif
63          default:
64              FAIL(FATAL_ERROR_INTERNAL);
65      }
66      // Clear the to-test bits
67      CLEAR_BOTH(hashAlg);
68      // Set the HMAC key to twice the digest size
69      digestSize = CryptHashGetDigestSize(hashAlg);
70      CryptHmacStart(&state, hashAlg, digestSize * 2,
71                     (BYTE *)c_hashTestKey.t.buffer);
72      CryptDigestUpdate(&state.hashState, 2 * CryptHashGetBlockSize(hashAlg),
73                        (BYTE *)c_hashTestData.t.buffer);
74      computed.t.size = digestSize;
75      CryptHmacEnd(&state, digestSize, computed.t.buffer);
76      if((testDigest->size != computed.t.size)
77         || (memcmp(testDigest->buffer, computed.t.buffer, computed.b.size) != 0))
78          SELF_TEST_FAILURE;
79      return TPM_RC_SUCCESS;
80  }
```

10.2.1.4 Symmetric Test Functions

10.2.1.4.1 MakeIv()

Internal function to make the appropriate IV depending on the mode.

```
81  static UINT32
82  MakeIv(
83      TPM_ALG_ID    mode,     // IN: symmetric mode
```

```
84      UINT32        size,     // IN: block size of the algorithm
85      BYTE         *iv        // OUT: IV to fill in
86      )
87  {
88      BYTE          i;
89      if(mode == ALG_ECB_VALUE)
90          return 0;
91      if(mode == ALG_CTR_VALUE)
92      {
93          // The test uses an IV that has 0xff in the last byte
94          for(i = 1; i <= size; i++)
95              *iv++ = 0xff - (BYTE)(size - i);
96      }
97      else
98      {
99          for(i = 0; i < size; i++)
100             *iv++ = i;
101     }
102     return size;
103 }
```

10.2.1.4.2 TestSymmetricAlgorithm()

Function to test a specific algorithm, key size, and mode.

```
104 static void
105 TestSymmetricAlgorithm(
106     const SYMMETRIC_TEST_VECTOR     *test,          //
107     TPM_ALG_ID                       mode           //
108     )
109 {
110     BYTE                            encrypted[MAX_SYM_BLOCK_SIZE * 2];
111     BYTE                            decrypted[MAX_SYM_BLOCK_SIZE * 2];
112     TPM2B_IV                        iv;
113 //
114     // Get the appropriate IV
115     iv.t.size = (UINT16)MakeIv(mode, test->ivSize, iv.t.buffer);
116     // Encrypt known data
117     CryptSymmetricEncrypt(encrypted, test->alg, test->keyBits, test->key, &iv,
118                           mode, test->dataInOutSize, test->dataIn);
119     // Check that it matches the expected value
120     if(!MemoryEqual(encrypted, test->dataOut[mode - ALG_CTR_VALUE],
121                     test->dataInOutSize))
122         SELF_TEST_FAILURE;
123     // Reinitialize the iv for decryption
124     MakeIv(mode, test->ivSize, iv.t.buffer);
125     CryptSymmetricDecrypt(decrypted, test->alg, test->keyBits, test->key, &iv,
126                           mode, test->dataInOutSize,
127                           test->dataOut[mode - ALG_CTR_VALUE]);
128     // Make sure that it matches what we started with
129     if(!MemoryEqual(decrypted, test->dataIn, test->dataInOutSize))
130         SELF_TEST_FAILURE;
131 }
```

10.2.1.4.3 AllSymsAreDone()

Checks if both symmetric algorithms have been tested. This is put here so that addition of a symmetric algorithm will be relatively easy to handle

```
132 static BOOL
133 AllSymsAreDone(
134     ALGORITHM_VECTOR        *toTest
135     )
```

```
{
    return (!TEST_BOTH(ALG_AES_VALUE) && !TEST_BOTH(ALG_SM4_VALUE));
}
```

10.2.1.4.4 AllModesAreDone()

Checks if all the modes have been tested

```
static BOOL
AllModesAreDone(
    ALGORITHM_VECTOR            *toTest
    )
{
    TPM_ALG_ID                  alg;
    for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
        if(TEST_BOTH(alg))
            return FALSE;
    return TRUE;
}
```

10.2.1.4.5 TestSymmetric()

If *alg* is a symmetric block cipher, then all of the modes that are selected are tested. If *alg* is a mode, then all algorithms of that mode are tested.

```
static TPM_RC
TestSymmetric(
    TPM_ALG_ID                   alg,
    ALGORITHM_VECTOR            *toTest
    )
{
    SYM_INDEX                    index;
    TPM_ALG_ID                   mode;
//
    if(!TEST_BIT(alg, *toTest))
        return TPM_RC_SUCCESS;
    if(alg == ALG_AES_VALUE || alg == ALG_SM4_VALUE)
    {
        // Will test the algorithm for all modes and key sizes
        CLEAR_BOTH(alg);
        // A test this algorithm for all modes
        for(index = 0; index < NUM_SYMS; index++)
        {
            if(c_symTestValues[index].alg == alg)
            {
                for(mode = TPM_SYM_MODE_FIRST;
                mode <= TPM_SYM_MODE_LAST;
                    mode++)
                {
                    if(TEST_BIT(mode, *toTest))
                        TestSymmetricAlgorithm(&c_symTestValues[index], mode);
                }
            }
        }
        // if all the symmetric tests are done
        if(AllSymsAreDone(toTest))
        {
            // all symmetric algorithms tested so no modes should be set
            for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
                CLEAR_BOTH(alg);
        }
    }
    else if(TPM_SYM_MODE_FIRST <= alg && alg <= TPM_SYM_MODE_LAST)
```

```
188         {
189             // Test this mode for all key sizes and algorithms
190             for(index = 0; index < NUM_SYMS; index++)
191             {
192                 // The mode testing only comes into play when doing self tests
193                 // by command. When doing self tests by command, the block ciphers are
194                 // tested first. That means that all of their modes would have been
195                 // tested for all key sizes. If there is no block cipher left to
196                 // test, then clear this mode bit.
197                 if(!TEST_BIT(ALG_AES_VALUE, *toTest)
198                    && !TEST_BIT(ALG_SM4_VALUE, *toTest))
199                 {
200                     CLEAR_BOTH(alg);
201                 }
202                 else
203                 {
204                     for(index = 0; index < NUM_SYMS; index++)
205                     {
206                         if(TEST_BIT(c_symTestValues[index].alg, *toTest))
207                             TestSymmetricAlgorithm(&c_symTestValues[index], alg);
208                     }
209                     // have tested this mode for all algorithms
210                     CLEAR_BOTH(alg);
211                 }
212             }
213             if(AllModesAreDone(toTest))
214             {
215                 CLEAR_BOTH(ALG_AES_VALUE);
216                 CLEAR_BOTH(ALG_SM4_VALUE);
217             }
218         }
219         else
220             pAssert(alg == 0 && alg != 0);
221         return TPM_RC_SUCCESS;
222     }
```

10.2.1.5 RSA Tests

```
223 #ifdef TPM_ALG_RSA
```

10.2.1.5.1 Introduction

The tests are for public key only operations and for private key operations. Signature verification and encryption are public key operations. They are tested by using a KVT. For signature verification, this means that a known good signature is checked by _cpri_ValidateSignatureRSA(). If it fails, then the TPM enters failure mode. For encryption, the TPM encrypts known values using the selected scheme and checks that the returned value matches the expected value.

For private key operations, a full scheme check is used. For a signing key, a known key is used to sign a known message. Then that signature is verified. since the signature may involve use of random values, the signature will be different each time and we can't always check that the signature matches a known value. The same technique is used for decryption (RSADP/RSAEP).

When an operation uses the public key and the verification has not been tested, the TPM will do a KVT.

The test for the signing algorithm is built into the call for the algorithm

10.2.1.5.2 RsaKeyInitialize()

The test key is defined by a public modulus and a private prime. The TPM's RSA code computes the second prime and the private exponent.

```
224 static void
225 RsaKeyInitialize(
226     OBJECT          *testObject
227     )
228 {
229     MemoryCopy2B(&testObject->publicArea.unique.rsa.b, (P2B)&c_rsaPublicModulus,
230                  sizeof(c_rsaPublicModulus));
231     MemoryCopy2B(&testObject->sensitive.sensitive.rsa.b, (P2B)&c_rsaPrivatePrime,
232                  sizeof(testObject->sensitive.sensitive.rsa.t.buffer));
233     testObject->publicArea.parameters.rsaDetail.keyBits = RSA_TEST_KEY_SIZE * 8;
234     // Use the default exponent
235     testObject->publicArea.parameters.rsaDetail.exponent = 0;
236     testObject->attributes.privateExp = 0;
237 }
```

10.2.1.5.3 TestRsaEncryptDecrypt()

These test are for an public key encryption that uses a random value

```
238 static TPM_RC
239 TestRsaEncryptDecrypt(
240     TPM_ALG_ID           scheme,            // IN: the scheme
241     ALGORITHM_VECTOR    *toTest             //
242     )
243 {
244     TPM2B_PUBLIC_KEY_RSA             testInput;
245     TPM2B_PUBLIC_KEY_RSA             testOutput;
246     const TPM2B_RSA_TEST_KEY        *kvtValue = &c_RsaesKvt;
247     TPM_RC                           result = TPM_RC_SUCCESS;
248     const TPM2B                     *testLabel = NULL;
249     OBJECT                           testObject;
250     TPMT_RSA_DECRYPT                 rsaScheme;
251 //
252     // Don't need to initialize much of the test object but do need to initialize
253     // the flag indicating that the private exponent has been computed.
254     testObject.attributes.privateExp = CLEAR;
255     RsaKeyInitialize(&testObject);
256     rsaScheme.scheme = scheme;
257     rsaScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
258     CLEAR_BOTH(scheme);
259     CLEAR_BOTH(ALG_NULL_VALUE);
260     if(scheme == TPM_ALG_NULL)
261     {
262         // This is an encryption scheme using the private key without any encoding.
263         memcpy(testInput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue));
264         testInput.t.size = sizeof(c_RsaTestValue);
265         if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
266                                              &testObject, &rsaScheme, NULL, NULL))
267             SELF_TEST_FAILURE;
268         if(!MemoryEqual(testOutput.t.buffer, c_RsaepKvt.buffer, c_RsaepKvt.size))
269             SELF_TEST_FAILURE;
270         MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
271         if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
272                                              &testObject, &rsaScheme, NULL))
273             SELF_TEST_FAILURE;
274         if(!MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
275                         sizeof(c_RsaTestValue)))
276             SELF_TEST_FAILURE;
277     }
278     else
279     {
280         // ALG_RSAES_VALUE:
281         // This is an decryption scheme using padding according to
282         // PKCS#1v2.1, 7.2. This padding uses random bits. To test a public
```

```
283             // key encryption that uses random data, encrypt a value and then
284             // decrypt the value and see that we get the encrypted data back.
285             // The hash is not used by this encryption so it can be TMP_ALG_NULL
286             // ALG_OAEP_VALUE:
287             // This is also an decryption scheme and it also uses a
288             // pseudo-random
289             // value. However, this also uses a hash algorithm. So, we may need
290             // to test that algorithm before use.
291             if(scheme == ALG_OAEP_VALUE)
292             {
293                 TEST_DEFAULT_TEST_HASH(toTest);
294                 kvtValue = &c_OaepKvt;
295                 testLabel = OAEP_TEST_STRING;
296             }
297             else if(scheme == ALG_RSAES_VALUE)
298             {
299                 kvtValue = &c_RsaesKvt;
300                 testLabel = NULL;
301             }
302             else
303                 SELF_TEST_FAILURE;
304             // Only use a digest-size portion of the test value
305             memcpy(testInput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
306             testInput.t.size = DEFAULT_TEST_DIGEST_SIZE;
307             // See if the encryption works
308             if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
309                                                  &testObject, &rsaScheme, testLabel,
310                                                  NULL))
311                 SELF_TEST_FAILURE;
312             MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
313             // see if we can decrypt this value and get the original data back
314             if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
315                                                  &testObject, &rsaScheme, testLabel))
316                 SELF_TEST_FAILURE;
317             // See if the results compare
318             if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
319                || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
320                                DEFAULT_TEST_DIGEST_SIZE))
321                 SELF_TEST_FAILURE;
322             // Now check that the decryption works on a known value
323             MemoryCopy2B(&testInput.b, (P2B)kvtValue,
324                          sizeof(testInput.t.buffer));
325             if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
326                                                  &testObject, &rsaScheme, testLabel))
327                 SELF_TEST_FAILURE;
328             if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
329                || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
330                                DEFAULT_TEST_DIGEST_SIZE))
331                 SELF_TEST_FAILURE;
332         }
333         return result;
334     }
```

10.2.1.5.4 TestRsaSignAndVerify()

This function does the testing of the RSA sign and verification functions. This test does a KVT.

```
335     static TPM_RC
336     TestRsaSignAndVerify(
337         TPM_ALG_ID              scheme,
338         ALGORITHM_VECTOR        *toTest
339         )
340     {
341         TPM_RC                      result = TPM_RC_SUCCESS;
```

```
342         OBJECT                      testObject;
343         TPM2B_DIGEST                testDigest;
344         TPMT_SIGNATURE              testSig;
345         // Do a sign and signature verification.
346         // RSASSA:
347         // This is a signing scheme according to PKCS#1-v2.1 8.2. It does not
348         // use random data so there is a KVT for the signing operation. On
349         // first use of the scheme for signing, use the TPM's RSA key to
350         // sign a portion of c_RsaTestData and compare the results to c_RsassaKvt. Then
351         // decrypt the data to see that it matches the starting value. This verifies
352         // the signature with a KVT
353         // Clear the bits indicating that the function has not been checked. This is to
354         // prevent looping
355         CLEAR_BOTH(scheme);
356         CLEAR_BOTH(ALG_NULL_VALUE);
357         CLEAR_BOTH(ALG_RSA_VALUE);
358         RsaKeyInitialize(&testObject);
359         memcpy(testDigest.t.buffer, (BYTE *)c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
360         testDigest.t.size = DEFAULT_TEST_DIGEST_SIZE;
361         testSig.sigAlg = scheme;
362         testSig.signature.rsapss.hash = DEFAULT_TEST_HASH;
363         // RSAPSS:
364         // This is a signing scheme a according to PKCS#1-v2.2 8.1 it uses
365         // random data in the signature so there is no KVT for the signing
366         // operation. To test signing, the TPM will use the TPM's RSA key
367         // to sign a portion of c_RsaTestValue and then it will verify the
368         // signature. For verification, c_RsapssKvt is verified before the
369         // user signature blob is verified. The worst case for testing of this
370         // algorithm is two private and one public key operation.
371         // The process is to sign known data. If RSASSA is being done, verify that the
372         // signature matches the precomputed value. For both, use the signed value and
373         // see that the verification says that it is a good signature. Then
374         // if testing RSAPSS, do a verify of a known good signature. This ensures that
375         // the validation function works.
376         if(TPM_RC_SUCCESS != CryptRsaSign(&testSig, &testObject, &testDigest, NULL))
377             SELF_TEST_FAILURE;
378         // For RSASSA, make sure the results is what we are looking for
379         if(testSig.sigAlg == ALG_RSASSA_VALUE)
380         {
381             if(testSig.signature.rsassa.sig.t.size != RSA_TEST_KEY_SIZE
382                || !MemoryEqual(c_RsassaKvt.buffer,
383                                testSig.signature.rsassa.sig.t.buffer,
384                                RSA_TEST_KEY_SIZE))
385                 SELF_TEST_FAILURE;
386         }
387         // See if the TPM will validate its own signatures
388         if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
389                                                        &testDigest))
390             SELF_TEST_FAILURE;
391         // If this is RSAPSS, check the verification with known signature
392         // Have to copy because  CrytpRsaValidateSignature() eats the signature
393         if(ALG_RSAPSS_VALUE == scheme)
394         {
395             MemoryCopy2B(&testSig.signature.rsapss.sig.b, (P2B)&c_RsapssKvt,
396                          sizeof(testSig.signature.rsapss.sig.t.buffer));
397             if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
398                                                            &testDigest))
399                 SELF_TEST_FAILURE;
400         }
401         return result;
402     }
```

10.2.1.5.5 TestRSA()

Function uses the provided vector to indicate which tests to run. It will clear the vector after each test is run and also clear *g_toTest*

```
403  static TPM_RC
404  TestRsa(
405      TPM_ALG_ID               alg,
406      ALGORITHM_VECTOR        *toTest
407      )
408  {
409      TPM_RC                   result = TPM_RC_SUCCESS;
410  //
411      switch(alg)
412      {
413          case ALG_NULL_VALUE:
414          // This is the RSAEP/RSADP function. If we are processing a list, don't
415          // need to test these now because any other test will validate
416          // RSAEP/RSADP. Can tell this is list of test by checking to see if
417          // 'toTest' is pointing at g_toTest. If so, this is an isolated test
418          // an need to go ahead and do the test;
419              if((toTest == &g_toTest)
420                 || (!TEST_BIT(ALG_RSASSA_VALUE, *toTest)
421                     && !TEST_BIT(ALG_RSAES_VALUE, *toTest)
422                     && !TEST_BIT(ALG_RSAPSS_VALUE, *toTest)
423                     && !TEST_BIT(ALG_OAEP_VALUE, *toTest)))
424                 // Not running a list of tests or no other tests on the list
425                 // so run the test now
426                  result = TestRsaEncryptDecrypt(alg, toTest);
427              // if not running the test now, leave the bit on, just in case things
428              // get interrupted
429              break;
430          case ALG_OAEP_VALUE:
431          case ALG_RSAES_VALUE:
432              result = TestRsaEncryptDecrypt(alg, toTest);
433              break;
434          case ALG_RSAPSS_VALUE:
435          case ALG_RSASSA_VALUE:
436              result = TestRsaSignAndVerify(alg, toTest);
437              break;
438          default:
439              SELF_TEST_FAILURE;
440      }
441      return result;
442  }
443  #endif // TPM_ALG_RSA
```

10.2.1.6 ECC Tests

```
444  #ifdef TPM_ALG_ECC
```

10.2.1.6.1 LoadEccParameter()

This function is mostly for readability and type checking

```
445  static void
446  LoadEccParameter(
447      TPM2B_ECC_PARAMETER          *to,       // target
448      const TPM2B_EC_TEST          *from      // source
449      )
450  {
451      MemoryCopy2B(&to->b, &from->b, sizeof(to->t.buffer));
```

```
452 }
```

10.2.1.6.2 LoadEccPoint()

```
453 static void
454 LoadEccPoint(
455     TPMS_ECC_POINT              *point,       // target
456     const TPM2B_EC_TEST         *x,             // source
457     const TPM2B_EC_TEST          *y
458     )
459 {
460     MemoryCopy2B(&point->x.b, (TPM2B *)x, sizeof(point->x.t.buffer));
461     MemoryCopy2B(&point->y.b, (TPM2B *)y, sizeof(point->y.t.buffer));
462 }
```

10.2.1.6.3 TestECDH()

This test does a KVT on a point multiply.

```
463 static TPM_RC
464 TestECDH(
465     TPM_ALG_ID          scheme,         // IN: for consistency
466     ALGORITHM_VECTOR    *toTest         // IN/OUT: modified after test is run
467     )
468 {
469     TPMS_ECC_POINT          Z;
470     TPMS_ECC_POINT          Qe;
471     TPM2B_ECC_PARAMETER     ds;
472     TPM_RC                  result = TPM_RC_SUCCESS;
473 //
474     NOT_REFERENCED(scheme);
475     CLEAR_BOTH(ALG_ECDH_VALUE);
476     LoadEccParameter(&ds, &c_ecTestKey_ds);
477     LoadEccPoint(&Qe, &c_ecTestKey_QeX, &c_ecTestKey_QeY);
478     if(TPM_RC_SUCCESS != CryptEccPointMultiply(&Z, c_testCurve, &Qe, &ds,
479                                                NULL, NULL))
480         SELF_TEST_FAILURE;
481     if(!MemoryEqual2B(&c_ecTestEcdh_X.b, &Z.x.b)
482        || !MemoryEqual2B(&c_ecTestEcdh_Y.b, &Z.y.b))
483         SELF_TEST_FAILURE;
484     return result;
485 }
486 static TPM_RC
487 TestEccSignAndVerify(
488     TPM_ALG_ID                  scheme,
489     ALGORITHM_VECTOR           *toTest
490     )
491 {
492     OBJECT                      testObject;
493     TPMT_SIGNATURE              testSig;
494     TPMT_ECC_SCHEME             eccScheme;
495     testSig.sigAlg = scheme;
496     testSig.signature.ecdsa.hash = DEFAULT_TEST_HASH;
497     eccScheme.scheme = scheme;
498     eccScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
499     CLEAR_BOTH(scheme);
500     CLEAR_BOTH(ALG_ECDH_VALUE);
501     // ECC signature verification testing uses a KVT.
502     switch(scheme)
503     {
504         case ALG_ECDSA_VALUE:
505             LoadEccParameter(&testSig.signature.ecdaa.signatureR, &c_TestEcDsa_r);
506             LoadEccParameter(&testSig.signature.ecdaa.signatureS, &c_TestEcDsa_s);
```

```
507             break;
508         case ALG_ECSCHNORR_VALUE:
509             LoadEccParameter(&testSig.signature.ecdaa.signatureR,
510                              &c_TestEcSchnorr_r);
511             LoadEccParameter(&testSig.signature.ecdaa.signatureS,
512                              &c_TestEcSchnorr_s);
513             break;
514         case ALG_SM2_VALUE:
515             // don't have a test for SM2
516             return TPM_RC_SUCCESS;
517         default:
518             SELF_TEST_FAILURE;
519             break;
520     }
521     TEST_DEFAULT_TEST_HASH(toTest);
522     // Have to copy the key. This is because the size used in the test vectors
523     // is the size of the ECC parameter for the test key while the size of a point
524     // is TPM dependent
525     MemoryCopy2B(&testObject.sensitive.sensitive.ecc.b, &c_ecTestKey_ds.b,
526                  sizeof(testObject.sensitive.sensitive.ecc.t.buffer));
527     LoadEccPoint(&testObject.publicArea.unique.ecc, &c_ecTestKey_QsX,
528                  &c_ecTestKey_QsY);
529     testObject.publicArea.parameters.eccDetail.curveID = c_testCurve;
530     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
531                                                    (TPM2B_DIGEST *)&c_ecTestValue.b))
532     {
533 //??? Don't have a valid test for ECSCORR right now because the algorithm has
534 //??? changed and the KVT has not been updated.
535         if(scheme != ALG_ECSCHNORR_VALUE)
536             SELF_TEST_FAILURE;
537     }
538     CHECK_CANCELED;
539     // Now sign and verify some data
540     if(TPM_RC_SUCCESS != CryptEccSign(&testSig, &testObject,
541                                       (TPM2B_DIGEST *)&c_ecTestValue,
542                                       &eccScheme, NULL))
543         SELF_TEST_FAILURE;
544     CHECK_CANCELED;
545     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
546                                                    (TPM2B_DIGEST *)&c_ecTestValue))
547         SELF_TEST_FAILURE;
548     CHECK_CANCELED;
549     return TPM_RC_SUCCESS;
550 }
551 static TPM_RC
552 TestEcc(
553     TPM_ALG_ID              alg,
554     ALGORITHM_VECTOR        *toTest
555     )
556 {
557     TPM_RC                  result = TPM_RC_SUCCESS;
558     NOT_REFERENCED(toTest);
559     switch(alg)
560     {
561         case ALG_ECC_VALUE:
562         case ALG_ECDH_VALUE:
563             // If this is in a loop then see if another test is going to deal with
564             // this.
565             // If toTest is not a self-test list
566             if((toTest == &g_toTest)
567                 // or this is the only ECC test in the list
568                || !(TEST_BIT(ALG_ECDSA_VALUE, *toTest)
569                     || TEST_BIT(ALG_ECSCHNORR, *toTest)
570                     || TEST_BIT(ALG_SM2_VALUE, *toTest)))
571             {
572                 result = TestECDH(alg, toTest);
```

```
573             }
574             break;
575         case ALG_ECDSA_VALUE:
576         case ALG_ECSCHNORR_VALUE:
577         case ALG_SM2_VALUE:
578             result = TestEccSignAndVerify(alg, toTest);
579             break;
580         default:
581             SELF_TEST_FAILURE;
582             break;
583     }
584     return result;
585 }
586 #endif // TPM_ALG_ECC
```

10.2.1.6.4 TestAlgorithm()

Dispatches to the correct test function for the algorithm. If algorithm is not If *toTest* is not NULL, then the test decisions are based on the algorithm selections in *toTest*. Otherwise, *g_toTest* is used. When bits are clear in *g_toTest* they will also be cleared *toTest*. If there doesn't happen to be a test for the algorithm, its associated bit quietly cleared. If *alg* is zero (TPM_ALG_ERROR), then the *toTest* vector is cleared of any bits for which there is a test (i.e. no tests are actually run but the vector is cleared.

NOTE: *toTest* will only ever have bits set for implemented algorithms but *alg* can be anything.

Error Returns	Meaning
TPM_RC_SUCCESS	test complete
TPM_RC_CANCELED	test was canceled

```
587 LIB_EXPORT
588 TPM_RC
589 TestAlgorithm(
590     TPM_ALG_ID              alg,
591     ALGORITHM_VECTOR        *toTest
592     )
593 {
594     TPM_ALG_ID              first = (alg == TPM_ALG_ERROR) ? TPM_ALG_FIRST : alg;
595     TPM_ALG_ID              last = (alg == TPM_ALG_ERROR) ? TPM_ALG_LAST : alg;
596     BOOL                    doTest = (alg != TPM_ALG_ERROR);
597     TPM_RC                  result = TPM_RC_SUCCESS;
598     if(toTest == NULL)
599         toTest = &g_toTest;
600     for(alg = first; (alg <= last); alg++)
601     {
602         // if alg was not TPM_ALG_ERROR, then we will be cycling through
603         // values, some of which may not be implemented. If the bit in toTest
604         // happens to be set, then we could either generated an assert, or just
605         // silently CLEAR it Decided to just clear.
606         if(!TEST_BIT(alg, g_implementedAlgorithms))
607         {
608             CLEAR_BIT(alg, *toTest);
609             continue;
610         }
611         // Process whatever is left.
612         // NOTE: since this switch will only be called if the algorithm is
613         // implemented, it i not necessary to modify this list except to comment out
614         // the algorithms for which there is no test
615         switch(alg)
616         {
617         // Symmetric block ciphers
618             case ALG_AES_VALUE:
```

```
619             // if SM4 is implemented, its test is like other block ciphers but there
620             // aren't any test vectors for it yet
621 //            case ALG_SM4_VALUE:
622         // Symmetric modes
623             case ALG_CFB_VALUE:
624                 if(doTest)
625                     result = TestSymmetric(alg, toTest);
626                 break;
627             case ALG_CTR_VALUE:
628             case ALG_OFB_VALUE:
629             case ALG_CBC_VALUE:
630             case ALG_ECB_VALUE:
631                 if(doTest)
632                     result = TestSymmetric(alg, toTest);
633                 else
634                     // If doing the initialization of g_toTest vector, only need
635                     // to test one of the modes for the symmetric algorithms. If
636                     // initializing for a SelfTest(FULL_TEST), allow all the modes.
637                     if(toTest == &g_toTest)
638                         CLEAR_BIT(alg, *toTest);
639                 break;
640             case ALG_HMAC_VALUE:
641                 CLEAR_BOTH(alg);
642                 if(doTest)
643                     TestHash(DEFAULT_TEST_HASH, toTest);
644                 else
645                     SET_BOTH(DEFAULT_TEST_HASH);
646                 break;
647 #ifdef TPM_ALG_SHA1
648             case ALG_SHA1_VALUE:
649 #endif // TPM_ALG_SHA1
650 #ifdef TPM_ALG_SHA256
651             case ALG_SHA256_VALUE:
652 #endif // TPM_ALG_SHA256
653 #ifdef TPM_ALG_SHA384
654             case ALG_SHA384_VALUE:
655 #endif // TPM_ALG_SHA384
656 #ifdef TPM_ALG_SHA512
657             case ALG_SHA512_VALUE:
658 #endif // TPM_ALG_SHA512
659             // if SM3 is implemented its test is like any other hash, but there
660             // aren't any test vectors yet.
661 //            case ALG_SM3_256_VALUE:
662                 if(doTest)
663                     result = TestHash(alg, toTest);
664                 break;
665     // RSA-dependent
666 #ifdef TPM_ALG_RSA
667             case ALG_RSA_VALUE:
668                 CLEAR_BOTH(alg);
669                 if(doTest)
670                     result = TestRsa(TPM_ALG_NULL, toTest);
671                 else
672                     SET_BOTH(TPM_ALG_NULL);
673                 break;
674             case ALG_RSASSA_VALUE:
675             case ALG_RSAES_VALUE:
676             case ALG_RSAPSS_VALUE:
677             case ALG_OAEP_VALUE:
678             case ALG_NULL_VALUE:    // used or RSADP
679                 if(doTest)
680                     result = TestRsa(alg, toTest);
681                 break;
682 #endif // TPM_ALG_RSA
683 #ifdef TPM_ALG_ECC
684     // ECC dependent but no tests
```

```
685         //        case ALG_ECDAA_VALUE:
686         //        case ALG_ECMQV_VALUE:
687         //        case ALG_KDF1_SP800_56a_VALUE:
688         //        case ALG_KDF2_VALUE:
689         //        case ALG_KDF1_SP800_108_VALUE:
690         //        case ALG_MGF1_VALUE:
691                 case ALG_ECC_VALUE:
692                     CLEAR_BOTH(alg);
693                     if(doTest)
694                         result = TestEcc(ALG_ECDH_VALUE, toTest);
695                     else
696                         SET_BOTH(ALG_ECDH_VALUE);
697                     break;
698                 case ALG_ECDSA_VALUE:
699                 case ALG_ECDH_VALUE:
700                 case ALG_ECSCHNORR_VALUE:
701     //            case ALG_SM2_VALUE:
702                     if(doTest)
703                         result = TestEcc(alg, toTest);
704                     break;
705     #endif // TPM_ALG_ECC
706                 default:
707                     CLEAR_BIT(alg, *toTest);
708                     break;
709             }
710             if(result != TPM_RC_SUCCESS)
711                 break;
712         }
713         return result;
714     }
715     #endif // SELF_TESTS
```

10.2.2 BnConvert.c

10.2.2.1 Introduction

This file contains the basic conversion functions that will convert TPM2B to/from the internal format. The internal format is a *bigNum*,

10.2.2.2 Includes

```
1   #include "Tpm.h"
```

10.2.2.3 Functions

10.2.2.3.1 BnFromBytes()

This function will convert a big-endian byte array to the internal number format. If bn is NULL, then the output is NULL. If bytes is null or the required size is 0, then the output is set to zero

```
2   LIB_EXPORT bigNum
3   BnFromBytes(
4       bigNum           bn,
5       const BYTE      *bytes,
6       NUMBYTES         nBytes
7       )
8   {
9       const BYTE      *pFrom; // 'p' points to the least significant bytes of source
10      BYTE            *pTo;   // points to least significant bytes of destination
11      crypt_uword_t    size;
```

```
12  //
13      size = (bytes != NULL) ? BYTES_TO_CRYPT_WORDS(nBytes) : 0;
14      // make sure things fit
15      pAssert(BnGetAllocated(bn) >= size);
16      // If nothing in, nothing out
17      if(bn == NULL)
18          return NULL;
19      if(size > 0)
20      {
21          // Clear the topmost word in case it is not filled with data
22          bn->d[size - 1] = 0;
23          // Moving the input bytes from the end of the list (LSB) end
24          pFrom = bytes + nBytes - 1;
25          // To the LS0 of the LSW of the bigNum.
26          pTo = (BYTE *)bn->d;
27          for(; nBytes != 0; nBytes--)
28              *pTo++ = *pFrom--;
29          // For a little-endian machine, the conversion is a straight byte
30          // reversal. For a big-endian machine, we have to put the words in
31          // big-endian byte order
32  #if BIG_ENDIAN_TPM
33          {
34              crypt_word_t   t;
35              for(t = (crypt_word_t)size - 1; t >= 0; t--)
36                  bn->d[t] = SWAP_CRYPT_WORD(bn->d[t]);
37          }
38  #endif
39      }
40      BnSetTop(bn, size);
41      return bn;
42  }
```

10.2.2.3.2 BnFrom2B()

Convert an TPM2B to a BIG_NUM. If the input value does not exist, or the output does not exist, or the input will not fit into the output the function returns NULL

```
43  LIB_EXPORT bigNum
44  BnFrom2B(
45      bigNum           bn,         // OUT:
46      const TPM2B     *a2B         // IN: number to convert
47      )
48  {
49      if(a2B != NULL)
50          return BnFromBytes(bn, a2B->buffer, a2B->size);
51      // Make sure that the number has an initialized value rather than whatever
52      // was there before
53      BnSetTop(bn, 0);
54      return NULL;
55  }
```

10.2.2.3.3 BnFromHex()

Convert a hex string into a *bigNum*. This is primarily used in debugging.

```
56  LIB_EXPORT bigNum
57  BnFromHex(
58      bigNum          bn,         // OUT:
59      const char      *hex        // IN:
60      )
61  {
62  #define FromHex(a)  ((a) - (((a) > 'a') ? ('a' + 10)                    \
63                                          : ((a) > 'A') ? ('A' - 10) : '0'))
```

```
64      unsigned             i;
65      unsigned             wordCount;
66      const char          *p;
67      BYTE                *d = (BYTE *)&(bn->d[0]);
68      i = strlen(hex);
69      wordCount = BYTES_TO_CRYPT_WORDS((i + 1) / 2);
70      if((i == 0) || (wordCount >= BnGetAllocated(bn)))
71          BnSetWord(bn, 0);
72      else
73      {
74          bn->d[wordCount - 1] = 0;
75          p = hex + i - 1;
76          for(;i > 1; i -= 2)
77          {
78              BYTE a;
79              a = FromHex(*p);
80              p--;
81              *d++ = a + (FromHex(*p) << 4);
82              p--;
83          }
84          if(i == 1)
85              *d = FromHex(*p);
86      }
87  #if BIG_ENDIAN_TPM == NO
88      for(i = 0; i < wordCount; i++)
89          bn->d[i] = SWAP_CRYPT_WORD(bn->d[i]);
90  #endif // BIG_ENDIAN_TPM
91      BnSetTop(bn, wordCount);
92      return bn;
93  }
```

10.2.2.3.4 BnToBytes()

This function converts a BIG_NUM to a byte array. If size is not large enough to hold the *bigNum* value, then the function return FALSE. Otherwise, it converts the *bigNum* to a big-endian byte string and sets *size* to the normalized value. If *size* is an input 0, then the receiving buffer is guaranteed to be large enough for the result and the size will be set to the size required for *bigNum* (leading zeros suppressed).

```
94  LIB_EXPORT BOOL
95  BnToBytes(
96      bigConst             bn,
97      BYTE                *buffer,
98      NUMBYTES            *size           // This the number of bytes that are
99                                          // available in the buffer. The result
100                                         // should be this big.
101     )
102 {
103     crypt_uword_t        requiredSize;
104     BYTE                *pFrom;
105     BYTE                *pTo;
106     crypt_uword_t        count;
107 //
108     // validate inputs
109     pAssert(bn != NULL && buffer != NULL && size != NULL);
110     requiredSize = (BnSizeInBits(bn) + 7) / 8;
111     if(*size == 0)
112         *size = (NUMBYTES)requiredSize;
113     pAssert(requiredSize <= *size);
114 #if BIG_ENDIAN_TPM
115     for(count = 0; count < bn->size; count++)
116         bn->d[count] = SWAP_CRYPT_WORD(bn->d[count]);
117 #endif
118     count = *size;
119     pFrom = (BYTE *)(&bn->d[0]) + requiredSize - 1;
```

```
120         pTo = buffer;
121         for(count = *size; count > requiredSize; count--)
122             *pTo++ = 0;
123         for(; requiredSize > 0; requiredSize--)
124             *pTo++ = *pFrom--;
125     #if BIG_ENDIAN_TPM
126         for(count = 0; count < bn->size; count++)
127             bn->d[count] = SWAP_CRYPT_WORD(bn->d[count]);
128     #endif
129         return TRUE;
130     }
```

10.2.2.3.5 BnTo2B()

Function to convert a BIG_NUM to TPM2B. The TPM2B size is set to the requested *size* which may require padding. If *size* is non-zero and less than required by the value in *bn* then an error is returned. If *size* is zero, then the TPM2B is assumed to be large enough for the data and a2b->size will be adjusted accordingly.

```
131     LIB_EXPORT BOOL
132     BnTo2B(
133         bigConst         bn,                // IN:
134         TPM2B           *a2B,               // OUT:
135         NUMBYTES         size               // IN: the desired size
136         )
137     {
138         // Set the output size
139         a2B->size = size;
140         return BnToBytes(bn, a2B->buffer, &a2B->size);
141     }
142     #ifdef TPM_ALG_ECC
```

10.2.2.3.6 BnPointFrom2B()

Function to create a BIG_POINT structure from a 2B point. A point is going to be two ECC values in the same buffer. The values are going to be the size of the modulus. They are in modular form.

```
143     LIB_EXPORT bn_point_t    *
144     BnPointFrom2B(
145         bigPoint             ecP,          // OUT: the preallocated point structure
146         TPMS_ECC_POINT      *p             // IN: the number to convert
147         )
148     {
149         if(p == NULL)
150             return NULL;
151         if(NULL != ecP)
152         {
153             BnFrom2B(ecP->x, &p->x.b);
154             BnFrom2B(ecP->y, &p->y.b);
155             BnSetWord(ecP->z, 1);
156         }
157         return ecP;
158     }
```

10.2.2.3.7 BnPointTo2B()

This function converts a BIG_POINT into a TPMS_ECC_POINT. A TPMS_ECC_POINT contains two TPM2B_ECC_PARAMETER values. The maximum size of the parameters is dependent on the maximum EC key size used in an implementation. The presumption is that the TPMS_ECC_POINT is large enough to hold 2 TPM2B values, each as large as a MAX_ECC_PARAMETER_BYTES


```
159 LIB_EXPORT BOOL
160 BnPointTo2B(
161     TPMS_ECC_POINT  *p,             // OUT: the converted 2B structure
162     bigPoint         ecP,           // IN: the values to be converted
163     bigCurve         E              // IN: curve descriptor for the point
164     )
165 {
166     UINT16           size = (UINT16)BITS_TO_BYTES(
167         BnMsb(CurveGetOrder(AccessCurveData(E))));
168     pAssert(p && ecP && E);
169     pAssert(BnEqualWord(ecP->z, 1));
170     BnTo2B(ecP->x, &p->x.b, size);
171     BnTo2B(ecP->y, &p->y.b, size);
172     return TRUE;
173 }
174 #endif // TPM_ALG_ECC
```

10.2.3 BnEccData.c

```
1  #include    "Tpm.h"
```

both the new, refactored code and the old code (this is necessary so that errata can be handled). Another script *(BnEccData*().pl) does the conversion and generates BnEccData.c for use in the refactored code.

```
2  #if defined TPM_ALG_ECC && defined USE_BN_ECC_DATA
3  const struct {
4      crypt_uword_t       allocated;
5      crypt_uword_t       size;
6      crypt_uword_t       d[BYTES_TO_CRYPT_WORDS(1)];
7  } BN_ZERO = {BYTES_TO_CRYPT_WORDS(4), BYTES_TO_CRYPT_WORDS(0), {0}};
8  const struct {
9      crypt_uword_t       allocated;
10     crypt_uword_t       size;
11     crypt_uword_t       d[BYTES_TO_CRYPT_WORDS(1)];
12 } BN_ONE = {BYTES_TO_CRYPT_WORDS(1), BYTES_TO_CRYPT_WORDS(1), {1}};
```

Defines for the sizes of ECC parameters

```
13 #if defined ECC_NIST_P192 && ECC_NIST_P192 == YES
14 const struct {
15     crypt_uword_t    allocated;
16     crypt_uword_t    size;
17     crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(24)];
18     } NIST_P192_p = {BYTES_TO_CRYPT_WORDS(24), BYTES_TO_CRYPT_WORDS(24),
19        {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
20         TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
21         TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)}};
22 const struct {
23     crypt_uword_t    allocated;
24     crypt_uword_t    size;
25     crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(24)];
26     } NIST_P192_a = {BYTES_TO_CRYPT_WORDS(24), BYTES_TO_CRYPT_WORDS(24),
27        {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC),
28         TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
29         TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)}};
30 const struct {
31     crypt_uword_t    allocated;
32     crypt_uword_t    size;
33     crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(24)];
34     } NIST_P192_b = {BYTES_TO_CRYPT_WORDS(24), BYTES_TO_CRYPT_WORDS(24),
35        {TO_CRYPT_WORD_64(0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1),
36         TO_CRYPT_WORD_64(0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49),
37         TO_CRYPT_WORD_64(0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7)}};
```



```
38  const struct {
39      crypt_uword_t    allocated;
40      crypt_uword_t    size;
41      crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(24)];
42      } NIST_P192_gX = {BYTES_TO_CRYPT_WORDS(24), BYTES_TO_CRYPT_WORDS(24),
43         {TO_CRYPT_WORD_64(0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12),
44          TO_CRYPT_WORD_64(0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00),
45          TO_CRYPT_WORD_64(0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6)}};
46  const struct {
47      crypt_uword_t    allocated;
48      crypt_uword_t    size;
49      crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(24)];
50      } NIST_P192_gY = {BYTES_TO_CRYPT_WORDS(24), BYTES_TO_CRYPT_WORDS(24),
51         {TO_CRYPT_WORD_64(0x73, 0xF9, 0x77, 0xA1, 0x1E, 0x79, 0x48, 0x11),
52          TO_CRYPT_WORD_64(0x63, 0x10, 0x11, 0xED, 0x6B, 0x24, 0xCD, 0xD5),
53          TO_CRYPT_WORD_64(0x07, 0x19, 0x2B, 0x95, 0xFF, 0xC8, 0xDA, 0x78)}};
54  const struct {
55      crypt_uword_t    allocated;
56      crypt_uword_t    size;
57      crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(24)];
58      } NIST_P192_n = {BYTES_TO_CRYPT_WORDS(24), BYTES_TO_CRYPT_WORDS(24),
59         {TO_CRYPT_WORD_64(0x14, 0x6B, 0xC9, 0xB1, 0xB4, 0xD2, 0x28, 0x31),
60          TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x99, 0xDE, 0xF8, 0x36),
61          TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)}};
62  #define NIST_P192_h BN_ONE
63  const ECC_CURVE_DATA NIST_P192 = {
64      (bigNum)&NIST_P192_p, (bigNum)&NIST_P192_n, (bigNum)&NIST_P192_h,
65      (bigNum)&NIST_P192_a,  (bigNum)&NIST_P192_b,
66      {(bigNum)&NIST_P192_gX, (bigNum)&NIST_P192_gY, (bigNum)&BN_ONE}};
67  #endif // ECC_NIST_P192
68  #if defined ECC_NIST_P224 && ECC_NIST_P224 == YES
69  const struct {
70      crypt_uword_t    allocated;
71      crypt_uword_t    size;
72      crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(28)];
73      } NIST_P224_p = {BYTES_TO_CRYPT_WORDS(28), BYTES_TO_CRYPT_WORDS(28),
74         {TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01),
75          TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
76          TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
77          TO_CRYPT_WORD_32(0xFF, 0xFF, 0xFF, 0xFF)}};
78  const struct {
79      crypt_uword_t    allocated;
80      crypt_uword_t    size;
81      crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(28)];
82      } NIST_P224_a = {BYTES_TO_CRYPT_WORDS(28), BYTES_TO_CRYPT_WORDS(28),
83         {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
84          TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
85          TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
86          TO_CRYPT_WORD_32(0xFF, 0xFF, 0xFF, 0xFF)}};
87  const struct {
88      crypt_uword_t    allocated;
89      crypt_uword_t    size;
90      crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(28)];
91      } NIST_P224_b = {BYTES_TO_CRYPT_WORDS(28), BYTES_TO_CRYPT_WORDS(28),
92         {TO_CRYPT_WORD_64(0x27, 0x0B, 0x39, 0x43, 0x23, 0x55, 0xFF, 0xB4),
93          TO_CRYPT_WORD_64(0x50, 0x44, 0xB0, 0xB7, 0xD7, 0xBF, 0xD8, 0xBA),
94          TO_CRYPT_WORD_64(0x0C, 0x04, 0xB3, 0xAB, 0xF5, 0x41, 0x32, 0x56),
95          TO_CRYPT_WORD_32(0xB4, 0x05, 0x0A, 0x85)}};
96  const struct {
97      crypt_uword_t    allocated;
98      crypt_uword_t    size;
99      crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(28)];
100     } NIST_P224_gX = {BYTES_TO_CRYPT_WORDS(28), BYTES_TO_CRYPT_WORDS(28),
101        {TO_CRYPT_WORD_64(0x34, 0x32, 0x80, 0xD6, 0x11, 0x5C, 0x1D, 0x21),
102         TO_CRYPT_WORD_64(0x4A, 0x03, 0xC1, 0xD3, 0x56, 0xC2, 0x11, 0x22),
103         TO_CRYPT_WORD_64(0x6B, 0xB4, 0xBF, 0x7F, 0x32, 0x13, 0x90, 0xB9),
```



```
104           TO_CRYPT_WORD_32(0xB7, 0x0E, 0x0C, 0xBD)}};
105     const struct {
106         crypt_uword_t    allocated;
107         crypt_uword_t    size;
108         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(28)];
109         } NIST_P224_gY = {BYTES_TO_CRYPT_WORDS(28), BYTES_TO_CRYPT_WORDS(28),
110            {TO_CRYPT_WORD_64(0x44, 0xD5, 0x81, 0x99, 0x85, 0x00, 0x7E, 0x34),
111             TO_CRYPT_WORD_64(0xCD, 0x43, 0x75, 0xA0, 0x5A, 0x07, 0x47, 0x64),
112             TO_CRYPT_WORD_64(0xB5, 0xF7, 0x23, 0xFB, 0x4C, 0x22, 0xDF, 0xE6),
113             TO_CRYPT_WORD_32(0xBD, 0x37, 0x63, 0x88)}};
114     const struct {
115         crypt_uword_t    allocated;
116         crypt_uword_t    size;
117         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(28)];
118         } NIST_P224_n = {BYTES_TO_CRYPT_WORDS(28), BYTES_TO_CRYPT_WORDS(28),
119            {TO_CRYPT_WORD_64(0x13, 0xDD, 0x29, 0x45, 0x5C, 0x5C, 0x2A, 0x3D),
120             TO_CRYPT_WORD_64(0xFF, 0xFF, 0x16, 0xA2, 0xE0, 0xB8, 0xF0, 0x3E),
121             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
122             TO_CRYPT_WORD_32(0xFF, 0xFF, 0xFF, 0xFF)}};
123     #define NIST_P224_h BN_ONE
124     const ECC_CURVE_DATA NIST_P224 = {
125         (bigNum)&NIST_P224_p, (bigNum)&NIST_P224_n, (bigNum)&NIST_P224_h,
126         (bigNum)&NIST_P224_a,  (bigNum)&NIST_P224_b,
127         {(bigNum)&NIST_P224_gX, (bigNum)&NIST_P224_gY, (bigNum)&BN_ONE}};
128     #endif // ECC_NIST_P224
129     #if defined ECC_NIST_P256 && ECC_NIST_P256 == YES
130     const struct {
131         crypt_uword_t    allocated;
132         crypt_uword_t    size;
133         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
134         } NIST_P256_p = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
135            {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
136             TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
137             TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
138             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01)}};
139     const struct {
140         crypt_uword_t    allocated;
141         crypt_uword_t    size;
142         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
143         } NIST_P256_a = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
144            {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC),
145             TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
146             TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
147             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01)}};
148     const struct {
149         crypt_uword_t    allocated;
150         crypt_uword_t    size;
151         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
152         } NIST_P256_b = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
153            {TO_CRYPT_WORD_64(0x3B, 0xCE, 0x3C, 0x3E, 0x27, 0xD2, 0x60, 0x4B),
154             TO_CRYPT_WORD_64(0x65, 0x1D, 0x06, 0xB0, 0xCC, 0x53, 0xB0, 0xF6),
155             TO_CRYPT_WORD_64(0xB3, 0xEB, 0xBD, 0x55, 0x76, 0x98, 0x86, 0xBC),
156             TO_CRYPT_WORD_64(0x5A, 0xC6, 0x35, 0xD8, 0xAA, 0x3A, 0x93, 0xE7)}};
157     const struct {
158         crypt_uword_t    allocated;
159         crypt_uword_t    size;
160         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
161         } NIST_P256_gX = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
162            {TO_CRYPT_WORD_64(0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96),
163             TO_CRYPT_WORD_64(0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0),
164             TO_CRYPT_WORD_64(0xF8, 0xBC, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2),
165             TO_CRYPT_WORD_64(0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47)}};
166     const struct {
167         crypt_uword_t    allocated;
168         crypt_uword_t    size;
169         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
```



```
170         } NIST_P256_gY = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
171            {TO_CRYPT_WORD_64(0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5),
172             TO_CRYPT_WORD_64(0x2B, 0xCE, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE),
173             TO_CRYPT_WORD_64(0x8E, 0xE7, 0xEB, 0x4A, 0x7C, 0x0F, 0x9E, 0x16),
174             TO_CRYPT_WORD_64(0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B)}};
175     const struct {
176         crypt_uword_t    allocated;
177         crypt_uword_t    size;
178         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
179         } NIST_P256_n = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
180            {TO_CRYPT_WORD_64(0xF3, 0xB9, 0xCA, 0xC2, 0xFC, 0x63, 0x25, 0x51),
181             TO_CRYPT_WORD_64(0xBC, 0xE6, 0xFA, 0xAD, 0xA7, 0x17, 0x9E, 0x84),
182             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
183             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00)}};
184     #define NIST_P256_h BN_ONE
185     const ECC_CURVE_DATA NIST_P256 = {
186         (bigNum)&NIST_P256_p, (bigNum)&NIST_P256_n, (bigNum)&NIST_P256_h,
187         (bigNum)&NIST_P256_a,  (bigNum)&NIST_P256_b,
188         {(bigNum)&NIST_P256_gX, (bigNum)&NIST_P256_gY, (bigNum)&BN_ONE}};
189     #endif // ECC_NIST_P256
190     #if defined ECC_NIST_P384 && ECC_NIST_P384 == YES
191     const struct {
192         crypt_uword_t    allocated;
193         crypt_uword_t    size;
194         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(48)];
195         } NIST_P384_p = {BYTES_TO_CRYPT_WORDS(48), BYTES_TO_CRYPT_WORDS(48),
196            {TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
197             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
198             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
199             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
200             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
201             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)}};
202     const struct {
203         crypt_uword_t    allocated;
204         crypt_uword_t    size;
205         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(48)];
206         } NIST_P384_a = {BYTES_TO_CRYPT_WORDS(48), BYTES_TO_CRYPT_WORDS(48),
207            {TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFC),
208             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
209             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
210             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
211             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
212             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)}};
213     const struct {
214         crypt_uword_t    allocated;
215         crypt_uword_t    size;
216         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(48)];
217         } NIST_P384_b = {BYTES_TO_CRYPT_WORDS(48), BYTES_TO_CRYPT_WORDS(48),
218            {TO_CRYPT_WORD_64(0x2A, 0x85, 0xC8, 0xED, 0xD3, 0xEC, 0x2A, 0xEF),
219             TO_CRYPT_WORD_64(0xC6, 0x56, 0x39, 0x8D, 0x8A, 0x2E, 0xD1, 0x9D),
220             TO_CRYPT_WORD_64(0x03, 0x14, 0x08, 0x8F, 0x50, 0x13, 0x87, 0x5A),
221             TO_CRYPT_WORD_64(0x18, 0x1D, 0x9C, 0x6E, 0xFE, 0x81, 0x41, 0x12),
222             TO_CRYPT_WORD_64(0x98, 0x8E, 0x05, 0x6B, 0xE3, 0xF8, 0x2D, 0x19),
223             TO_CRYPT_WORD_64(0xB3, 0x31, 0x2F, 0xA7, 0xE2, 0x3E, 0xE7, 0xE4)}};
224     const struct {
225         crypt_uword_t    allocated;
226         crypt_uword_t    size;
227         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(48)];
228         } NIST_P384_gX = {BYTES_TO_CRYPT_WORDS(48), BYTES_TO_CRYPT_WORDS(48),
229            {TO_CRYPT_WORD_64(0x3A, 0x54, 0x5E, 0x38, 0x72, 0x76, 0x0A, 0xB7),
230             TO_CRYPT_WORD_64(0x55, 0x02, 0xF2, 0x5D, 0xBF, 0x55, 0x29, 0x6C),
231             TO_CRYPT_WORD_64(0x59, 0xF7, 0x41, 0xE0, 0x82, 0x54, 0x2A, 0x38),
232             TO_CRYPT_WORD_64(0x6E, 0x1D, 0x3B, 0x62, 0x8B, 0xA7, 0x9B, 0x98),
233             TO_CRYPT_WORD_64(0x8E, 0xB1, 0xC7, 0x1E, 0xF3, 0x20, 0xAD, 0x74),
234             TO_CRYPT_WORD_64(0xAA, 0x87, 0xCA, 0x22, 0xBE, 0x8B, 0x05, 0x37)}};
235     const struct {
```



```
236         crypt_uword_t    allocated;
237         crypt_uword_t    size;
238         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(48)];
239         } NIST_P384_gY = {BYTES_TO_CRYPT_WORDS(48), BYTES_TO_CRYPT_WORDS(48),
240            {TO_CRYPT_WORD_64(0x7A, 0x43, 0x1D, 0x7C, 0x90, 0xEA, 0x0E, 0x5F),
241             TO_CRYPT_WORD_64(0x0A, 0x60, 0xB1, 0xCE, 0x1D, 0x7E, 0x81, 0x9D),
242             TO_CRYPT_WORD_64(0xE9, 0xDA, 0x31, 0x13, 0xB5, 0xF0, 0xB8, 0xC0),
243             TO_CRYPT_WORD_64(0xF8, 0xF4, 0x1D, 0xBD, 0x28, 0x9A, 0x14, 0x7C),
244             TO_CRYPT_WORD_64(0x5D, 0x9E, 0x98, 0xBF, 0x92, 0x92, 0xDC, 0x29),
245             TO_CRYPT_WORD_64(0x36, 0x17, 0xDE, 0x4A, 0x96, 0x26, 0x2C, 0x6F)}};
246     const struct {
247         crypt_uword_t    allocated;
248         crypt_uword_t    size;
249         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(48)];
250         } NIST_P384_n = {BYTES_TO_CRYPT_WORDS(48), BYTES_TO_CRYPT_WORDS(48),
251            {TO_CRYPT_WORD_64(0xEC, 0xEC, 0x19, 0x6A, 0xCC, 0xC5, 0x29, 0x73),
252             TO_CRYPT_WORD_64(0x58, 0x1A, 0x0D, 0xB2, 0x48, 0xB0, 0xA7, 0x7A),
253             TO_CRYPT_WORD_64(0xC7, 0x63, 0x4D, 0x81, 0xF4, 0x37, 0x2D, 0xDF),
254             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
255             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
256             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)}};
257     #define NIST_P384_h BN_ONE
258     const ECC_CURVE_DATA NIST_P384 = {
259         (bigNum)&NIST_P384_p, (bigNum)&NIST_P384_n, (bigNum)&NIST_P384_h,
260         (bigNum)&NIST_P384_a,  (bigNum)&NIST_P384_b,
261         {(bigNum)&NIST_P384_gX, (bigNum)&NIST_P384_gY, (bigNum)&BN_ONE}};
262     #endif // ECC_NIST_P384
263     #if defined ECC_NIST_P521 && ECC_NIST_P521 == YES
264     const struct {
265         crypt_uword_t    allocated;
266         crypt_uword_t    size;
267         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(66)];
268         } NIST_P521_p = {BYTES_TO_CRYPT_WORDS(66), BYTES_TO_CRYPT_WORDS(66),
269            {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
270             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
271             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
272             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
273             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
274             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
275             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
276             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
277             TO_CRYPT_WORD_32(0x00, 0x00, 0x01, 0xFF)}};
278     const struct {
279         crypt_uword_t    allocated;
280         crypt_uword_t    size;
281         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(66)];
282         } NIST_P521_a = {BYTES_TO_CRYPT_WORDS(66), BYTES_TO_CRYPT_WORDS(66),
283            {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC),
284             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
285             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
286             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
287             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
288             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
289             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
290             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
291             TO_CRYPT_WORD_32(0x00, 0x00, 0x01, 0xFF)}};
292     const struct {
293         crypt_uword_t    allocated;
294         crypt_uword_t    size;
295         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(66)];
296         } NIST_P521_b = {BYTES_TO_CRYPT_WORDS(66), BYTES_TO_CRYPT_WORDS(66),
297            {TO_CRYPT_WORD_64(0xEF, 0x45, 0x1F, 0xD4, 0x6B, 0x50, 0x3F, 0x00),
298             TO_CRYPT_WORD_64(0x35, 0x73, 0xDF, 0x88, 0x3D, 0x2C, 0x34, 0xF1),
299             TO_CRYPT_WORD_64(0x16, 0x52, 0xC0, 0xBD, 0x3B, 0xB1, 0xBF, 0x07),
300             TO_CRYPT_WORD_64(0x56, 0x19, 0x39, 0x51, 0xEC, 0x7E, 0x93, 0x7B),
301             TO_CRYPT_WORD_64(0xB8, 0xB4, 0x89, 0x91, 0x8E, 0xF1, 0x09, 0xE1),
```

```
302           TO_CRYPT_WORD_64(0xA2, 0xDA, 0x72, 0x5B, 0x99, 0xB3, 0x15, 0xF3),
303           TO_CRYPT_WORD_64(0x92, 0x9A, 0x21, 0xA0, 0xB6, 0x85, 0x40, 0xEE),
304           TO_CRYPT_WORD_64(0x95, 0x3E, 0xB9, 0x61, 0x8E, 0x1C, 0x9A, 0x1F),
305           TO_CRYPT_WORD_32(0x00, 0x00, 0x00, 0x51)}};
306   const struct {
307       crypt_uword_t    allocated;
308       crypt_uword_t    size;
309       crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(66)];
310       } NIST_P521_gX = {BYTES_TO_CRYPT_WORDS(66), BYTES_TO_CRYPT_WORDS(66),
311          {TO_CRYPT_WORD_64(0xF9, 0x7E, 0x7E, 0x31, 0xC2, 0xE5, 0xBD, 0x66),
312           TO_CRYPT_WORD_64(0x33, 0x48, 0xB3, 0xC1, 0x85, 0x6A, 0x42, 0x9B),
313           TO_CRYPT_WORD_64(0xFE, 0x1D, 0xC1, 0x27, 0xA2, 0xFF, 0xA8, 0xDE),
314           TO_CRYPT_WORD_64(0xA1, 0x4B, 0x5E, 0x77, 0xEF, 0xE7, 0x59, 0x28),
315           TO_CRYPT_WORD_64(0xF8, 0x28, 0xAF, 0x60, 0x6B, 0x4D, 0x3D, 0xBA),
316           TO_CRYPT_WORD_64(0x9C, 0x64, 0x81, 0x39, 0x05, 0x3F, 0xB5, 0x21),
317           TO_CRYPT_WORD_64(0x9E, 0x3E, 0xCB, 0x66, 0x23, 0x95, 0xB4, 0x42),
318           TO_CRYPT_WORD_64(0x85, 0x8E, 0x06, 0xB7, 0x04, 0x04, 0xE9, 0xCD),
319           TO_CRYPT_WORD_32(0x00, 0x00, 0x00, 0xC6)}};
320   const struct {
321       crypt_uword_t    allocated;
322       crypt_uword_t    size;
323       crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(66)];
324       } NIST_P521_gY = {BYTES_TO_CRYPT_WORDS(66), BYTES_TO_CRYPT_WORDS(66),
325          {TO_CRYPT_WORD_64(0x88, 0xBE, 0x94, 0x76, 0x9F, 0xD1, 0x66, 0x50),
326           TO_CRYPT_WORD_64(0x35, 0x3C, 0x70, 0x86, 0xA2, 0x72, 0xC2, 0x40),
327           TO_CRYPT_WORD_64(0xC5, 0x50, 0xB9, 0x01, 0x3F, 0xAD, 0x07, 0x61),
328           TO_CRYPT_WORD_64(0x97, 0xEE, 0x72, 0x99, 0x5E, 0xF4, 0x26, 0x40),
329           TO_CRYPT_WORD_64(0x17, 0xAF, 0xBD, 0x17, 0x27, 0x3E, 0x66, 0x2C),
330           TO_CRYPT_WORD_64(0x98, 0xF5, 0x44, 0x49, 0x57, 0x9B, 0x44, 0x68),
331           TO_CRYPT_WORD_64(0x5C, 0x8A, 0x5F, 0xB4, 0x2C, 0x7D, 0x1B, 0xD9),
332           TO_CRYPT_WORD_64(0x39, 0x29, 0x6A, 0x78, 0x9A, 0x3B, 0xC0, 0x04),
333           TO_CRYPT_WORD_32(0x00, 0x00, 0x01, 0x18)}};
334   const struct {
335       crypt_uword_t    allocated;
336       crypt_uword_t    size;
337       crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(66)];
338       } NIST_P521_n = {BYTES_TO_CRYPT_WORDS(66), BYTES_TO_CRYPT_WORDS(66),
339          {TO_CRYPT_WORD_64(0xBB, 0x6F, 0xB7, 0x1E, 0x91, 0x38, 0x64, 0x09),
340           TO_CRYPT_WORD_64(0x3B, 0xB5, 0xC9, 0xB8, 0x89, 0x9C, 0x47, 0xAE),
341           TO_CRYPT_WORD_64(0x7F, 0xCC, 0x01, 0x48, 0xF7, 0x09, 0xA5, 0xD0),
342           TO_CRYPT_WORD_64(0x51, 0x86, 0x87, 0x83, 0xBF, 0x2F, 0x96, 0x6B),
343           TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFA),
344           TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
345           TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
346           TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
347           TO_CRYPT_WORD_32(0x00, 0x00, 0x01, 0xFF)}};
348   #define NIST_P521_h BN_ONE
349   const ECC_CURVE_DATA NIST_P521 = {
350       (bigNum)&NIST_P521_p, (bigNum)&NIST_P521_n, (bigNum)&NIST_P521_h,
351       (bigNum)&NIST_P521_a,  (bigNum)&NIST_P521_b,
352       {(bigNum)&NIST_P521_gX, (bigNum)&NIST_P521_gY, (bigNum)&BN_ONE}};
353   #endif // ECC_NIST_P521
354   #if defined ECC_BN_P256 && ECC_BN_P256 == YES
355   const struct {
356       crypt_uword_t    allocated;
357       crypt_uword_t    size;
358       crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
359       } BN_P256_p = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
360          {TO_CRYPT_WORD_64(0xD3, 0x29, 0x2D, 0xDB, 0xAE, 0xD3, 0x30, 0x13),
361           TO_CRYPT_WORD_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x98, 0x0A, 0x82),
362           TO_CRYPT_WORD_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9F),
363           TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD)}};
364   #define BN_P256_a BN_ZERO
365   const struct {
366       crypt_uword_t    allocated;
367       crypt_uword_t    size;
```

```
368         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(1)];
369         } BN_P256_b = {BYTES_TO_CRYPT_WORDS(1), BYTES_TO_CRYPT_WORDS(1),
370            {TO_CRYPT_WORD_32(0x00, 0x00, 0x00, 0x03)}};
371     #define BN_P256_gX BN_ONE
372     const struct {
373         crypt_uword_t    allocated;
374         crypt_uword_t    size;
375         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(1)];
376         } BN_P256_gY = {BYTES_TO_CRYPT_WORDS(1), BYTES_TO_CRYPT_WORDS(1),
377            {TO_CRYPT_WORD_32(0x00, 0x00, 0x00, 0x02)}};
378     const struct {
379         crypt_uword_t    allocated;
380         crypt_uword_t    size;
381         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
382         } BN_P256_n = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
383            {TO_CRYPT_WORD_64(0xF6, 0x2D, 0x53, 0x6C, 0xD1, 0x0B, 0x50, 0x0D),
384             TO_CRYPT_WORD_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x99, 0x92, 0x1A),
385             TO_CRYPT_WORD_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9E),
386             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD)}};
387     #define BN_P256_h BN_ONE
388     const ECC_CURVE_DATA BN_P256 = {
389         (bigNum)&BN_P256_p, (bigNum)&BN_P256_n, (bigNum)&BN_P256_h,
390         (bigNum)&BN_P256_a,  (bigNum)&BN_P256_b,
391         {(bigNum)&BN_P256_gX, (bigNum)&BN_P256_gY, (bigNum)&BN_ONE}};
392     #endif // ECC_BN_P256
393     #if defined ECC_BN_P638 && ECC_BN_P638 == YES
394     const struct {
395         crypt_uword_t    allocated;
396         crypt_uword_t    size;
397         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(80)];
398         } BN_P638_p = {BYTES_TO_CRYPT_WORDS(80), BYTES_TO_CRYPT_WORDS(80),
399            {TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67),
400             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
401             TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
402             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
403             TO_CRYPT_WORD_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
404             TO_CRYPT_WORD_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
405             TO_CRYPT_WORD_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
406             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
407             TO_CRYPT_WORD_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
408             TO_CRYPT_WORD_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D)}};
409     #define BN_P638_a BN_ZERO
410     const struct {
411         crypt_uword_t    allocated;
412         crypt_uword_t    size;
413         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(2)];
414         } BN_P638_b = {BYTES_TO_CRYPT_WORDS(2), BYTES_TO_CRYPT_WORDS(2),
415            {TO_CRYPT_WORD_32(0x00, 0x00, 0x01, 0x01)}};
416     const struct {
417         crypt_uword_t    allocated;
418         crypt_uword_t    size;
419         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(80)];
420         } BN_P638_gX = {BYTES_TO_CRYPT_WORDS(80), BYTES_TO_CRYPT_WORDS(80),
421            {TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66),
422             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
423             TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
424             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
425             TO_CRYPT_WORD_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
426             TO_CRYPT_WORD_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
427             TO_CRYPT_WORD_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
428             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
429             TO_CRYPT_WORD_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
430             TO_CRYPT_WORD_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D)}};
431     const struct {
432         crypt_uword_t    allocated;
433         crypt_uword_t    size;
```

```
434         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(1)];
435         } BN_P638_gY = {BYTES_TO_CRYPT_WORDS(1), BYTES_TO_CRYPT_WORDS(1),
436            {TO_CRYPT_WORD_32(0x00, 0x00, 0x00, 0x10)}};
437     const struct {
438         crypt_uword_t    allocated;
439         crypt_uword_t    size;
440         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(80)];
441         } BN_P638_n = {BYTES_TO_CRYPT_WORDS(80), BYTES_TO_CRYPT_WORDS(80),
442            {TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61),
443             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xA0),
444             TO_CRYPT_WORD_64(0x00, 0x00, 0x00, 0x49, 0x80, 0x01, 0x54, 0xD9),
445             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xF5, 0x4F, 0xFF, 0xF4, 0xEA, 0xC0),
446             TO_CRYPT_WORD_64(0x60, 0x00, 0x86, 0x55, 0x00, 0x21, 0xE5, 0x55),
447             TO_CRYPT_WORD_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
448             TO_CRYPT_WORD_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
449             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
450             TO_CRYPT_WORD_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
451             TO_CRYPT_WORD_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D)}};
452     #define BN_P638_h BN_ONE
453     const ECC_CURVE_DATA BN_P638 = {
454         (bigNum)&BN_P638_p, (bigNum)&BN_P638_n, (bigNum)&BN_P638_h,
455         (bigNum)&BN_P638_a,  (bigNum)&BN_P638_b,
456         {(bigNum)&BN_P638_gX, (bigNum)&BN_P638_gY, (bigNum)&BN_ONE}};
457     #endif // ECC_BN_P638
458     #if defined ECC_SM2_P256 && ECC_SM2_P256 == YES
459     const struct {
460         crypt_uword_t    allocated;
461         crypt_uword_t    size;
462         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
463         } SM2_P256_p = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
464            {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
465             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
466             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
467             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF)}};
468     const struct {
469         crypt_uword_t    allocated;
470         crypt_uword_t    size;
471         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
472         } SM2_P256_a = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
473            {TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC),
474             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
475             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
476             TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF)}};
477     const struct {
478         crypt_uword_t    allocated;
479         crypt_uword_t    size;
480         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
481         } SM2_P256_b = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
482            {TO_CRYPT_WORD_64(0xDD, 0xBC, 0xBD, 0x41, 0x4D, 0x94, 0x0E, 0x93),
483             TO_CRYPT_WORD_64(0xF3, 0x97, 0x89, 0xF5, 0x15, 0xAB, 0x8F, 0x92),
484             TO_CRYPT_WORD_64(0x4D, 0x5A, 0x9E, 0x4B, 0xCF, 0x65, 0x09, 0xA7),
485             TO_CRYPT_WORD_64(0x28, 0xE9, 0xFA, 0x9E, 0x9D, 0x9F, 0x5E, 0x34)}};
486     const struct {
487         crypt_uword_t    allocated;
488         crypt_uword_t    size;
489         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
490         } SM2_P256_gX = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
491            {TO_CRYPT_WORD_64(0x71, 0x5A, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7),
492             TO_CRYPT_WORD_64(0x8F, 0xE3, 0x0B, 0xBF, 0xF2, 0x66, 0x0B, 0xE1),
493             TO_CRYPT_WORD_64(0x5F, 0x99, 0x04, 0x46, 0x6A, 0x39, 0xC9, 0x94),
494             TO_CRYPT_WORD_64(0x32, 0xC4, 0xAE, 0x2C, 0x1F, 0x19, 0x81, 0x19)}};
495     const struct {
496         crypt_uword_t    allocated;
497         crypt_uword_t    size;
498         crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
499         } SM2_P256_gY = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
```

```
500          {TO_CRYPT_WORD_64(0x02, 0xDF, 0x32, 0xE5, 0x21, 0x39, 0xF0, 0xA0),
501           TO_CRYPT_WORD_64(0xD0, 0xA9, 0x87, 0x7C, 0xC6, 0x2A, 0x47, 0x40),
502           TO_CRYPT_WORD_64(0x59, 0xBD, 0xCE, 0xE3, 0x6B, 0x69, 0x21, 0x53),
503           TO_CRYPT_WORD_64(0xBC, 0x37, 0x36, 0xA2, 0xF4, 0xF6, 0x77, 0x9C)}};
504    const struct {
505        crypt_uword_t    allocated;
506        crypt_uword_t    size;
507        crypt_uword_t    d[BYTES_TO_CRYPT_WORDS(32)];
508        } SM2_P256_n = {BYTES_TO_CRYPT_WORDS(32), BYTES_TO_CRYPT_WORDS(32),
509           {TO_CRYPT_WORD_64(0x53, 0xBB, 0xF4, 0x09, 0x39, 0xD5, 0x41, 0x23),
510            TO_CRYPT_WORD_64(0x72, 0x03, 0xDF, 0x6B, 0x21, 0xC6, 0x05, 0x2B),
511            TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
512            TO_CRYPT_WORD_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF)}};
513    #define SM2_P256_h BN_ONE
514    const ECC_CURVE_DATA SM2_P256 = {
515        (bigNum)&SM2_P256_p, (bigNum)&SM2_P256_n, (bigNum)&SM2_P256_h,
516        (bigNum)&SM2_P256_a,  (bigNum)&SM2_P256_b,
517        {(bigNum)&SM2_P256_gX, (bigNum)&SM2_P256_gY, (bigNum)&BN_ONE}};
518    #endif // ECC_SM2_P256
519    #define comma
520    const ECC_CURVE   eccCurves[] = {
521    #if defined ECC_NIST_P192 && ECC_NIST_P192 == YES
522    comma
523    {TPM_ECC_NIST_P192,
524    192,
525    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
526    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
527    &NIST_P192
528    CURVE_NAME("NIST_P192")}
529    #   undef comma
530    #   define comma ,
531    #endif // ECC_NIST_P192
532    #if defined ECC_NIST_P224 && ECC_NIST_P224 == YES
533    comma
534    {TPM_ECC_NIST_P224,
535    224,
536    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
537    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
538    &NIST_P224
539    CURVE_NAME("NIST_P224")}
540    #   undef comma
541    #   define comma ,
542    #endif // ECC_NIST_P224
543    #if defined ECC_NIST_P256 && ECC_NIST_P256 == YES
544    comma
545    {TPM_ECC_NIST_P256,
546    256,
547    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
548    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
549    &NIST_P256
550    CURVE_NAME("NIST_P256")}
551    #   undef comma
552    #   define comma ,
553    #endif // ECC_NIST_P256
554    #if defined ECC_NIST_P384 && ECC_NIST_P384 == YES
555    comma
556    {TPM_ECC_NIST_P384,
557    384,
558    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA384_VALUE}}},
559    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
560    &NIST_P384
561    CURVE_NAME("NIST_P384")}
562    #   undef comma
563    #   define comma ,
564    #endif // ECC_NIST_P384
565    #if defined ECC_NIST_P521 && ECC_NIST_P521 == YES
```



```
566     comma
567     {TPM_ECC_NIST_P521,
568     521,
569     {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA512_VALUE}}},
570     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
571     &NIST_P521
572     CURVE_NAME("NIST_P521")}
573     #   undef comma
574     #   define comma ,
575     #endif // ECC_NIST_P521
576     #if defined ECC_BN_P256 && ECC_BN_P256 == YES
577     comma
578     {TPM_ECC_BN_P256,
579     256,
580     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
581     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
582     &BN_P256
583     CURVE_NAME("BN_P256")}
584     #   undef comma
585     #   define comma ,
586     #endif // ECC_BN_P256
587     #if defined ECC_BN_P638 && ECC_BN_P638 == YES
588     comma
589     {TPM_ECC_BN_P638,
590     638,
591     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
592     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
593     &BN_P638
594     CURVE_NAME("BN_P638")}
595     #   undef comma
596     #   define comma ,
597     #endif // ECC_BN_P638
598     #if defined ECC_SM2_P256 && ECC_SM2_P256 == YES
599     comma
600     {TPM_ECC_SM2_P256,
601     256,
602     {ALG_KDF1_SP800_56A_VALUE, {{ALG_SM3_256_VALUE}}},
603     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
604     &SM2_P256
605     CURVE_NAME("SM2_P256")}
606     #   undef comma
607     #   define comma ,
608     #endif // ECC_SM2_P256
609     };
610     #endif // TPM_ALG_ECC
```

10.2.4 BnMath.c

10.2.4.1 Introduction

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. When operating on big numbers, the data format is changed for easier manipulation. The format is native words in little-endian format. As the magnitude of the number decreases, the length of the array containing the number decreases but the starting address doesn't change.

This functions in this file perform simple operations on these big numbers. Only the more complex operations are passed to the underlying support library. Although the support library would have most of these functions, the interface code to convert the format for the values is greater than the size of the code to implement the functions here. So, rather than incur the overhead of conversion, they are done here.

If an implementer would prefer, the underlying library can be used simply by making code substitutions here.

NOTE: There is an intention to continue to augment these functions so that there would be no need to use an external big number library.

Many of these functions have no error returns and will always return TRUE. This is to allow them to be used in **guarded** sequences. That is: OK = OK || *BnSomething*(s); where the BnSomething() function should not be called if OK isn't true.

10.2.4.2 Includes

```
1   #include "Tpm.h"
```

A constant value of zero as a stand in for NULL *bigNum* values

```
2   const bignum_t   BnConstZero = {1, 0, {0}};
```

10.2.4.3 Functions

10.2.4.3.1 AddSame()

Adds two values that are the same size. This function allows *result* to be the same as either of the addends. This is a nice function to put into assembly because handling the carry for multi-precision stuff is not as easy in C (unless there is a REALLY smart compiler). It would be nice if there were idioms in a language that a compiler could recognize what is going on and optimize loops like this.

Return Value	Meaning
0	no carry out
1	carry out

```
3   static BOOL
4   AddSame(
5       crypt_uword_t           *result,
6       const crypt_uword_t     *op1,
7       const crypt_uword_t     *op2,
8       int                      count
9       )
10  {
11      int         carry = 0;
12      int         i;
13      for(i = 0; i < count; i++)
14      {
15          crypt_uword_t        a = op1[i];
16          crypt_uword_t        sum = a + op2[i];
17          result[i] = sum + carry;
18          // generate a carry if the sum is less than either of the inputs
19          // propagate a carry if there was a carry and the sum + carry is zero
20          // do this using bit operations rather than logical operations so that
21          // the time is about the same.
22          //             propagate term      | generate term
23          carry = ((result[i] == 0) & carry) | (sum < a);
24      }
25      return carry;
26  }
```

10.2.4.3.2 CarryProp()

Propagate a carry

```
27  static int
28  CarryProp(
29      crypt_uword_t           *result,
30      const crypt_uword_t     *op,
31      int                      count,
32      int                      carry
33      )
34  {
35      for(; count; count--)
36          carry = ((*result++ = *op++ + carry) == 0) & carry;
37      return carry;
38  }
39  static void
40  CarryResolve(
41      bigNum          result,
42      int             stop,
43      int             carry
44      )
45  {
46      if(carry)
47      {
48          pAssert((unsigned)stop < result->allocated);
49          result->d[stop++] = 1;
50      }
51      BnSetTop(result, stop);
52  }
```

10.2.4.3.3 BnAdd()

Function to add two *bigNum* values. Always returns TRUEF

```
53  LIB_EXPORT BOOL
54  BnAdd(
55      bigNum           result,
56      bigConst         op1,
57      bigConst         op2
58      )
59  {
60      crypt_uword_t    stop;
61      int              carry;
62      const bignum_t   *n1 = op1;
63      const bignum_t   *n2 = op2;
64  //
65      if(n2->size > n1->size)
66      {
67          n1 = op2;
68          n2 = op1;
69      }
70      pAssert(result->allocated >= n1->size);
71      stop = MIN(n1->size, n2->allocated);
72      carry = AddSame(result->d, n1->d, n2->d, stop);
73      if(n1->size > stop)
74          carry = CarryProp(&result->d[stop], &n1->d[stop], n1->size - stop, carry);
75      CarryResolve(result, n1->size, carry);
76      return TRUE;
77  }
```

10.2.4.3.4 BnAddWord()

Adds a word value to a *bigNum*.

```
78  LIB_EXPORT BOOL
79  BnAddWord(
```

```
80      bigNum          result,
81      bigConst        op,
82      crypt_uword_t   word
83      )
84  {
85      int             carry;
86  //
87      carry = (result->d[0] = op->d[0] + word) < word;
88      carry = CarryProp(&result->d[1], &op->d[1], op->size - 1, carry);
89      CarryResolve(result, op->size, carry);
90      return TRUE;
91  }
```

10.2.4.3.5 SubSame()

Subtract two values that have the same size.

```
92  static int
93  SubSame(
94      crypt_uword_t           *result,
95      const crypt_uword_t     *op1,
96      const crypt_uword_t     *op2,
97      int                      count
98      )
99  {
100     int                 borrow = 0;
101     int                 i;
102     for(i = 0; i < count; i++)
103     {
104         crypt_uword_t   a = op1[i];
105         crypt_uword_t   diff = a - op2[i];
106         result[i] = diff - borrow;
107         //       generate   |      propagate
108         borrow = (diff > a) | ((diff == 0) & borrow);
109     }
110     return borrow;
111 }
```

10.2.4.3.6 BorrowProp()

This propagates a borrow. If borrow is true when the end of the array is reached, then it means that op2 was larger than op1 and we don't handle that case so an assert is generated. This design choice was made because our only *bigNum* computations are on large positive numbers (primes) or on fields. Propagate a borrow.

```
112 static int
113 BorrowProp(
114     crypt_uword_t           *result,
115     const crypt_uword_t     *op,
116     int                      size,
117     int                      borrow
118     )
119 {
120     for(; size > 0; size--)
121         borrow = ((*result++ = *op++ - borrow) == MAX_CRYPT_UWORD) && borrow;
122     return borrow;
123 }
```

10.2.4.3.7 BnSub()

Function to do subtraction of result = op1 - op2 when op1 is greater than op2. If it isn't then a fault is generated.

```
124  LIB_EXPORT BOOL
125  BnSub(
126      bigNum           result,
127      bigConst         op1,
128      bigConst         op2
129      )
130  {
131      int             borrow;
132      crypt_uword_t   stop = MIN(op1->size, op2->allocated);
133  //
134      // Make sure that op2 is not obviously larger than op1
135      pAssert(op1->size >= op2->size);
136      borrow = SubSame(result->d, op1->d, op2->d, stop);
137      if(op1->size > stop)
138          borrow = BorrowProp(&result->d[stop], &op1->d[stop], op1->size - stop,
139                              borrow);
140      pAssert(!borrow);
141      BnSetTop(result, op1->size);
142      return TRUE;
143  }
```

10.2.4.3.8 BnSubWord()

Subtract a word value from a *bigNum*.

```
144  LIB_EXPORT BOOL
145  BnSubWord(
146      bigNum           result,
147      bigConst     op,
148      crypt_uword_t    word
149      )
150  {
151      int             borrow;
152  //
153      pAssert(op->size > 1 || word <= op->d[0]);
154      borrow = word > op->d[0];
155      result->d[0] = op->d[0] - word;
156      borrow = BorrowProp(&result->d[1], &op->d[1], op->size - 1, borrow);
157      pAssert(!borrow);
158      BnSetTop(result, op->size);
159      return TRUE;
160  }
```

10.2.4.3.9 BnUnsignedCmp()

This function performs a comparison of op1 to op2. The compare is approximately constant time if the size of the values used in the compare is consistent across calls (from the same line in the calling code).

Return Value	Meaning
< 0	op1 is less than op2
0	op1 is equal to op2
> 0	op1 is greater than op2

```
161  LIB_EXPORT int
```



```
162 BnUnsignedCmp(
163     bigConst              op1,
164     bigConst              op2
165     )
166 {
167     int             retVal;
168     int             diff;
169     int              i;
170 //
171     pAssert((op1 != NULL) && (op2 != NULL));
172     retVal = op1->size - op2->size;
173     if(retVal == 0)
174     {
175         for(i = (int)(op1->size - 1); i >= 0; i--)
176         {
177             diff = (op1->d[i] < op2->d[i]) ? -1 : (op1->d[i] != op2->d[i]);
178             retVal = retVal == 0 ? diff : retVal;
179         }
180     }
181     else
182         retVal = (retVal < 0) ? -1 : 1;
183     return retVal;
184 }
```

10.2.4.3.10 BnUnsignedCmpWord()

Compare a *bigNum* to a crypt_uword_t. -1 op1 is less that word 0 op1 is equal to word 1 op1 is greater than word

```
185 LIB_EXPORT int
186 BnUnsignedCmpWord(
187     bigConst             op1,
188     crypt_uword_t        word
189     )
190 {
191     if(op1->size > 1)
192         return 1;
193     else if(op1->size == 1)
194         return (op1->d[0] < word) ? -1 : (op1->d[0] > word);
195     else // op1 is zero
196         // equal if word is zero
197         return (word == 0) ? 0 : -1;
198 }
```

10.2.4.3.11 BnModWord()

Find the modulus of a big number when the modulus is a word value

```
199 LIB_EXPORT crypt_word_t
200 BnModWord(
201     bigConst         numerator,
202     crypt_word_t     modulus
203     )
204 {
205     BN_MAX(remainder);
206     BN_VAR(mod, RADIX_BITS);
207 //
208     mod->d[0] = modulus;
209     mod->size = (modulus != 0);
210     BnDiv(NULL, remainder, numerator, mod);
211     return remainder->d[0];
212 }
```


10.2.4.3.12 Msb()

Returns the bit number of the most significant bit of a crypt_uword_t. The number for the least significant bit of any *bigNum* value is 0. The maximum return value is RADIX_BITS - 1,

Return Value	Meaning
-1	the word was zero
n	the bit number of the most significant bit in the word

```
213  LIB_EXPORT int
214  Msb(
215      crypt_uword_t           word
216      )
217  {
218      int             retVal = -1;
219  //
220  #if RADIX_BITS == 64
221      if(word & 0xffffffff00000000) { retVal += 32; word >>= 32; }
222  #endif
223      if(word & 0xffff0000) { retVal += 16; word >>= 16; }
224      if(word & 0x0000ff00) { retVal += 8; word >>= 8; }
225      if(word & 0x000000f0) { retVal += 4; word >>= 4; }
226      if(word & 0x0000000c) { retVal += 2; word >>= 2; }
227      if(word & 0x00000002) { retVal += 1; word >>= 1; }
228      return retVal + (int)word;
229  }
```

10.2.4.3.13 BnMsb()

Returns the number of the MSb(). Returns a negative number if the value is zero or *bn* is NULL.

```
230  LIB_EXPORT int
231  BnMsb(
232      bigConst            bn
233      )
234  {
235      // If the value is NULL, or the size is zero then treat as zero and return -1
236      if(bn != NULL && bn->size > 0)
237      {
238          int         retVal = Msb(bn->d[bn->size - 1]);
239          retVal += (bn->size - 1) * RADIX_BITS;
240          return retVal;
241      }
242      else
243          return -1;
244  }
```

10.2.4.3.14 BnSizeInBits()

Returns the number of bits required to hold a number.

```
245  LIB_EXPORT unsigned
246  BnSizeInBits(
247      bigConst                n
248      )
249  {
250      return BnMsb(n) + 1;
251  }
```

10.2.4.3.15 BnSetWord()

Change the value of a bignum_t to a word value.

```
252  LIB_EXPORT bigNum
253  BnSetWord(
254      bigNum              n,
255      crypt_uword_t       w
256      )
257  {
258      if(n != NULL)
259      {
260          pAssert(n->allocated > 1);
261          n->d[0] = w;
262          BnSetTop(n, (w != 0) ? 1 : 0);
263      }
264      return n;
265  }
```

10.2.4.3.16 BnSetBit()

SET a bit in a *bigNum*. Bit 0 is the least-significant bit in the 0th digit_t. The function always return TRUE

```
266  LIB_EXPORT BOOL
267  BnSetBit(
268      bigNum          bn,         // IN/OUT: big number to modify
269      unsigned int    bitNum      // IN: Bit number to SET
270      )
271  {
272      crypt_uword_t           offset = bitNum / RADIX_BITS;
273      pAssert(bn->allocated * RADIX_BITS >= bitNum);
274      // Grow the number if necessary to set the bit.
275      while(bn->size <= offset)
276          bn->d[bn->size++] = 0;
277      bn->d[offset] |= (1 << RADIX_MOD(bitNum));
278      return TRUE;
279  }
```

10.2.4.3.17 BnTestBit()

Check to see if a bit is SET in a bignum_t. The 0th bit is the LSb() of d[0] If a bit is outside the range of the number, it returns FALSE

Return Value	Meaning
TRUE	the bit is set
FALSE	the bit is not set or the number is out of range

```
280  LIB_EXPORT BOOL
281  BnTestBit(
282      bigNum              bn,         // IN: number to check
283      unsigned int        bitNum      // IN: bit to test
284      )
285  {
286      crypt_uword_t        offset = RADIX_DIV(bitNum);
287  //
288      if(bn->size > offset)
289          return ((bn->d[offset] & (((crypt_uword_t)1) << RADIX_MOD(bitNum))) != 0);
290      else
291          return FALSE;
292  }
```

10.2.4.3.18 BnMaskBits()

Function to mask off high order bits of a big number. The returned value will have no more than *maskBit* bits set.

NOTE: There is a requirement that unused words of a bignum_t are set to zero.

Return Value	Meaning
TRUE	result masked
FALSE	the input was not as large as the mask

```
293  LIB_EXPORT BOOL
294  BnMaskBits(
295      bigNum           bn,         // IN/OUT: number to mask
296      crypt_uword_t    maskBit     // IN: the bit number for the mask.
297      )
298  {
299      crypt_uword_t    finalSize;
300      BOOL             retVal;
301      finalSize = BITS_TO_CRYPT_WORDS(maskBit);
302      retVal = (finalSize <= bn->allocated);
303      if(retVal && (finalSize > 0))
304      {
305          crypt_uword_t   mask;
306          mask = ~((crypt_uword_t)0) >> RADIX_MOD(maskBit);
307          bn->d[finalSize - 1] &= mask;
308      }
309      BnSetTop(bn, finalSize);
310      return retVal;
311  }
```

10.2.4.3.19 BnShiftRight()

Function will shift a *bigNum* to the right by the *shiftAmount*

```
312  LIB_EXPORT BOOL
313  BnShiftRight(
314      bigNum           result,
315      bigConst         toShift,
316      uint32_t         shiftAmount
317      )
318  {
319      uint32_t         offset = (shiftAmount >> RADIX_LOG2);
320      uint32_t         i;
321      uint32_t         shiftIn;
322      crypt_uword_t    finalSize;
323  //
324      shiftAmount = shiftAmount & RADIX_MASK;
325      shiftIn = RADIX_BITS - shiftAmount;
326      // The end size is toShift->size - offset less one additional
327      // word if the shiftAmount would make the upper word == 0
328      if(toShift->size > offset)
329      {
330          finalSize = toShift->size - offset;
331          finalSize -= (toShift->d[toShift->size - 1] >> shiftAmount) == 0 ? 1 : 0;
332      }
333      else
334          finalSize = 0;
335      pAssert(finalSize <= result->allocated);
336      if(finalSize != 0)
337      {
```

```
338         for(i = 0; i < finalSize; i++)
339         {
340             result->d[i] = (toShift->d[i + offset] >> shiftAmount)
341                 | (toShift->d[i + offset + 1] << shiftIn);
342         }
343         if(offset == 0)
344             result->d[i] = toShift->d[i] >> shiftAmount;
345     }
346     BnSetTop(result, finalSize);
347     return TRUE;
348 }
```

10.2.4.3.20 BnGetRandomBits()

```
349 LIB_EXPORT BOOL
350 BnGetRandomBits(
351     bigNum           n,
352     size_t           bits,
353     RAND_STATE      *rand
354     )
355 {
356     n->size = BITS_TO_CRYPT_WORDS(bits);
357     if(n->size > n->allocated)
358         n->size = n->allocated;
359     DRBG_Generate(rand, (BYTE *)n->d, (UINT16)(n->size * RADIX_BYTES));
360     BnMaskBits(n, bits);
361     return TRUE;
362 }
```

10.2.4.3.21 BnGenerateRandomInRange()

Function to generate a random number r in the range 1 <= r < limit. The function gets a random number of bits that is the size of limit. There is some some probability that the returned number is going to be greater than or equal to the limit. If it is, try again. There is no more than 50% chance that the next number is also greater, so try again. We keep trying until we get a value that meets the criteria. Since limit is very often a number with a LOT of high order ones, this rarely would need a second try.

```
363 LIB_EXPORT BOOL
364 BnGenerateRandomInRange(
365     bigNum           dest,
366     bigConst         limit,
367     RAND_STATE      *rand
368     )
369 {
370     size_t   bits = BnSizeInBits(limit);
371 //
372     if(bits < 2)
373     {
374         BnSetWord(dest, 0);
375         return FALSE;
376     }
377     else
378     {
379         do
380         {
381             BnGetRandomBits(dest, bits, rand);
382         } while(BnEqualZero(dest) || BnUnsignedCmp(dest, limit) >= 0);
383     }
384     return TRUE;
385 }
```


10.2.5 BnMemory.c

10.2.5.1 Introduction

This file contains the memory setup functions used by the *bigNum* functions in CryptoEngine()

10.2.5.2 Includes

```
#include "Tpm.h"
```

10.2.5.3 Functions

10.2.5.3.1 BnSetTop()

This function is used when the size of a bignum_t is changed. It makes sure that the unused words are set to zero and that any significant words of zeros are eliminated from the used size indicator.

```
LIB_EXPORT bigNum
BnSetTop(
    bigNum           bn,        // IN/OUT: number to clean
    crypt_uword_t    top        // IN: the new top
    )
{
    if(bn != NULL)
    {
        pAssert(top <= bn->allocated);
        // If forcing the size to be decreased, make sure that the words being
        // discarded are being set to 0
        while(bn->size > top)
            bn->d[--bn->size] = 0;
        bn->size = top;
        // Now make sure that the words that are left are 'normalized' (no high-order
        // words of zero.
        while((bn->size > 0) && (bn->d[bn->size - 1] == 0))
            bn->size -= 1;
    }
    return bn;
}
```

10.2.5.3.2 BnClearTop()

This function will make sure that all unused words are zero.

```
LIB_EXPORT bigNum
BnClearTop(
    bigNum          bn
    )
{
    crypt_uword_t       i;
//
    if(bn != NULL)
    {
        for(i = bn->size; i < bn->allocated; i++)
            bn->d[i] = 0;
        while((bn->size > 0) && (bn->d[bn->size] == 0))
            bn->size -= 1;
    }
    return bn;
}
```


10.2.5.3.3 BnInitializeWord()

This function is used to initialize an allocated *bigNum* with a word value. The *bigNum* does not have to be allocated with a single word.

```
39  LIB_EXPORT bigNum
40  BnInitializeWord(
41      bigNum          bn,         // IN:
42      crypt_uword_t   allocated,  // IN:
43      crypt_uword_t   word        // IN:
44      )
45  {
46      bn->allocated = allocated;
47      bn->size = (word != 0);
48      bn->d[0] = word;
49      while(allocated > 1)
50          bn->d[--allocated] = 0;
51      return bn;
52  }
```

10.2.5.3.4 BnInit()

This function initializes a stack allocated bignum_t. It initializes *allocated* and *size* and zeros the words of *d*.

```
53  LIB_EXPORT bigNum
54  BnInit(
55      bigNum              bn,
56      crypt_uword_t       allocated
57      )
58  {
59      if(bn != NULL)
60      {
61          bn->allocated = allocated;
62          bn->size = 0;
63          while(allocated != 0)
64              bn->d[--allocated] = 0;
65      }
66      return bn;
67  }
```

10.2.5.3.5 BnCopy()

Function to copy a bignum_t. If the output is NULL, then nothing happens. If the input is NULL, the output is set to zero.

```
68  LIB_EXPORT BOOL
69  BnCopy(
70      bigNum          out,
71      bigConst        in
72      )
73  {
74      if(in == out)
75          BnSetTop(out, BnGetSize(out));
76      else if(out != NULL)
77      {
78          if(in != NULL)
79          {
80              unsigned int        i;
81              pAssert(BnGetAllocated(out) >= BnGetSize(in));
82              for(i = 0; i < BnGetSize(in); i++)
83                  out->d[i] = in->d[i];
```



```
84              BnSetTop(out, BnGetSize(in));
85          }
86          else
87              BnSetTop(out, 0);
88      }
89      return TRUE;
90  }
91  #ifdef TPM_ALG_ECC
```

10.2.5.3.6 BnPointCopy()

Function to copy a bn point.

```
92  LIB_EXPORT BOOL
93  BnPointCopy(
94      bigPoint                pOut,
95      pointConst              pIn
96      )
97  {
98      return BnCopy(pOut->x, pIn->x)
99          && BnCopy(pOut->y, pIn->y)
100         && BnCopy(pOut->z, pIn->z);
101 }
```

10.2.5.3.7 BnInitializePoint()

This function is used to initialize a point structure with the addresses of the coordinates.

```
102 LIB_EXPORT bn_point_t *
103 BnInitializePoint(
104     bigPoint            p,      // OUT: structure to receive pointers
105     bigNum              x,      // IN: x coordinate
106     bigNum              y,      // IN: y coordinate
107     bigNum              z       // IN: x coordinate
108     )
109 {
110     p->x = x;
111     p->y = y;
112     p->z = z;
113     BnSetWord(z, 1);
114     return p;
115 }
116 #endif // TPM_ALG_ECC
```

10.2.6 CryptUtil.c

10.2.6.1 Introduction

This module contains the interfaces to the CryptoEngine() and provides miscellaneous cryptographic functions in support of the TPM.

10.2.6.2 Includes

```
1   #include "Tpm.h"
```

10.2.6.3 Hash/HMAC Functions

10.2.6.3.1 CryptHmacSign()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

Error Returns	Meaning
TPM_RC_HASH	not a valid hash

```
2   static TPM_RC
3   CryptHmacSign(
4       TPMT_SIGNATURE      *signature,     // OUT: signature
5       OBJECT              *signKey,       // IN: HMAC key sign the hash
6       TPM2B_DIGEST        *hashData       // IN: hash to be signed
7       )
8   {
9       HMAC_STATE       hmacState;
10      UINT32           digestSize;
11      digestSize = CryptHmacStart2B(&hmacState, signature->signature.any.hashAlg,
12                                    &signKey->sensitive.sensitive.bits.b);
13      CryptDigestUpdate2B(&hmacState.hashState, &hashData->b);
14      CryptHmacEnd(&hmacState, digestSize,
15                   (BYTE *)&signature->signature.hmac.digest);
16      return TPM_RC_SUCCESS;
17  }
```

10.2.6.3.2 CryptHMACVerifySignature()

This function will verify a signature signed by a HMAC key. Note that a caller needs to prepare *signature* with the signature algorithm (TPM_ALG_HMAC) and the hash algorithm to use. This function then builds a signature of that type.

Error Returns	Meaning
TPM_RC_SCHEME	not the proper scheme for this key type
TPM_RC_SIGNATURE	if invalid input or signature is not genuine

```
18  static TPM_RC
19  CryptHMACVerifySignature(
20      OBJECT              *signKey,       // IN: HMAC key signed the hash
21      TPM2B_DIGEST        *hashData,      // IN: digest being verified
22      TPMT_SIGNATURE      *signature      // IN: signature to be verified
23      )
24  {
25      TPMT_SIGNATURE           test;
26      TPMT_KEYEDHASH_SCHEME   *keyScheme =
27                           &signKey->publicArea.parameters.keyedHashDetail.scheme;
28  //
29      if((signature->sigAlg != TPM_ALG_HMAC)
30         || (signature->signature.hmac.hashAlg == TPM_ALG_NULL))
31          return TPM_RC_SCHEME;
32      // This check is not really needed for verification purposes. However, it does
33      // prevent someone from trying to validate a signature using a weaker hash
34      // algorithm than otherwise allowed by the key. That is, a key with a scheme
35      // other than TMP_ALG_NULL can only be used to validate signatures that have
36      // a matching scheme.
37      if((keyScheme->scheme != TPM_ALG_NULL)
38         && ((keyScheme->scheme != signature->sigAlg)
39             || (keyScheme->details.hmac.hashAlg != signature->signature.any.hashAlg)))
40          return TPM_RC_SIGNATURE;
```



```
41      test.sigAlg = signature->sigAlg;
42      test.signature.hmac.hashAlg = signature->signature.hmac.hashAlg;
43      CryptHmacSign(&test, signKey, hashData);
44      // Compare digest
45      if(!MemoryEqual(&test.signature.hmac.digest,
46                      &signature->signature.hmac.digest,
47                      CryptHashGetDigestSize(signature->signature.any.hashAlg)))
48          return TPM_RC_SIGNATURE;
49      return TPM_RC_SUCCESS;
50  }
```

10.2.6.3.3 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Error Returns	Meaning
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```
51  static TPM_RC
52  CryptGenerateKeyedHash(
53      TPMT_PUBLIC             *publicArea,        // IN/OUT: the public area template
54                                                  //     for the new key.
55      TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
56      TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
57      RAND_STATE              *rand               // IN: "entropy" source
58      )
59  {
60      TPMT_KEYEDHASH_SCHEME   *scheme;
61      TPM_ALG_ID               hashAlg;
62      UINT16                   hashBlockSize;
63      UINT16                   digestSize;
64      scheme = &publicArea->parameters.keyedHashDetail.scheme;
65      if(publicArea->type != TPM_ALG_KEYEDHASH)
66          return TPM_RC_FAILURE;
67      // Pick the limiting hash algorithm
68      if(scheme->scheme == TPM_ALG_NULL)
69          hashAlg = publicArea->nameAlg;
70      else if(scheme->scheme == TPM_ALG_XOR)
71          hashAlg = scheme->details.xor.hashAlg;
72      else
73          hashAlg = scheme->details.hmac.hashAlg;
74      hashBlockSize = CryptHashGetBlockSize(hashAlg);
75      digestSize = CryptHashGetDigestSize(hashAlg);
76      // if this is a signing or a decryption key, then the limit
77      // for the data size is the block size of the hash. This limit
78      // is set because larger values have lower entropy because of the
79      // HMAC function. The lower limit is 1/2 the size of the digest
80      //
81      //If the user provided the key, check that it is a proper size
82      if(sensitiveCreate->data.t.size != 0)
83      {
84          if(publicArea->objectAttributes.decrypt
85             || publicArea->objectAttributes.sign)
86          {
87              if(sensitiveCreate->data.t.size > hashBlockSize)
88                  return TPM_RC_SIZE;
89  #if 0   // May make this a FIPS-mode requirement
90              if(sensitiveCreate->data.t.size < (digestSize / 2))
91                  return TPM_RC_SIZE;
92  #endif
93          }
94          // If this is a data blob, then anything that will get past the unmarshaling
95          // is OK
```

```
96          MemoryCopy2B(&sensitive->sensitive.bits.b, &sensitiveCreate->data.b,
97                       sizeof(sensitive->sensitive.bits.t.buffer));
98      }
99      else
100     {
101         // If the TPM is going to generate the data, then set the size to be the
102         // size of the digest of the algorithm
103         int                      sizeInBits = digestSize * 8;
104         TPM2B_SENSITIVE_DATA    *key = &sensitive->sensitive.bits;
105         key->t.size = CryptRandMinMax(key->t.buffer, sizeInBits, sizeInBits / 2,
106                                       rand);
107     }
108     return TPM_RC_SUCCESS;
109 }
```

10.2.6.3.4 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.

```
110 BOOL
111 CryptIsSchemeAnonymous(
112     TPM_ALG_ID       scheme         // IN: the scheme algorithm to test
113     )
114 {
115     return scheme == ALG_ECDAA_VALUE;
116 }
```

10.2.6.4 Symmetric Functions

10.2.6.4.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```
117 void
118 ParmDecryptSym(
119     TPM_ALG_ID       symAlg,        // IN: the symmetric algorithm
120     TPM_ALG_ID       hash,          // IN: hash algorithm for KDFa
121     UINT16           keySizeInBits, // IN: the key size in bits
122     TPM2B           *key,           // IN: KDF HMAC key
123     TPM2B           *nonceCaller,   // IN: nonce caller
124     TPM2B           *nonceTpm,      // IN: nonce TPM
125     UINT32           dataSize,      // IN: size of parameter buffer
126     BYTE            *data           // OUT: buffer to be decrypted
127     )
128 {
129     // KDF output buffer
130     // It contains parameters for the CFB encryption
131     // From MSB to LSB, they are the key and iv
132     BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
133     // Symmetric key size in byte
134     UINT16           keySize = (keySizeInBits + 7) / 8;
135     TPM2B_IV         iv;
136     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
137     // If there is decryption to do...
138     if(iv.t.size > 0)
139     {
140         // Generate key and iv
141         CryptKDFa(hash, key, CFB_KEY, nonceCaller, nonceTpm,
142                   keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
143         MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
144         CryptSymmetricDecrypt(data, symAlg, keySizeInBits, symParmString,
```

```
                                &iv, TPM_ALG_CFB, dataSize, data);
    }
    return;
}
```

10.2.6.4.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
void
ParmEncryptSym(
    TPM_ALG_ID       symAlg,        // IN: symmetric algorithm
    TPM_ALG_ID       hash,          // IN: hash algorithm for KDFa
    UINT16           keySizeInBits, // IN: AES key size in bits
    TPM2B           *key,           // IN: KDF HMAC key
    TPM2B           *nonceCaller,   // IN: nonce caller
    TPM2B           *nonceTpm,      // IN: nonce TPM
    UINT32           dataSize,      // IN: size of parameter buffer
    BYTE            *data           // OUT: buffer to be encrypted
    )
{
    // KDF output buffer
    // It contains parameters for the CFB encryption
    BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
    // Symmetric key size in bytes
    UINT16           keySize = (keySizeInBits + 7) / 8;
    TPM2B_IV         iv;
    iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
    // See if there is any encryption to do
    if(iv.t.size > 0)
    {
        // Generate key and iv
        CryptKDFa(hash, key, CFB_KEY, nonceTpm, nonceCaller,
                  keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
        MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
        CryptSymmetricEncrypt(data, symAlg, keySizeInBits, symParmString, &iv,
                              TPM_ALG_CFB, dataSize, data);
    }
    return;
}
```

10.2.6.4.3 CryptGenerateKeySymmetric()

This function generates a symmetric cipher key. The derivation process is determined by the type of the provided *rand*

Error Returns	Meaning
TPM_RCS_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area
TPM_RCS_KEY	provided key value is not allowed

```
static TPM_RC
CryptGenerateKeySymmetric(
    TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
                                                //     for the new key.
    TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
    TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
    RAND_STATE              *rand               // IN: the "entropy" source for
    )
{
    UINT16           keyBits = publicArea->parameters.symDetail.sym.keyBits.sym;
```

```
190         TPM_RC          result;
191     //
192         // only do multiples of RADIX_BITS
193         if((keyBits % RADIX_BITS) != 0)
194             return TPM_RC_KEY_SIZE;
195         // If this is not a new key, then the provided key data must be the right size
196         if(sensitiveCreate->data.t.size != 0)
197         {
198             result = CryptSymKeyValidate(&publicArea->parameters.symDetail.sym,
199                                          (TPM2B_SYM_KEY *)&sensitiveCreate->data);
200             if(result == TPM_RC_SUCCESS)
201                 MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
202                              sizeof(sensitive->sensitive.sym.t.buffer));
203         }
204     #ifdef TPM_ALG_TDES
205         else if(publicArea->parameters.symDetail.sym.algorithm == TPM_ALG_TDES)
206         {
207             sensitive->sensitive.sym.t.size = keyBits / 8;
208             result = CryptGenerateKeyDes(publicArea, sensitive, rand);
209         }
210     #endif
211         else
212         {
213             sensitive->sensitive.sym.t.size = CryptRandMinMax(
214                         sensitive->sensitive.sym.t.buffer, keyBits, keyBits / 2, rand);
215             result = TPM_RC_SUCCESS;
216         }
217         return result;
218     }
```

10.2.6.4.4 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.

```
219     void
220     CryptXORObfuscation(
221         TPM_ALG_ID       hash,          // IN: hash algorithm for KDF
222         TPM2B           *key,           // IN: KDF key
223         TPM2B           *contextU,      // IN: contextU
224         TPM2B           *contextV,      // IN: contextV
225         UINT32           dataSize,      // IN: size of data buffer
226         BYTE            *data           // IN/OUT: data to be XORed in place
227         )
228     {
229         BYTE             mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
230         BYTE            *pm;
231         UINT32           i;
232         UINT32           counter = 0;
233         UINT16           hLen = CryptHashGetDigestSize(hash);
234         UINT32           requestSize = dataSize * 8;
235         INT32            remainBytes = (INT32)dataSize;
236         pAssert((key != NULL) && (data != NULL) && (hLen != 0));
237         // Call KDFa to generate XOR mask
238         for(; remainBytes > 0; remainBytes -= hLen)
239         {
240             // Make a call to KDFa to get next iteration
241             CryptKDFa(hash, key, XOR_KEY, contextU, contextV,
242                       requestSize, mask, &counter, TRUE);
243         // XOR next piece of the data
244             pm = mask;
245             for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
246                 *data++ ^= *pm++;
247         }
```

```
248         return;
249     }
```

10.2.6.5 Initialization and shut down

10.2.6.5.1 CryptInit()

This function is called when the TPM receives a _TPM_Init() indication.

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.

Return Value	Meaning
TRUE	initializations succeeded
FALSE	initialization failed and caller should place the TPM into Failure Mode

```
250     BOOL
251     CryptInit(
252         void
253         )
254     {
255         BOOL         ok;
256         // Initialize the vector of implemented algorithms
257         AlgorithmGetImplementedVector(&g_implementedAlgorithms);
258         // Indicate that all test are necessary
259         CryptInitializeToTest();
260         // Do any library initializations that are necessary. If any fails,
261         // the caller should go into failure mode;
262         ok = SupportLibInit();
263         ok = ok && CryptSymInit();
264         ok = ok && CryptRandInit();
265         ok = ok && CryptHashInit();
266     #ifdef TPM_ALG_RSA
267         ok = ok && CryptRsaInit();
268     #endif // TPM_ALG_RSA
269     #ifdef TPM_ALG_ECC
270         ok = ok && CryptEccInit();
271     #endif // TPM_ALG_ECC
272         return ok;
273     }
```

10.2.6.5.2 CryptStartup()

This function is called by TPM2_Startup() to initialize the functions in this cryptographic library and in the provided CryptoLibrary(). This function and CryptUtilInit() are both provided so that the implementation may move the initialization around to get the best interaction.

Return Value	Meaning
TRUE	startup succeeded
FALSE	startup failed and caller should place the TPM into Failure Mode

```
274     BOOL
275     CryptStartup(
276         STARTUP_TYPE     type           // IN: the startup type
277         )
278     {
279         BOOL            OK;
```

```
280         NOT_REFERENCED(type);
281         OK = CryptSymStartup() && CryptRandStartup() && CryptHashStartup()
282     #ifdef TPM_ALG_RSA
283             &&  CryptRsaStartup()
284     #endif // TPM_ALG_RSA
285     #ifdef TPM_ALG_ECC
286             &&  CryptEccStartup()
287     #endif // TPM_ALG_ECC
288             ;
289     #ifdef TPM_ALG_ECC
290         // Don't directly check for SU_RESET because that is the default
291         if(OK && (type != SU_RESTART) && (type != SU_RESUME))
292         {
293         // If the shutdown was orderly, then the values recovered from NV will
294         // be OK to use.
295             // Get a new  random commit nonce
296             gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
297             CryptRandomGenerate(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
298             // Reset the counter and commit array
299             gr.commitCounter = 0;
300             MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
301         }
302     #endif // TPM_ALG_ECC
303         return OK;
304     }
```

10.2.6.6 Algorithm-Independent Functions

10.2.6.6.1 Introduction

These functions are used generically when a function of a general type (e.g., symmetric encryption) is required. The functions will modify the parameters as required to interface to the indicated algorithms.

10.2.6.6.2 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

```
305     BOOL
306     CryptIsAsymAlgorithm(
307         TPM_ALG_ID       algID          // IN: algorithm ID
308         )
309     {
310         switch(algID)
311         {
312     #ifdef TPM_ALG_RSA
313             case  TPM_ALG_RSA:
314     #endif
315     #ifdef TPM_ALG_ECC
316             case TPM_ALG_ECC:
317     #endif
318                 return TRUE;
319                 break;
320             default:
321                 break;
322         }
323         return FALSE;
```

```
}
```

10.2.6.6.3 CryptSecretEncrypt()

This function creates a secret value and its associated secret structure using an asymmetric algorithm.

This function is used by TPM2_Rewrap() TPM2_MakeCredential(), and TPM2_Duplicate().

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a valid decryption key
TPM_RC_KEY	invalid ECC key (public point is not on the curve)
TPM_RC_SCHEME	RSA key with an unsupported padding scheme
TPM_RC_VALUE	numeric value of the data to be decrypted is greater than the RSA key modulus

```
TPM_RC
CryptSecretEncrypt(
    OBJECT                  *encryptKey,    // IN: encryption key object
    const TPM2B             *label,         // IN: a null-terminated string as L
    TPM2B_DATA              *data,          // OUT: secret value
    TPM2B_ENCRYPTED_SECRET  *secret         // OUT: secret structure
    )
{
    TPMT_RSA_DECRYPT         scheme;
    TPM_RC                   result = TPM_RC_SUCCESS;
//
    if(data == NULL || secret == NULL)
        return TPM_RC_FAILURE;
    // The output secret value has the size of the digest produced by the nameAlg.
    data->t.size = CryptHashGetDigestSize(encryptKey->publicArea.nameAlg);
    // The encryption scheme is OAEP using the nameAlg of the encrypt key.
    scheme.scheme = TPM_ALG_OAEP;
    scheme.details.anySig.hashAlg = encryptKey->publicArea.nameAlg;
    if(encryptKey->publicArea.objectAttributes.decrypt != SET)
        return TPM_RC_ATTRIBUTES;
    switch(encryptKey->publicArea.type)
    {
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
        {
            // Create secret data from RNG
            CryptRandomGenerate(data->t.size, data->t.buffer);
            // Encrypt the data by RSA OAEP into encrypted secret
            result = CryptRsaEncrypt((TPM2B_PUBLIC_KEY_RSA *)secret, &data->b,
                                     encryptKey, &scheme, label, NULL);
        }
        break;
#endif //TPM_ALG_RSA
#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
        {
            TPMS_ECC_POINT      eccPublic;
            TPM2B_ECC_PARAMETER eccPrivate;
            TPMS_ECC_POINT      eccSecret;
            BYTE                *buffer = secret->t.secret;
            // Need to make sure that the public point of the key is on the
            // curve defined by the key.
            if(!CryptEccIsPointOnCurve(
                encryptKey->publicArea.parameters.eccDetail.curveID,
                &encryptKey->publicArea.unique.ecc))
                result = TPM_RC_KEY;
```



```
371             else
372             {
373                 // Call crypto engine to create an auxiliary ECC key
374                 // We assume crypt engine initialization should always success.
375                 // Otherwise, TPM should go to failure mode.
376                 CryptEccNewKeyPair(&eccPublic, &eccPrivate,
377                                    encryptKey-
    >publicArea.parameters.eccDetail.curveID);
378                     // Marshal ECC public to secret structure. This will be used by
    the
379                     // recipient to decrypt the secret with their private key.
380                 secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
381                 // Compute ECDH shared secret which is R = [d]Q where d is the
382                 // private part of the ephemeral key and Q is the public part of a
383                 // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
384                 // because the auxiliary ECC key is just created according to the
385                 // parameters of input ECC encrypt key.
386                 if(CryptEccPointMultiply(&eccSecret,
387                                          encryptKey-
    >publicArea.parameters.eccDetail.curveID,
388                                          &encryptKey->publicArea.unique.ecc,
    &eccPrivate,
389                                          NULL, NULL) != TPM_RC_SUCCESS)
390                     result = TPM_RC_KEY;
391                 else
392                 {
393                     // The secret value is computed from Z using KDFe as:
394                     // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
395                     // Where:
396                     //  HashID  the nameAlg of the decrypt key
397                     //  Z   the x coordinate (Px) of the product (P) of the point
398                     //      (Q) of the secret and the private x coordinate (de,V)
399                     //      of the decryption key
400                     //  Use a null-terminated string containing "SECRET"
401                     //  PartyUInfo  the x coordinate of the point in the secret
402                     //              (Qe,U )
403                     //  PartyVInfo  the x coordinate of the public key (Qs,V )
404                     //  bits    the number of bits in the digest of HashID
405                     // Retrieve seed from KDFe
406                     CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
407                               label, &eccPublic.x.b,
408                               &encryptKey->publicArea.unique.ecc.x.b,
409                               data->t.size * 8, data->t.buffer);
410                 }
411             }
412         }
413         break;
414 #endif //TPM_ALG_ECC
415         default:
416             FAIL(FATAL_ERROR_INTERNAL);
417             break;
418     }
419     return result;
420 }
```

10.2.6.6.4 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

```
421  TPM_RC
422  CryptSecretDecrypt(
423      OBJECT                  *decryptKey,    // IN: decrypt key
424      TPM2B_NONCE             *nonceCaller,   // IN: nonceCaller.  It is needed for
425                                              //     symmetric decryption.  For
426                                              //     asymmetric decryption, this
427                                              //     parameter is NULL
428      const TPM2B             *label,         // IN: a value for L
429      TPM2B_ENCRYPTED_SECRET  *secret,        // IN: input secret
430      TPM2B_DATA              *data           // OUT: decrypted secret value
431      )
432  {
433      TPM_RC      result = TPM_RC_SUCCESS;
434      // Decryption for secret
435      switch(decryptKey->publicArea.type)
436      {
437  #ifdef TPM_ALG_RSA
438          case TPM_ALG_RSA:
439          {
440              TPMT_RSA_DECRYPT        scheme;
441              TPMT_RSA_SCHEME         *keyScheme
442                  = &decryptKey->publicArea.parameters.rsaDetail.scheme;
443              UINT16                   digestSize;
444              scheme = *(TPMT_RSA_DECRYPT *)keyScheme;
445              // If the key scheme is TPM_ALG_NULL, set the scheme to OAEP and
446              // set the algorithm to the name algorithm.
447              if(scheme.scheme == TPM_ALG_NULL)
448              {
449                  // Use OAEP scheme
450                  scheme.scheme = TPM_ALG_OAEP;
451                  scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
452              }
453              // use the digestSize as an indicator of whether or not the scheme
454              // is using a supported hash algorithm.
455              // Note: depending on the scheme used for encryption, a hashAlg might
456              // not be needed. However, the return value has to have some upper
457              // limit on the size. In this case, it is the size of the digest of the
458              // hash algorithm. It is checked after the decryption is done but, there
459              // is no point in doing the decryption if the size is going to be
460              // 'wrong' anyway.
461              digestSize = CryptHashGetDigestSize(scheme.details.oaep.hashAlg);
462              if(scheme.scheme != TPM_ALG_OAEP || digestSize == 0)
463                  return TPM_RC_SCHEME;
464              // Set the output buffer capacity
465              data->t.size = sizeof(data->t.buffer);
```

```
466             // Decrypt seed by RSA OAEP
467             result = CryptRsaDecrypt(&data->b, &secret->b,
468                                      decryptKey, &scheme, label);
469             if((result == TPM_RC_SUCCESS) && (data->t.size > digestSize))
470                 result = TPM_RC_VALUE;
471         }
472         break;
473 #endif //TPM_ALG_RSA
474 #ifdef TPM_ALG_ECC
475         case TPM_ALG_ECC:
476         {
477             TPMS_ECC_POINT       eccPublic;
478             TPMS_ECC_POINT       eccSecret;
479             BYTE                *buffer = secret->t.secret;
480             INT32                size = secret->t.size;
481             // Retrieve ECC point from secret buffer
482             result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
483             if(result == TPM_RC_SUCCESS)
484             {
485                 result = CryptEccPointMultiply(&eccSecret,
486                              decryptKey->publicArea.parameters.eccDetail.curveID,
487                              &eccPublic, &decryptKey->sensitive.sensitive.ecc,
488                              NULL, NULL);
489                 if(result == TPM_RC_SUCCESS)
490                 {
491                     // Set the size of the "recovered" secret value to be the size
492                     // of the digest produced by the nameAlg.
493                     data->t.size =
494                         CryptHashGetDigestSize(decryptKey->publicArea.nameAlg);
495                 // The secret value is computed from Z using KDFe as:
496                 // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
497                 // Where:
498                 //  HashID -- the nameAlg of the decrypt key
499                 //  Z --  the x coordinate (Px) of the product (P) of the point
500                 //        (Q) of the secret and the private x coordinate (de,V)
501                 //        of the decryption key
502                 //  Use -- a null-terminated string containing "SECRET"
503                 //  PartyUInfo -- the x coordinate of the point in the secret
504                 //              (Qe,U )
505                 //  PartyVInfo -- the x coordinate of the public key (Qs,V )
506                 //  bits -- the number of bits in the digest of HashID
507                 // Retrieve seed from KDFe
508                     CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
509                              &eccPublic.x.b,
510                              &decryptKey->publicArea.unique.ecc.x.b,
511                              data->t.size * 8, data->t.buffer);
512                 }
513             }
514         }
515         break;
516 #endif //TPM_ALG_ECC
517 #ifndef TPM_ALG_KEYEDHASH
518 #   error   "KEYEDHASH support is required"
519 #endif
520         case TPM_ALG_KEYEDHASH:
521             // The seed size can not be bigger than the digest size of nameAlg
522             if(secret->t.size >
523                CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
524                 result = TPM_RC_VALUE;
525             else
526             {
527                 // Retrieve seed by XOR Obfuscation:
528                 //    seed = XOR(secret, hash, key, nonceCaller, nullNonce)
529                 //    where:
530                 //    secret  the secret parameter from the TPM2_StartAuthHMAC
531                 //            command
```

```
532                 //            which contains the seed value
533                 //    hash    nameAlg  of tpmKey
534                 //    key     the key or data value in the object referenced by
535                 //            entityHandle in the TPM2_StartAuthHMAC command
536                 //    nonceCaller the parameter from the TPM2_StartAuthHMAC command
537                 //    nullNonce   a zero-length nonce
538                 // XOR Obfuscation in place
539                 CryptXORObfuscation(decryptKey->publicArea.nameAlg,
540                                     &decryptKey->sensitive.sensitive.bits.b,
541                                     &nonceCaller->b, NULL,
542                                     secret->t.size, secret->t.secret);
543                 // Copy decrypted seed
544                 MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
545             }
546             break;
547         case TPM_ALG_SYMCIPHER:
548         {
549             TPM2B_IV                iv = {{0}};
550             TPMT_SYM_DEF_OBJECT     *symDef;
551             // The seed size can not be bigger than the digest size of nameAlg
552             if(secret->t.size >
553                CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
554                 result = TPM_RC_VALUE;
555             else
556             {
557                 symDef = &decryptKey->publicArea.parameters.symDetail.sym;
558                 iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
559                                                        symDef->keyBits.sym);
560                 if(iv.t.size == 0)
561                     return TPM_RC_FAILURE;
562                 if(nonceCaller->t.size >= iv.t.size)
563                 {
564                     MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
565                 }
566                 else
567                 {
568                     if(nonceCaller->t.size > sizeof(iv.t.buffer))
569                         return TPM_RC_FAILURE;
570                     MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
571                                nonceCaller->t.size);
572                 }
573                 // make sure secret will fit
574                 if(secret->t.size > data->t.size)
575                     return TPM_RC_FAILURE;
576                 data->t.size = secret->t.size;
577                 // CFB decrypt, using nonceCaller as iv
578                 CryptSymmetricDecrypt(data->t.buffer, symDef->algorithm,
579                                       symDef->keyBits.sym,
580                                       decryptKey->sensitive.sensitive.sym.t.buffer,
581                                       &iv, TPM_ALG_CFB, secret->t.size,
582                                       secret->t.secret);
583             }
584         }
585         break;
586         default:
587             FAIL(FATAL_ERROR_INTERNAL);
588             break;
589     }
590     return result;
591 }
```

10.2.6.6.5 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```
592 void
593 CryptParameterEncryption(
594     TPM_HANDLE       handle,            // IN: encrypt session handle
595     TPM2B           *nonceCaller,       // IN: nonce caller
596     UINT16           leadingSizeInByte, // IN: the size of the leading size field in
597                                         //     bytes
598     TPM2B_AUTH      *extraKey,          // IN: additional key material other than
599                                         //     sessionAuth
600     BYTE            *buffer             // IN/OUT: parameter buffer to be encrypted
601     )
602 {
603     SESSION     *session = SessionGet(handle);  // encrypt session
604     TPM2B_TYPE(TEMP_KEY, (sizeof(extraKey->t.buffer)
605                           + sizeof(session->sessionKey.t.buffer)));
606     TPM2B_TEMP_KEY        key;               // encryption key
607     UINT32               cipherSize = 0;    // size of cipher text
608 //
609     // Retrieve encrypted data size.
610     if(leadingSizeInByte == 2)
611     {
612         // Extract the first two bytes as the size field as the data size
613         // encrypt
614         cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
615         // advance the buffer
616         buffer = &buffer[2];
617     }
618 #ifdef      TPM4B
619     else if(leadingSizeInByte == 4)
620     {
621         // use the first four bytes to indicate the number of bytes to encrypt
622         cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
623         //advance pointer
624         buffer = &buffer[4];
625     }
626 #endif
627     else
628     {
629         FAIL(FATAL_ERROR_INTERNAL);
630     }
631     // Compute encryption key by concatenating sessionAuth with extra key
632     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
633     MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
634     if(session->symmetric.algorithm == TPM_ALG_XOR)
635         // XOR parameter encryption formulation:
636         //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
637         CryptXORObfuscation(session->authHashAlg, &(key.b),
638                             &(session->nonceTPM.b),
639                             nonceCaller, cipherSize, buffer);
640     else
641         ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
642                        session->symmetric.keyBits.aes, &(key.b),
643                        nonceCaller, &(session->nonceTPM.b),
644                        cipherSize, buffer);
645     return;
646 }
```

10.2.6.6.6 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.

```
647 TPM_RC
648 CryptParameterDecryption(
649     TPM_HANDLE       handle,            // IN: encrypted session handle
650     TPM2B           *nonceCaller,       // IN: nonce caller
651     UINT32           bufferSize,        // IN: size of parameter buffer
652     UINT16           leadingSizeInByte, // IN: the size of the leading size field in
653                                         //     byte
654     TPM2B_AUTH      *extraKey,          // IN: the authValue
655     BYTE            *buffer             // IN/OUT: parameter buffer to be decrypted
656     )
657 {
658     SESSION         *session = SessionGet(handle);  // encrypt session
659     // The HMAC key is going to be the concatenation of the session key and any
660     // additional key material (like the authValue). The size of both of these
661     // is the size of the buffer which can contain a TPMT_HA.
662     TPM2B_TYPE(HMAC_KEY, (sizeof(extraKey->t.buffer)
663                           + sizeof(session->sessionKey.t.buffer)));
664     TPM2B_HMAC_KEY          key;            // decryption key
665     UINT32                  cipherSize = 0; // size of cipher text
666 //
667     // Retrieve encrypted data size.
668     if(leadingSizeInByte == 2)
669     {
670         // The first two bytes of the buffer are the size of the
671         // data to be decrypted
672         cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
673         buffer = &buffer[2];   // advance the buffer
674     }
675 #ifdef  TPM4B
676     else if(leadingSizeInByte == 4)
677     {
678         // the leading size is four bytes so get the four byte size field
679         cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
680         buffer = &buffer[4];   //advance pointer
681     }
682 #endif
683     else
684     {
685         FAIL(FATAL_ERROR_INTERNAL);
686     }
687     if(cipherSize > bufferSize)
688         return TPM_RC_SIZE;
689     // Compute decryption key by concatenating sessionAuth with extra input key
690     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
691     MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
692     if(session->symmetric.algorithm == TPM_ALG_XOR)
693         // XOR parameter decryption formulation:
694         //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
695         // Call XOR obfuscation function
696         CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
697                             &(session->nonceTPM.b), cipherSize, buffer);
698     else
699         // Assume that it is one of the symmetric block ciphers.
700         ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
701                        session->symmetric.keyBits.sym,
702                        &key.b, nonceCaller, &session->nonceTPM.b,
703                        cipherSize, buffer);
704     return TPM_RC_SUCCESS;
705 }
```


10.2.6.6.7 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```
706  void
707  CryptComputeSymmetricUnique(
708      TPMT_PUBLIC     *publicArea,    // IN: the object's public area
709      TPMT_SENSITIVE  *sensitive,     // IN: the associated sensitive area
710      TPM2B_DIGEST    *unique         // OUT: unique buffer
711      )
712  {
713      // For parents (symmetric and derivation), use an HMAC to compute
714      // the 'unique' field
715      if(publicArea->objectAttributes.restricted
716         && publicArea->objectAttributes.decrypt)
717      {
718          // Unique field is HMAC(sensitive->seedValue, sensitive->sensitive)
719          HMAC_STATE      hmacState;
720          unique->b.size = CryptHmacStart2B(&hmacState, publicArea->nameAlg,
721                                            &sensitive->seedValue.b);
722          CryptDigestUpdate2B(&hmacState.hashState,
723                              &sensitive->sensitive.any.b);
724          CryptHmacEnd2B(&hmacState, &unique->b);
725      }
726      else
727      {
728          HASH_STATE  hashState;
729          // Unique := Hash(sensitive->seedValue || sensitive->sensitive)
730          unique->t.size = CryptHashStart(&hashState, publicArea->nameAlg);
731          CryptDigestUpdate2B(&hashState, &sensitive->seedValue.b);
732          CryptDigestUpdate2B(&hashState, &sensitive->sensitive.any.b);
733          CryptHashEnd2B(&hashState, &unique->b);
734      }
735      return;
736  }
```

10.2.6.6.8 CryptCreateObject()

This function creates an object. For an asymmetric key, it will create a key pair and, for a parent key, a seed value for child protections.

For an symmetric object, (TPM_ALG_SYMCIPHER or TPM_ALG_KEYEDHASH), it will create a secret key if the caller did not provide one. It will create a random secret seed value that is hashed with the secret value to create the public unique value.

publicArea, *sensitive*, and *sensitiveCreate* are the only required parameters and are the only ones that are used by TPM2_Create(). The other parameters are optional and are used when the generated Object needs to be deterministic. This is the case for both Primary Objects and Derived Objects.

When a seed value is provided, a RAND_STATE will be populated and used for all operations in the object generation that require a random number. In the simplest case, TPM2_CreatePrimary() will use *seed*, *label* and *context* with context being the hash of the template. If the Primary Object is in the Endorsement hierarchy, it will also populate *proof* with *ehProof*.

For derived keys, *seed* will be the secret value from the parent, *label* and *context* will be set according to the parameters of TPM2_Derive() and *hashAlg* will be set which causes the RAND_STATE to be a KDF generator.

Error Returns	Meaning
TPM_RC_KEY	a provided key is not an allowed value
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

```
737  TPM_RC
738  CryptCreateObject(
739      OBJECT                  *object,            // IN: new object structure pointer
740      TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation
741      RAND_STATE              *rand               // IN: the random number generator
742                                                  //      to use
743      )
744  {
745      TPMT_PUBLIC             *publicArea = &object->publicArea;
746      TPM_RC                   result = TPM_RC_SUCCESS;
747  //
748      // Set the sensitive type for the object
749      object->sensitive.sensitiveType = publicArea->type;
750      // For all objects, copy the initial authorization data
751      object->sensitive.authValue = sensitiveCreate->userAuth;
752      // If the TPM is the source of the data, set the size of the provided data to
753      // zero so that there's no confusion about what to do.
754      if(object->publicArea.objectAttributes.sensitiveDataOrigin)
755          sensitiveCreate->data.t.size = 0;
756      // Generate the key and unique fields for the asymmetric keys and just the
757      // sensitive value for symmetric object
758      switch(publicArea->type)
759      {
760  #ifdef TPM_ALG_RSA
761          // Create RSA key
762          case TPM_ALG_RSA:
763              // RSA uses full object so that it has a place to put the private
764              // exponent
765              result = CryptRsaGenerateKey(object, rand);
766              break;
767  #endif // TPM_ALG_RSA
768  #ifdef TPM_ALG_ECC
769          // Create ECC key
770          case TPM_ALG_ECC:
771              result = CryptEccGenerateKey(&object->publicArea, &object->sensitive,
772                                           rand);
773              break;
774  #endif // TPM_ALG_ECC
775          case TPM_ALG_SYMCIPHER:
776              result = CryptGenerateKeySymmetric(&object->publicArea,
777                                                 &object->sensitive,
778                                                 sensitiveCreate, rand);
779              break;
780          case TPM_ALG_KEYEDHASH:
781              result = CryptGenerateKeyedHash(&object->publicArea, &object->sensitive,
782                                              sensitiveCreate, rand);
783              break;
784          default:
785              FAIL(FATAL_ERROR_INTERNAL);
786              break;
```

```
787         }
788         if(result != TPM_RC_SUCCESS)
789             return result;
790     // Create the sensitive seed value
791         // If this is a primary key in the endorsement hierarchy, stir the DRBG state
792         // This implementation uses the proof of the storage hierarchy but the
793         // proof of the endorsement hierarchy would also work
794         if(object->attributes.primary && object->attributes.epsHierarchy)
795             DRBG_AdditionalData((DRBG_STATE *)rand, &gp.shProof.b);
796         // Set the seed value to the size of the digest produced by the nameAlg
797         object->sensitive.seedValue.b.size
798             = CryptHashGetDigestSize(publicArea->nameAlg);
799         object->sensitive.seedValue.t.size = CryptRandMinMax(
800                     object->sensitive.seedValue.t.buffer,
801                     object->sensitive.seedValue.t.size * 8,
802                     object->sensitive.seedValue.t.size * 8 / 2, rand);
803         // For symmetric values, need to compute the unique value
804         if(publicArea->type == TPM_ALG_SYMCIPHER
805            || publicArea->type == TPM_ALG_KEYEDHASH)
806         {
807             CryptComputeSymmetricUnique(&object->publicArea, &object->sensitive,
808                                         &object->publicArea.unique.sym);
809         }
810         else
811         {
812             // if this is an asymmetric key and it isn't a parent, then
813             // can get rid of the seed.
814             if(publicArea->objectAttributes.sign
815                || !publicArea->objectAttributes.restricted)
816                 memset(&object->sensitive.seedValue, 0,
817                        sizeof(object->sensitive.seedValue));
818         }
819         // Compute the name
820         PublicMarshalAndComputeName(&object->publicArea, &object->name);
821         return result;
822     }
```

10.2.6.6.9 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

```
823     TPMI_ALG_HASH
824     CryptGetSignHashAlg(
825         TPMT_SIGNATURE  *auth           // IN: signature
826         )
827     {
828         if(auth->sigAlg == TPM_ALG_NULL)
829             FAIL(FATAL_ERROR_INTERNAL);
830         // Get authHash algorithm based on signing scheme
831         switch(auth->sigAlg)
832         {
833     #ifdef  TPM_ALG_RSA
834         // If RSA is supported, both RSASSA and RSAPSS are required
835     #   if !defined TPM_ALG_RSASSA || !defined TPM_ALG_RSAPSS
836     #       error "RSASSA and RSAPSS are required for RSA"
837     #   endif
838             case TPM_ALG_RSASSA:
839                 return auth->signature.rsassa.hash;
840             case TPM_ALG_RSAPSS:
841                 return auth->signature.rsapss.hash;
842     #endif //TPM_ALG_RSA
843     #ifdef TPM_ALG_ECC
844         // If ECC is defined, ECDSA is mandatory
```



```
#   ifndef  TPM_ALG_ECDSA
#       error "ECDSA is requried for ECC"
#   endif
        case TPM_ALG_ECDSA:
    // SM2 and ECSCHNORR are optional
#   ifdef    TPM_ALG_SM2
        case TPM_ALG_SM2:
#   endif
#   ifdef    TPM_ALG_ECSCHNORR
        case TPM_ALG_ECSCHNORR:
#   endif
        //all ECC signatures look the same
            return auth->signature.ecdsa.hash;
#   ifdef TPM_ALG_ECDAA
        // Don't know how to verify an ECDAA signature
        case TPM_ALG_ECDAA:
            break;
#   endif
#endif //TPM_ALG_ECC
        case TPM_ALG_HMAC:
            return auth->signature.hmac.hashAlg;
        default:
            break;
    }
    return TPM_ALG_NULL;
}
```

10.2.6.6.10 CryptIsSplitSign()

This function us used to determine if the signing operation is a split signing operation that required a TPM2_Commit().

```
BOOL
CryptIsSplitSign(
    TPM_ALG_ID       scheme         // IN: the algorithm selector
    )
{
    switch(scheme)
    {
#   ifdef   TPM_ALG_ECDAA
        case TPM_ALG_ECDAA:
            return TRUE;
            break;
#   endif   // TPM_ALG_ECDAA
        default:
            return FALSE;
            break;
    }
}
```

10.2.6.6.11 CryptIsAsymSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```
BOOL
CryptIsAsymSignScheme(
    TPMI_ALG_PUBLIC          publicType,        // IN: Type of the object
    TPMI_ALG_ASYM_SCHEME     scheme             // IN: the scheme
    )
{
    BOOL            isSignScheme = TRUE;
    switch(publicType)
    {
```

```
897 #ifdef TPM_ALG_RSA
898         case TPM_ALG_RSA:
899             switch(scheme)
900             {
901 #   if !defined TPM_ALG_RSASSA  || !defined TPM_ALG_RSAPSS
902 #       error "RSASSA and PSAPSS required if RSA used."
903 #   endif
904                 case TPM_ALG_RSASSA:
905                 case TPM_ALG_RSAPSS:
906                     break;
907                 default:
908                     isSignScheme = FALSE;
909                     break;
910             }
911             break;
912 #endif //TPM_ALG_RSA
913 #ifdef TPM_ALG_ECC
914         // If ECC is implemented ECDSA is required
915         case TPM_ALG_ECC:
916             switch(scheme)
917             {
918                 // Support for ECDSA is required for ECC
919                 case TPM_ALG_ECDSA:
920 #ifdef  TPM_ALG_ECDAA // ECDAA is optional
921                 case TPM_ALG_ECDAA:
922 #endif
923 #ifdef   TPM_ALG_ECSCHNORR // Schnorr is also optional
924                 case TPM_ALG_ECSCHNORR:
925 #endif
926 #ifdef  TPM_ALG_SM2 // SM2 is optional
927                 case TPM_ALG_SM2:
928 #endif
929                     break;
930                 default:
931                     isSignScheme = FALSE;
932                     break;
933             }
934             break;
935 #endif //TPM_ALG_ECC
936         default:
937             isSignScheme = FALSE;
938             break;
939     }
940     return isSignScheme;
941 }
```

10.2.6.6.12 CryptIsAsymDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
942 BOOL
943 CryptIsAsymDecryptScheme(
944     TPMI_ALG_PUBLIC          publicType,        // IN: Type of the object
945     TPMI_ALG_ASYM_SCHEME     scheme             // IN: the scheme
946     )
947 {
948     BOOL        isDecryptScheme = TRUE;
949     switch(publicType)
950     {
951 #ifdef TPM_ALG_RSA
952         case TPM_ALG_RSA:
953             switch(scheme)
954             {
955                 case TPM_ALG_RSAES:
```

```
956                 case TPM_ALG_OAEP:
957                     break;
958                 default:
959                     isDecryptScheme = FALSE;
960                     break;
961             }
962             break;
963 #endif //TPM_ALG_RSA
964 #ifdef TPM_ALG_ECC
965         // If ECC is implemented ECDH is required
966         case TPM_ALG_ECC:
967             switch(scheme)
968             {
969 #ifndef TPM_ALG_ECDH
970 #   error "ECDH is required for ECC"
971 #endif
972                 case TPM_ALG_ECDH:
973 #ifdef TPM_ALG_SM2
974                 case TPM_ALG_SM2:
975 #endif
976 #ifdef  TPM_ALG_ECMQV
977                 case TPM_ALG_ECMQV:
978 #endif
979                     break;
980                 default:
981                     isDecryptScheme = FALSE;
982                     break;
983             }
984             break;
985 #endif //TPM_ALG_ECC
986         default:
987             isDecryptScheme = FALSE;
988             break;
989     }
990     return isDecryptScheme;
991 }
```

10.2.6.6.13 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.

This function should not be called if '*signObject->publicArea*.type' == TPM_ALG_SYMCIPHER.

Return Value	Meaning
FALSE	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*
TRUE	scheme selected

```
992 BOOL
993 CryptSelectSignScheme(
994     OBJECT              *signObject,    // IN: signing key
995     TPMT_SIG_SCHEME     *scheme         // IN/OUT: signing scheme
996     )
997 {
998     TPMT_SIG_SCHEME     *objectScheme;
```

```
 999       TPMT_PUBLIC         *publicArea;
1000       BOOL                 OK;
1001       // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
1002       // of the setting of scheme
1003       if(signObject == NULL)
1004       {
1005           OK = TRUE;
1006           scheme->scheme = TPM_ALG_NULL;
1007           scheme->details.any.hashAlg = TPM_ALG_NULL;
1008       }
1009       else
1010       {
1011           // assignment to save typing.
1012           publicArea = &signObject->publicArea;
1013           // A symmetric cipher can be used to encrypt and decrypt but it can't
1014           // be used for signing
1015           if(publicArea->type == TPM_ALG_SYMCIPHER)
1016               return FALSE;
1017           // Point to the scheme object
1018           if(CryptIsAsymAlgorithm(publicArea->type))
1019               objectScheme =
1020               (TPMT_SIG_SCHEME *)&publicArea->parameters.asymDetail.scheme;
1021           else
1022               objectScheme =
1023               (TPMT_SIG_SCHEME *)&publicArea->parameters.keyedHashDetail.scheme;
1024       // If the object doesn't have a default scheme, then use the
1025       // input scheme.
1026           if(objectScheme->scheme == TPM_ALG_NULL)
1027           {
1028               // Input and default can't both be NULL
1029               OK = (scheme->scheme != TPM_ALG_NULL);
1030               // Assume that the scheme is compatible with the key. If not,
1031               // an error will be generated in the signing operation.
1032           }
1033           else if(scheme->scheme == TPM_ALG_NULL)
1034           {
1035               // input scheme is NULL so use default
1036               // First, check to see if the default requires that the caller
1037               // provided scheme data
1038               OK = !CryptIsSplitSign(objectScheme->scheme);
1039               if(OK)
1040               {
1041                   // The object has a scheme and the input is TPM_ALG_NULL so copy
1042                   // the object scheme as the final scheme. It is better to use a
1043                   // structure copy than a copy of the individual fields.
1044                   *scheme = *objectScheme;
1045               }
1046           }
1047           else
1048           {
1049               // Both input and object have scheme selectors
1050               // If the scheme and the hash are not the same then...
1051               // NOTE: the reason that there is no copy here is that the input
1052               // might contain extra data for a split signing scheme and that
1053               // data is not in the object so, it has to be preserved.
1054               OK = (objectScheme->scheme == scheme->scheme)
1055                   && (objectScheme->details.any.hashAlg
1056                       == scheme->details.any.hashAlg);
1057           }
1058       }
1059       return OK;
1060   }
```

10.2.6.6.14 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)

```
1061  TPM_RC
1062  CryptSign(
1063      OBJECT              *signKey,       // IN: signing key
1064      TPMT_SIG_SCHEME     *signScheme,    // IN: sign scheme.
1065      TPM2B_DIGEST        *digest,        // IN: The digest being signed
1066      TPMT_SIGNATURE      *signature      // OUT: signature
1067      )
1068  {
1069      TPM_RC               result = TPM_RC_SCHEME;
1070      // Initialize signature scheme
1071      signature->sigAlg = signScheme->scheme;
1072      // If the signature algorithm is TPM_ALG_NULL or the signing key is NULL,
1073      // then we are done
1074      if((signature->sigAlg == TPM_ALG_NULL) || (signKey == NULL))
1075          return TPM_RC_SUCCESS;
1076      // Initialize signature hash
1077      // Note: need to do the check for TPM_ALG_NULL first because the null scheme
1078      // doesn't have a hashAlg member.
1079      signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
1080      // perform sign operation based on different key type
1081      switch(signKey->publicArea.type)
1082      {
1083  #ifdef TPM_ALG_RSA
1084          case TPM_ALG_RSA:
1085              result = CryptRsaSign(signature, signKey, digest, NULL);
1086              break;
1087  #endif //TPM_ALG_RSA
1088  #ifdef TPM_ALG_ECC
1089          case TPM_ALG_ECC:
1090              // The reason that signScheme is passed to CryptEccSign but not to the
1091              // other signing methods is that the signing for ECC may be split and
1092              // need the 'r' value that is in the scheme but not in the signature.
1093              result = CryptEccSign(signature, signKey, digest,
1094                                    (TPMT_ECC_SCHEME *)signScheme, NULL);
1095              break;
1096  #endif //TPM_ALG_ECC
1097          case TPM_ALG_KEYEDHASH:
1098              result = CryptHmacSign(signature, signKey, digest);
1099              break;
1100          default:
1101              FAIL(FATAL_ERROR_INTERNAL);
1102              break;
1103      }
1104      return result;
1105  }
```


10.2.6.6.15 CryptValidateSignature()

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.

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SCHEME	the scheme is not supported
TPM_RC_HANDLE	an HMAC key was selected but the private part of the key is not loaded

```
1106  TPM_RC
1107  CryptValidateSignature(
1108      TPMI_DH_OBJECT   keyHandle,     // IN: The handle of sign key
1109      TPM2B_DIGEST    *digest,        // IN: The digest being validated
1110      TPMT_SIGNATURE  *signature      // IN: signature
1111      )
1112  {
1113      // NOTE: HandleToObject will either return a pointer to a loaded object or
1114      // will assert. It will never return a non-valid value. This makes it save
1115      // to initialize 'publicArea' with the return value from HandleToObject()
1116      // without checking it first.
1117      OBJECT              *signObject = HandleToObject(keyHandle);
1118      TPMT_PUBLIC         *publicArea = &signObject->publicArea;
1119      TPM_RC               result = TPM_RC_SCHEME;
1120      // The input unmarshaling should prevent any input signature from being
1121      // a NULL signature, but just in case
1122      if(signature->sigAlg == TPM_ALG_NULL)
1123          return TPM_RC_SIGNATURE;
1124      switch(publicArea->type)
1125      {
1126  #ifdef TPM_ALG_RSA
1127          case TPM_ALG_RSA:
1128          {
1129      //
1130              // Call RSA code to verify signature
1131              result = CryptRsaValidateSignature(signature, signObject, digest);
1132              break;
1133          }
1134  #endif //TPM_ALG_RSA
1135  #ifdef TPM_ALG_ECC
1136          case TPM_ALG_ECC:
1137              result = CryptEccValidateSignature(signature, signObject, digest);
1138              break;
1139  #endif // TPM_ALG_ECC
1140          case TPM_ALG_KEYEDHASH:
1141              if(signObject->attributes.publicOnly)
1142                  result = TPM_RCS_HANDLE;
1143              else
1144                  result = CryptHMACVerifySignature(signObject, digest, signature);
1145              break;
1146          default:
1147              break;
1148      }
1149      return result;
1150  }
```


10.2.6.6.16 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().

```
1151  TPM_RC
1152  CryptGetTestResult(
1153      TPM2B_MAX_BUFFER    *outData        // OUT: test result data
1154      )
1155  {
1156      outData->t.size = 0;
1157      return TPM_RC_SUCCESS;
1158  }
```

10.2.6.6.17 CryptIsUniqueSizeValid()

This function validates that the unique values are consistent.

NOTE: This is not a comprehensive test of the public key.

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```
1159  BOOL
1160  CryptIsUniqueSizeValid(
1161      TPMT_PUBLIC     *publicArea     // IN: the public area to check
1162      )
1163  {
1164      BOOL            consistent = FALSE;
1165      UINT16          keySizeInBytes;
1166      switch(publicArea->type)
1167      {
1168  #ifdef TPM_ALG_RSA
1169          case TPM_ALG_RSA:
1170              keySizeInBytes = BITS_TO_BYTES(
1171                                              publicArea->parameters.rsaDetail.keyBits);
1172              consistent = publicArea->unique.rsa.t.size == keySizeInBytes;
1173              break;
1174  #endif //TPM_ALG_RSA
1175  #ifdef TPM_ALG_ECC
1176          case TPM_ALG_ECC:
1177          {
1178              keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(
1179                  publicArea->parameters.eccDetail.curveID));
1180              consistent = keySizeInBytes > 0
1181                  && publicArea->unique.ecc.x.t.size <= keySizeInBytes
1182                  && publicArea->unique.ecc.y.t.size <= keySizeInBytes;
1183          }
1184          break;
1185  #endif //TPM_ALG_ECC
1186          default:
1187              // For SYMCIPHER and KEYDEDHASH objects, the unique field is the size
1188              // of the nameAlg digest.
1189              consistent = publicArea->unique.sym.t.size
1190                  == CryptHashGetDigestSize(publicArea->nameAlg);
1191              break;
1192      }
```

```
1193        return consistent;
1194    }
```

10.2.6.6.18 CryptIsSensitiveSizeValid()

This function is used by TPM2_LoadExternal() to validate that the sensitive area contains a *sensitive* value that is consistent with the values in the public area.

```
1195    BOOL
1196    CryptIsSensitiveSizeValid(
1197        TPMT_PUBLIC             *publicArea,        // IN: the object's public part
1198        TPMT_SENSITIVE          *sensitiveArea      // IN: the object's sensitive part
1199        )
1200    {
1201        BOOL                     consistent;
1202        UINT16                   keySizeInBytes;
1203        switch(publicArea->type)
1204        {
1205    #ifdef TPM_ALG_RSA
1206            case TPM_ALG_RSA:
1207                // sensitive prime value has to be half the size of the public modulus
1208                keySizeInBytes = BITS_TO_BYTES(publicArea->parameters.rsaDetail.keyBits);
1209                consistent = ((sensitiveArea->sensitive.rsa.t.size * 2)
1210                              == keySizeInBytes);
1211                break;
1212    #endif
1213    #ifdef TPM_ALG_ECC
1214            case TPM_ALG_ECC:
1215                keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(
1216                    publicArea->parameters.eccDetail.curveID));
1217                consistent = keySizeInBytes > 0
1218                    && sensitiveArea->sensitive.ecc.t.size == keySizeInBytes;
1219                break;
1220    #endif
1221            case TPM_ALG_SYMCIPHER:
1222                keySizeInBytes = BITS_TO_BYTES(
1223                                    publicArea->parameters.symDetail.sym.keyBits.sym);
1224                consistent = keySizeInBytes == sensitiveArea->sensitive.sym.t.size;
1225                break;
1226            case TPM_ALG_KEYEDHASH:
1227                keySizeInBytes = CryptHashGetBlockSize(publicArea->nameAlg);
1228                // if the block size is 0, then the algorithm is TPM_ALG_NULL and the
1229                // size of the private part is limited to 128. If the algorithm block
1230                // size is over 128 bytes, then the size is limited to 128 bytes for
1231                // interoperability reasons.
1232                if((keySizeInBytes == 0) || (keySizeInBytes > 128))
1233                    keySizeInBytes = 128;
1234                consistent = sensitiveArea->sensitive.bits.t.size <= keySizeInBytes;
1235                break;
1236            default:
1237                consistent = TRUE;
1238                break;
1239        }
1240        return consistent;
1241    }
```

10.2.6.6.19 CryptValidateKeys()

This function is used to verify that the key material of and object is valid. For a *publicOnly* object, the key is verified for size and, if it is an ECC key, it is verified to be on the specified curve. For a key with a sensitive area, the binding between the public and private parts of the key are verified. If the *nameAlg* of the key is TPM_ALG_NULL, then the size of the sensitive area is verified but the public portion is not

verified, unless the key is an RSA key. For an RSA key, the reason for loading the sensitive area is to use it. The only way to use a private RSA key is to compute the private exponent. To compute the private exponent, the public modulus is used.

Error Returns	Meaning
TPM_RC_BINDING	the public and private parts are not cryptographically bound
TPM_RC_HASH	cannot have a *publicOnly* key with *nameAlg* of TPM_ALG_NULL
TPM_RC_KEY	the public unique is not valid
TPM_RC_KEY_SIZE	the private area key is not valid
TPM_RC_TYPE	the types of the sensitive and private parts do not match

```
1242  TPM_RC
1243  CryptValidateKeys(
1244      TPMT_PUBLIC      *publicArea,
1245      TPMT_SENSITIVE   *sensitive,
1246      TPM_RC            blamePublic,
1247      TPM_RC            blameSensitive
1248      )
1249  {
1250      TPM_RC               result;
1251      UINT16               keySizeInBytes;
1252      UINT16               digestSize = CryptHashGetDigestSize(publicArea->nameAlg);
1253      TPMU_PUBLIC_PARMS   *params = &publicArea->parameters;
1254      TPMU_PUBLIC_ID      *unique = &publicArea->unique;
1255      if(sensitive != NULL)
1256      {
1257          // Make sure that the types of the public and sensitive are compatible
1258          if(publicArea->type != sensitive->sensitiveType)
1259              return TPM_RCS_TYPE + blameSensitive;
1260          // Make sure that the authValue is not bigger than allowed
1261          // If there is no name algorithm, then the size just needs to be less than
1262          // the maximum size of the buffer used for authorization. That size check
1263          // was made during unmarshaling of the sensitive area
1264          if((sensitive->authValue.t.size) > digestSize && (digestSize > 0))
1265              return TPM_RCS_SIZE + blameSensitive;
1266      }
1267      switch(publicArea->type)
1268      {
1269  #ifdef TPM_ALG_RSA
1270          case TPM_ALG_RSA:
1271              keySizeInBytes = BITS_TO_BYTES(params->rsaDetail.keyBits);
1272              // Regardless of whether there is a sensitive area, the public modulus
1273              // needs to have the correct size. Otherwise, it can't be used for
1274              // any public key operation nor can it be used to compute the private
1275              // exponent.
1276              // NOTE: This implementation only supports key sizes that are multiples
1277              // of 1024 bits which means that the MSb of the 0th byte will always be
1278              // SET in either a prime or the public modulus.
1279              if((unique->rsa.t.size != keySizeInBytes)
1280                  || (unique->rsa.t.buffer[0] < 0x80))
1281                  return TPM_RCS_KEY + blamePublic;
1282              if(params->rsaDetail.exponent != 0
1283                 && params->rsaDetail.exponent < 7)
1284                  return TPM_RCS_VALUE + blamePublic;
1285              if(sensitive != NULL)
1286              {
1287                  // If there is a sensitive area, it has to be the correct size
1288                  // including having the correct high order bit SET.
1289                  if(((sensitive->sensitive.rsa.t.size * 2) != keySizeInBytes)
1290                     || (sensitive->sensitive.rsa.t.buffer[0] < 0x80))
1291                      return TPM_RCS_KEY_SIZE + blameSensitive;
```

```
1292            }
1293            break;
1294    #endif
1295    #ifdef TPM_ALG_ECC
1296            case TPM_ALG_ECC:
1297            {
1298                TPMI_ECC_CURVE      curveId;
1299                curveId = params->eccDetail.curveID;
1300                keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(curveId));
1301                if(sensitive == NULL)
1302                {
1303                    // Validate the public key size
1304                    if(unique->ecc.x.t.size != keySizeInBytes
1305                       || unique->ecc.y.t.size != keySizeInBytes)
1306                        return TPM_RCS_KEY + blamePublic;
1307                    if(publicArea->nameAlg != TPM_ALG_NULL)
1308                    {
1309                        if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
1310                            return TPM_RCS_ECC_POINT + blamePublic;
1311                    }
1312                }
1313                else
1314                {
1315                    // If the nameAlg is TPM_ALG_NULL, then only verify that the
1316                    // private part of the key is OK.
1317                    if(!CryptEccIsValidPrivateKey(&sensitive->sensitive.ecc,
1318                                                  curveId))
1319                            return TPM_RCS_KEY_SIZE;
1320                    if(publicArea->nameAlg != TPM_ALG_NULL)
1321                    {
1322                    // Full key load, verify that the public point belongs to the
1323                    // private key.
1324                        TPMS_ECC_POINT          toCompare;
1325                        result = CryptEccPointMultiply(&toCompare, curveId, NULL,
1326                                                       &sensitive->sensitive.ecc,
1327                                                       NULL, NULL);
1328                        if(result != TPM_RC_SUCCESS)
1329                            return TPM_RCS_BINDING;
1330                        else
1331                        {
1332                        // Make sure that the private key generated the public key.
1333                        // The input values and the values produced by the point
1334                        // multiply may not be the same size so adjust the computed
1335                        // value to match the size of the input value by adding or
1336                        // removing zeros.
1337                            AdjustNumberB(&toCompare.x.b, unique->ecc.x.t.size);
1338                            AdjustNumberB(&toCompare.y.b, unique->ecc.y.t.size);
1339                            if(!MemoryEqual2B(&unique->ecc.x.b, &toCompare.x.b)
1340                               || !MemoryEqual2B(&unique->ecc.y.b, &toCompare.y.b))
1341                                return TPM_RCS_BINDING;
1342                        }
1343                    }
1344                }
1345                break;
1346            }
1347    #endif
1348            default:
1349                // Checks for SYMCIPHER and KEYEDHASH are largely the same
1350                // If public area has a nameAlg, then validate the public area size
1351                // and if there is also a sensitive area, validate the binding
1352                // For consistency, if the object is public-only just make sure that
1353                // the unique field is consistent with the name algorithm
1354                if(sensitive == NULL)
1355                {
1356                    if(unique->sym.t.size != digestSize)
1357                        return TPM_RCS_KEY + blamePublic;
```

```
1358                }
1359                else
1360                {
1361                    // Make sure that the key size in the sensitive area is consistent.
1362                    if(publicArea->type == TPM_ALG_SYMCIPHER)
1363                    {
1364                        result = CryptSymKeyValidate(&params->symDetail.sym,
1365                                                     &sensitive->sensitive.sym);
1366                        if(result != TPM_RC_SUCCESS)
1367                            return result + blameSensitive;
1368                    }
1369                    else
1370                    {
1371                        // For a keyed hash object, the key has to be less than the
1372                        // smaller of the block size of the hash used in the scheme or
1373                        // 128 bytes. The worst case value is limited by the
1374                        // unmarshaling code so the only thing left to be checked is
1375                        // that it does not exceed the block size of the hash.
1376                        // by the hash algorithm of the scheme.
1377                        TPMT_KEYEDHASH_SCHEME       *scheme;
1378                        TPM_ALG_ID                   hashAlg;
1379                        scheme = &params->keyedHashDetail.scheme;
1380                        if(scheme->scheme == TPM_ALG_XOR)
1381                            hashAlg = scheme->details.xor.hashAlg;
1382                        else if(scheme->scheme == TPM_ALG_HMAC)
1383                            hashAlg = scheme->details.hmac.hashAlg;
1384                        else if(scheme->scheme == TPM_ALG_NULL)
1385                            hashAlg = publicArea->nameAlg;
1386                        else
1387                            return TPM_RCS_SCHEME + blamePublic;
1388                        if(sensitive->sensitive.bits.t.size
1389                            > CryptHashGetBlockSize(hashAlg))
1390                            return TPM_RCS_KEY_SIZE + blameSensitive;
1391                    }
1392                    // If there is a nameAlg, check the binding
1393                    if(publicArea->nameAlg != TPM_ALG_NULL)
1394                    {
1395                        TPM2B_DIGEST            compare;
1396                        if(sensitive->seedValue.t.size != digestSize)
1397                            return TPM_RCS_KEY_SIZE;
1398                        CryptComputeSymmetricUnique(publicArea, sensitive, &compare);
1399                        if(!MemoryEqual2B(&unique->sym.b, &compare.b))
1400                            return TPM_RC_BINDING;
1401                    }
1402                }
1403                break;
1404        }
1405        // For a parent, need to check that the seedValue is the correct size for
1406        // protections. It should be at least half the size of the nameAlg
1407        if(publicArea->objectAttributes.restricted
1408           && publicArea->objectAttributes.decrypt
1409           && sensitive != NULL
1410           && publicArea->nameAlg != TPM_ALG_NULL)
1411        {
1412            if((sensitive->seedValue.t.size < (digestSize / 2))
1413               || (sensitive->seedValue.t.size > digestSize))
1414                return TPM_RCS_SIZE + blameSensitive;
1415        }
1416        return TPM_RC_SUCCESS;
1417    }
```


10.2.6.6.20 CryptAlgSetImplemented()

This function initializes the bit vector with one bit for each implemented algorithm. This function is called from _TPM_Init(). The vector of implemented algorithms should be generated by the part 2 parser so that the *g_implementedAlgorithms* vector can be a constant. That's not how it is now

```
1418  void
1419  CryptAlgsSetImplemented(
1420      void
1421      )
1422  {
1423      AlgorithmGetImplementedVector(&g_implementedAlgorithms);
1424  }
```

10.2.7 CryptSelfTest.c

10.2.7.1 Introduction

The functions in this file are designed to support self-test of cryptographic functions in the TPM. The TPM allows the user to decide whether to run self-test on a demand basis or to run all the self-tests before proceeding.

The self-tests are controlled by a set of bit vectors. The **g_untestedDecryptionAlgorithms** vector has a bit for each decryption algorithm that needs to be tested and **g_untestedEncryptionAlgorithms** has a bit for each encryption algorithm that needs to be tested. Before an algorithm is used, the appropriate vector is checked (indexed using the algorithm ID). If the bit is 1, then the test function should be called.

For more information, see TpmSelfTests().txt

```
1   #include "Tpm.h"
```

10.2.7.2 Functions

10.2.7.2.1 RunSelfTest()

Local function to run self-test

```
2   static TPM_RC
3   CryptRunSelfTests(
4       ALGORITHM_VECTOR    *toTest         // IN: the vector of the algorithms to test
5       )
6   {
7       TPM_ALG_ID           alg;
8       // For each of the algorithms that are in the toTestVecor, need to run a
9       // test
10      for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
11      {
12          if(TEST_BIT(alg, *toTest))
13          {
14              TPM_RC          result = CryptTestAlgorithm(alg, toTest);
15              if(result != TPM_RC_SUCCESS)
16                  return result;
17          }
18      }
19      return TPM_RC_SUCCESS;
20  }
```


10.2.7.2.2 CryptSelfTest()

This function is called to start/complete a full self-test. If *fullTest* is NO, then only the untested algorithms will be run. If *fullTest* is YES, then *g_untestedDecryptionAlgorithms* is reinitialized and then all tests are run. This implementation of the reference design does not support processing outside the framework of a TPM command. As a consequence, this command does not complete until all tests are done. Since this can take a long time, the TPM will check after each test to see if the command is canceled. If so, then the TPM will returned TPM_RC_CANCELLED. To continue with the self-tests, call TPM2_SelfTest(*fullTest* == No) and the TPM will complete the testing.

Error Returns	Meaning
TPM_RC_CANCELED	if the command is canceled

```
21  LIB_EXPORT
22  TPM_RC
23  CryptSelfTest(
24      TPMI_YES_NO      fullTest       // IN: if full test is required
25      )
26  {
27  #ifdef SIMULATION
28      if(g_forceFailureMode)
29          FAIL(FATAL_ERROR_FORCED);
30  #endif
31      // If the caller requested a full test, then reset the to test vector so that
32      // all the tests will be run
33      if(fullTest == YES)
34      {
35          MemoryCopy(g_toTest,
36                     g_implementedAlgorithms,
37                     sizeof(g_toTest));
38      }
39      return CryptRunSelfTests(&g_toTest);
40  }
```

10.2.7.2.3 CryptIncrementalSelfTest()

This function is used to perform an incremental self-test. This implementation will perform the *toTest* values before returning. That is, it assumes that the TPM cannot perform background tasks between commands.

This command may be canceled. If it is, then there is no return result. However, this command can be run again and the incremental progress will not be lost.

Error Returns	Meaning
TPM_RC_CANCELED	processing of this command was canceled
TPM_RC_TESTING	if *toTest* list is not empty
TPM_RC_VALUE	an algorithm in the *toTest* list is not implemented

```
41  TPM_RC
42  CryptIncrementalSelfTest(
43      TPML_ALG            *toTest,        // IN: list of algorithms to be tested
44      TPML_ALG            *toDoList       // OUT: list of algorithms needing test
45      )
46  {
47      ALGORITHM_VECTOR     toTestVector = {0};
48      TPM_ALG_ID           alg;
49      UINT32               i;
50      pAssert(toTest != NULL && toDoList != NULL);
```



```
51      if(toTest->count > 0)
52      {
53          // Transcribe the toTest list into the toTestVector
54          for(i = 0; i < toTest->count; i++)
55          {
56              alg = toTest->algorithms[i];
57              // make sure that the algorithm value is not out of range
58              if((alg > TPM_ALG_LAST) || !TEST_BIT(alg, g_implementedAlgorithms))
59                  return TPM_RC_VALUE;
60              SET_BIT(alg, toTestVector);
61          }
62          // Run the test
63          if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
64              return TPM_RC_CANCELED;
65      }
66      // Fill in the toDoList with the algorithms that are still untested
67      toDoList->count = 0;
68      for(alg = TPM_ALG_FIRST;
69      toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
70          alg++)
71      {
72          if(TEST_BIT(alg, g_toTest))
73              toDoList->algorithms[toDoList->count++] = alg;
74      }
75      return TPM_RC_SUCCESS;
76  }
```

10.2.7.2.4 CryptInitializeToTest()

This function will initialize the data structures for testing all the algorithms. This should not be called unless CryptAlgsSetImplemented() has been called

```
77  void
78  CryptInitializeToTest(
79      void
80      )
81  {
82      // Indicate that nothing has been tested
83      memset(&g_cryptoSelfTestState, 0, sizeof(g_cryptoSelfTestState));
84      // Copy the implemented algorithm vector
85      MemoryCopy(g_toTest, g_implementedAlgorithms, sizeof(g_toTest));
86      // Setting the algorithm to null causes the test function to just clear
87      // out any algorithms for which there is no test.
88      CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);
89      return;
90  }
```

10.2.7.2.5 CryptTestAlgorithm()

Only point of contact with the actual self tests. If a self-test fails, there is no return and the TPM goes into failure mode. The call to TestAlgorithm() uses an algorithms selector and a bit vector. When the test is run, the corresponding bit in *toTest* and in *g_toTest* is CLEAR. If *toTest* is NULL, then only the bit in *g_toTest* is CLEAR. There is a special case for the call to TestAlgorithm(). When *alg* is TPM_ALG_ERROR, TestAlgorithm() will CLEAR any bit in *toTest* for which it has no test. This allows the knowledge about which algorithms have test to be accessed through the interface that provides the test.

Error Returns	Meaning
TPM_RC_SUCCESS	test complete
TPM_RC_CANCELED	test was canceled

```
 91  LIB_EXPORT
 92  TPM_RC
 93  CryptTestAlgorithm(
 94      TPM_ALG_ID           alg,
 95      ALGORITHM_VECTOR    *toTest
 96      )
 97  {
 98      TPM_RC                   result;
 99  #if defined SELF_TEST
100      result = TestAlgorithm(alg, toTest);
101  #else
102      // If this is an attempt to determine the algorithms for which there is a
103      // self test, pretend that all of them do. We do that by not clearing any
104      // of the algorithm bits. When/if this function is called to run tests, it
105      // will over report. This can be changed so that any call to check on which
106      // algorithms have tests, 'toTest' can be cleared.
107      if(alg != TPM_ALG_ERROR)
108      {
109          CLEAR_BIT(alg, g_toTest);
110          if(toTest != NULL)
111              CLEAR_BIT(alg, *toTest);
112      }
113      result = TPM_RC_SUCCESS;
114  #endif
115      return result;
116  }
```

10.2.8 CryptDataEcc.c

```
 1  #include    "Tpm.h"
 2  #ifdef TPM_ALG_ECC
```

Defines for the sizes of ECC parameters

```
 3  TPM2B_BYTE_VALUE(1);
 4  TPM2B_BYTE_VALUE(16);
 5  TPM2B_BYTE_VALUE(2);
 6  TPM2B_BYTE_VALUE(24);
 7  TPM2B_BYTE_VALUE(28);
 8  TPM2B_BYTE_VALUE(32);
 9  TPM2B_BYTE_VALUE(4);
10  TPM2B_BYTE_VALUE(48);
11  TPM2B_BYTE_VALUE(64);
12  TPM2B_BYTE_VALUE(66);
13  TPM2B_BYTE_VALUE(8);
14  TPM2B_BYTE_VALUE(80);
15  #if defined ECC_NIST_P192 && ECC_NIST_P192 == YES
16  const TPM2B_24_BYTE_VALUE NIST_P192_p = {24,
17          {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
18           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
19           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
20  const TPM2B_24_BYTE_VALUE NIST_P192_a = {24,
21          {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
22           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
23           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC}};
24  const TPM2B_24_BYTE_VALUE NIST_P192_b = {24,
25          {0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7,
26           0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49,
```

```
27            0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1}};
28    const TPM2B_24_BYTE_VALUE NIST_P192_gX = {24,
29           {0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6,
30            0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00,
31            0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12}};
32    const TPM2B_24_BYTE_VALUE NIST_P192_gY = {24,
33           {0x07, 0x19, 0x2B, 0x95, 0xFF, 0xC8, 0xDA, 0x78,
34            0x63, 0x10, 0x11, 0xED, 0x6B, 0x24, 0xCD, 0xD5,
35            0x73, 0xF9, 0x77, 0xA1, 0x1E, 0x79, 0x48, 0x11}};
36    const TPM2B_24_BYTE_VALUE NIST_P192_n = {24,
37           {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
38            0xFF, 0xFF, 0xFF, 0xFF, 0x99, 0xDE, 0xF8, 0x36,
39            0x14, 0x6B, 0xC9, 0xB1, 0xB4, 0xD2, 0x28, 0x31}};
40    const TPM2B_1_BYTE_VALUE NIST_P192_h = {1,{1}};
41    const ECC_CURVE_DATA NIST_P192 = {&NIST_P192_p.b, &NIST_P192_a.b, &NIST_P192_b.b,
42                                      &NIST_P192_gX.b, &NIST_P192_gY.b, &NIST_P192_n.b,
43                                      &NIST_P192_h.b};
44    #endif // ECC_NIST_P192
45    #if defined ECC_NIST_P224 && ECC_NIST_P224 == YES
46    const TPM2B_28_BYTE_VALUE NIST_P224_p = {28,
47           {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
48            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
49            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
50            0x00, 0x00, 0x00, 0x01}};
51    const TPM2B_28_BYTE_VALUE NIST_P224_a = {28,
52           {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
53            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
54            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
55            0xFF, 0xFF, 0xFF, 0xFE}};
56    const TPM2B_28_BYTE_VALUE NIST_P224_b = {28,
57           {0xB4, 0x05, 0x0A, 0x85, 0x0C, 0x04, 0xB3, 0xAB,
58            0xF5, 0x41, 0x32, 0x56, 0x50, 0x44, 0xB0, 0xB7,
59            0xD7, 0xBF, 0xD8, 0xBA, 0x27, 0x0B, 0x39, 0x43,
60            0x23, 0x55, 0xFF, 0xB4}};
61    const TPM2B_28_BYTE_VALUE NIST_P224_gX = {28,
62           {0xB7, 0x0E, 0x0C, 0xBD, 0x6B, 0xB4, 0xBF, 0x7F,
63            0x32, 0x13, 0x90, 0xB9, 0x4A, 0x03, 0xC1, 0xD3,
64            0x56, 0xC2, 0x11, 0x22, 0x34, 0x32, 0x80, 0xD6,
65            0x11, 0x5C, 0x1D, 0x21}};
66    const TPM2B_28_BYTE_VALUE NIST_P224_gY = {28,
67           {0xBD, 0x37, 0x63, 0x88, 0xB5, 0xF7, 0x23, 0xFB,
68            0x4C, 0x22, 0xDF, 0xE6, 0xCD, 0x43, 0x75, 0xA0,
69            0x5A, 0x07, 0x47, 0x64, 0x44, 0xD5, 0x81, 0x99,
70            0x85, 0x00, 0x7E, 0x34}};
71    const TPM2B_28_BYTE_VALUE NIST_P224_n = {28,
72           {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
73            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x16, 0xA2,
74            0xE0, 0xB8, 0xF0, 0x3E, 0x13, 0xDD, 0x29, 0x45,
75            0x5C, 0x5C, 0x2A, 0x3D}};
76    const TPM2B_1_BYTE_VALUE NIST_P224_h = {1,{1}};
77    const ECC_CURVE_DATA NIST_P224 = {&NIST_P224_p.b, &NIST_P224_a.b, &NIST_P224_b.b,
78                                      &NIST_P224_gX.b, &NIST_P224_gY.b, &NIST_P224_n.b,
79                                      &NIST_P224_h.b};
80    #endif // ECC_NIST_P224
81    #if defined ECC_NIST_P256 && ECC_NIST_P256 == YES
82    const TPM2B_32_BYTE_VALUE NIST_P256_p = {32,
83           {0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01,
84            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
85            0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
86            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
87    const TPM2B_32_BYTE_VALUE NIST_P256_a = {32,
88           {0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01,
89            0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
90            0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
91            0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC}};
92    const TPM2B_32_BYTE_VALUE NIST_P256_b = {32,
```

```
 93          {0x5A, 0xC6, 0x35, 0xD8, 0xAA, 0x3A, 0x93, 0xE7,
 94           0xB3, 0xEB, 0xBD, 0x55, 0x76, 0x98, 0x86, 0xBC,
 95           0x65, 0x1D, 0x06, 0xB0, 0xCC, 0x53, 0xB0, 0xF6,
 96           0x3B, 0xCE, 0x3C, 0x3E, 0x27, 0xD2, 0x60, 0x4B}};
 97   const TPM2B_32_BYTE_VALUE NIST_P256_gX = {32,
 98          {0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47,
 99           0xF8, 0xBC, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2,
100           0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0,
101           0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96}};
102   const TPM2B_32_BYTE_VALUE NIST_P256_gY = {32,
103          {0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B,
104           0x8E, 0xE7, 0xEB, 0x4A, 0x7C, 0x0F, 0x9E, 0x16,
105           0x2B, 0xCE, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE,
106           0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5}};
107   const TPM2B_32_BYTE_VALUE NIST_P256_n = {32,
108          {0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
109           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
110           0xBC, 0xE6, 0xFA, 0xAD, 0xA7, 0x17, 0x9E, 0x84,
111           0xF3, 0xB9, 0xCA, 0xC2, 0xFC, 0x63, 0x25, 0x51}};
112   const TPM2B_1_BYTE_VALUE NIST_P256_h = {1,{1}};
113   const ECC_CURVE_DATA NIST_P256 = {&NIST_P256_p.b, &NIST_P256_a.b, &NIST_P256_b.b,
114                                     &NIST_P256_gX.b, &NIST_P256_gY.b, &NIST_P256_n.b,
115                                     &NIST_P256_h.b};
116   #endif // ECC_NIST_P256
117   #if defined ECC_NIST_P384 && ECC_NIST_P384 == YES
118   const TPM2B_48_BYTE_VALUE NIST_P384_p = {48,
119          {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
120           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
121           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
122           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
123           0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
124           0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF}};
125   const TPM2B_48_BYTE_VALUE NIST_P384_a = {48,
126          {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
127           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
128           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
129           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE,
130           0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
131           0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFC}};
132   const TPM2B_48_BYTE_VALUE NIST_P384_b = {48,
133          {0xB3, 0x31, 0x2F, 0xA7, 0xE2, 0x3E, 0xE7, 0xE4,
134           0x98, 0x8E, 0x05, 0x6B, 0xE3, 0xF8, 0x2D, 0x19,
135           0x18, 0x1D, 0x9C, 0x6E, 0xFE, 0x81, 0x41, 0x12,
136           0x03, 0x14, 0x08, 0x8F, 0x50, 0x13, 0x87, 0x5A,
137           0xC6, 0x56, 0x39, 0x8D, 0x8A, 0x2E, 0xD1, 0x9D,
138           0x2A, 0x85, 0xC8, 0xED, 0xD3, 0xEC, 0x2A, 0xEF}};
139   const TPM2B_48_BYTE_VALUE NIST_P384_gX = {48,
140          {0xAA, 0x87, 0xCA, 0x22, 0xBE, 0x8B, 0x05, 0x37,
141           0x8E, 0xB1, 0xC7, 0x1E, 0xF3, 0x20, 0xAD, 0x74,
142           0x6E, 0x1D, 0x3B, 0x62, 0x8B, 0xA7, 0x9B, 0x98,
143           0x59, 0xF7, 0x41, 0xE0, 0x82, 0x54, 0x2A, 0x38,
144           0x55, 0x02, 0xF2, 0x5D, 0xBF, 0x55, 0x29, 0x6C,
145           0x3A, 0x54, 0x5E, 0x38, 0x72, 0x76, 0x0A, 0xB7}};
146   const TPM2B_48_BYTE_VALUE NIST_P384_gY = {48,
147          {0x36, 0x17, 0xDE, 0x4A, 0x96, 0x26, 0x2C, 0x6F,
148           0x5D, 0x9E, 0x98, 0xBF, 0x92, 0x92, 0xDC, 0x29,
149           0xF8, 0xF4, 0x1D, 0xBD, 0x28, 0x9A, 0x14, 0x7C,
150           0xE9, 0xDA, 0x31, 0x13, 0xB5, 0xF0, 0xB8, 0xC0,
151           0x0A, 0x60, 0xB1, 0xCE, 0x1D, 0x7E, 0x81, 0x9D,
152           0x7A, 0x43, 0x1D, 0x7C, 0x90, 0xEA, 0x0E, 0x5F}};
153   const TPM2B_48_BYTE_VALUE NIST_P384_n = {48,
154          {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
155           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
156           0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
157           0xC7, 0x63, 0x4D, 0x81, 0xF4, 0x37, 0x2D, 0xDF,
158           0x58, 0x1A, 0x0D, 0xB2, 0x48, 0xB0, 0xA7, 0x7A,
```



```
159             0xEC, 0xEC, 0x19, 0x6A, 0xCC, 0xC5, 0x29, 0x73}};
160     const TPM2B_1_BYTE_VALUE NIST_P384_h = {1,{1}};
161     const ECC_CURVE_DATA NIST_P384 = {&NIST_P384_p.b, &NIST_P384_a.b, &NIST_P384_b.b,
162                                       &NIST_P384_gX.b, &NIST_P384_gY.b, &NIST_P384_n.b,
163                                       &NIST_P384_h.b};
164     #endif // ECC_NIST_P384
165     #if defined ECC_NIST_P521 && ECC_NIST_P521 == YES
166     const TPM2B_66_BYTE_VALUE NIST_P521_p = {66,
167             {0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
168              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
169              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
170              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
171              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
172              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
173              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
174              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
175              0xFF, 0xFF}};
176     const TPM2B_66_BYTE_VALUE NIST_P521_a = {66,
177             {0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
178              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
179              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
180              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
181              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
182              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
183              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
184              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
185              0xFF, 0xFC}};
186     const TPM2B_66_BYTE_VALUE NIST_P521_b = {66,
187             {0x00, 0x51, 0x95, 0x3E, 0xB9, 0x61, 0x8E, 0x1C,
188              0x9A, 0x1F, 0x92, 0x9A, 0x21, 0xA0, 0xB6, 0x85,
189              0x40, 0xEE, 0xA2, 0xDA, 0x72, 0x5B, 0x99, 0xB3,
190              0x15, 0xF3, 0xB8, 0xB4, 0x89, 0x91, 0x8E, 0xF1,
191              0x09, 0xE1, 0x56, 0x19, 0x39, 0x51, 0xEC, 0x7E,
192              0x93, 0x7B, 0x16, 0x52, 0xC0, 0xBD, 0x3B, 0xB1,
193              0xBF, 0x07, 0x35, 0x73, 0xDF, 0x88, 0x3D, 0x2C,
194              0x34, 0xF1, 0xEF, 0x45, 0x1F, 0xD4, 0x6B, 0x50,
195              0x3F, 0x00}};
196     const TPM2B_66_BYTE_VALUE NIST_P521_gX = {66,
197             {0x00, 0xC6, 0x85, 0x8E, 0x06, 0xB7, 0x04, 0x04,
198              0xE9, 0xCD, 0x9E, 0x3E, 0xCB, 0x66, 0x23, 0x95,
199              0xB4, 0x42, 0x9C, 0x64, 0x81, 0x39, 0x05, 0x3F,
200              0xB5, 0x21, 0xF8, 0x28, 0xAF, 0x60, 0x6B, 0x4D,
201              0x3D, 0xBA, 0xA1, 0x4B, 0x5E, 0x77, 0xEF, 0xE7,
202              0x59, 0x28, 0xFE, 0x1D, 0xC1, 0x27, 0xA2, 0xFF,
203              0xA8, 0xDE, 0x33, 0x48, 0xB3, 0xC1, 0x85, 0x6A,
204              0x42, 0x9B, 0xF9, 0x7E, 0x7E, 0x31, 0xC2, 0xE5,
205              0xBD, 0x66}};
206     const TPM2B_66_BYTE_VALUE NIST_P521_gY = {66,
207             {0x01, 0x18, 0x39, 0x29, 0x6A, 0x78, 0x9A, 0x3B,
208              0xC0, 0x04, 0x5C, 0x8A, 0x5F, 0xB4, 0x2C, 0x7D,
209              0x1B, 0xD9, 0x98, 0xF5, 0x44, 0x49, 0x57, 0x9B,
210              0x44, 0x68, 0x17, 0xAF, 0xBD, 0x17, 0x27, 0x3E,
211              0x66, 0x2C, 0x97, 0xEE, 0x72, 0x99, 0x5E, 0xF4,
212              0x26, 0x40, 0xC5, 0x50, 0xB9, 0x01, 0x3F, 0xAD,
213              0x07, 0x61, 0x35, 0x3C, 0x70, 0x86, 0xA2, 0x72,
214              0xC2, 0x40, 0x88, 0xBE, 0x94, 0x76, 0x9F, 0xD1,
215              0x66, 0x50}};
216     const TPM2B_66_BYTE_VALUE NIST_P521_n = {66,
217             {0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
218              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
219              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
220              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
221              0xFF, 0xFA, 0x51, 0x86, 0x87, 0x83, 0xBF, 0x2F,
222              0x96, 0x6B, 0x7F, 0xCC, 0x01, 0x48, 0xF7, 0x09,
223              0xA5, 0xD0, 0x3B, 0xB5, 0xC9, 0xB8, 0x89, 0x9C,
224              0x47, 0xAE, 0xBB, 0x6F, 0xB7, 0x1E, 0x91, 0x38,
```



```
225             0x64, 0x09}};
226     const TPM2B_1_BYTE_VALUE NIST_P521_h = {1,{1}};
227     const ECC_CURVE_DATA NIST_P521 = {&NIST_P521_p.b, &NIST_P521_a.b, &NIST_P521_b.b,
228                                       &NIST_P521_gX.b, &NIST_P521_gY.b, &NIST_P521_n.b,
229                                       &NIST_P521_h.b};
230     #endif // ECC_NIST_P521
231     #if defined ECC_BN_P256 && ECC_BN_P256 == YES
232     const TPM2B_32_BYTE_VALUE BN_P256_p = {32,
233             {0xFF, 0XFF, 0XFF, 0XFF, 0XFF, 0XFC, 0XF0, 0XCD,
234              0X46, 0XE5, 0XF2, 0X5E, 0XEE, 0X71, 0XA4, 0X9F,
235              0X0C, 0XDC, 0X65, 0XFB, 0X12, 0X98, 0X0A, 0X82,
236              0XD3, 0X29, 0X2D, 0XDB, 0XAE, 0XD3, 0X30, 0X13}};
237     const TPM2B_1_BYTE_VALUE BN_P256_a = {1,{0}};
238     const TPM2B_1_BYTE_VALUE BN_P256_b = {1,{3}};
239     const TPM2B_1_BYTE_VALUE BN_P256_gX = {1,{1}};
240     const TPM2B_1_BYTE_VALUE BN_P256_gY = {1,{2}};;
241     const TPM2B_32_BYTE_VALUE BN_P256_n = {32,
242             {0xFF, 0XFF, 0XFF, 0XFF, 0XFF, 0XFC, 0XF0, 0XCD,
243              0X46, 0XE5, 0XF2, 0X5E, 0XEE, 0X71, 0XA4, 0X9E,
244              0X0C, 0XDC, 0X65, 0XFB, 0X12, 0X99, 0X92, 0X1A,
245              0XF6, 0X2D, 0X53, 0X6C, 0XD1, 0X0B, 0X50, 0X0D}};
246     const TPM2B_1_BYTE_VALUE BN_P256_h = {1,{1}};
247     const ECC_CURVE_DATA BN_P256 = {&BN_P256_p.b, &BN_P256_a.b, &BN_P256_b.b,
248                                     &BN_P256_gX.b, &BN_P256_gY.b, &BN_P256_n.b,
249                                     &BN_P256_h.b};
250     #endif // ECC_BN_P256
251     #if defined ECC_BN_P638 && ECC_BN_P638 == YES
252     const TPM2B_80_BYTE_VALUE BN_P638_p = {80,
253             {0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D,
254              0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3,
255              0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E,
256              0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F,
257              0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55,
258              0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B,
259              0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80,
260              0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD,
261              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0,
262              0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67}};
263     const TPM2B_1_BYTE_VALUE BN_P638_a = {1,{0}};
264     const TPM2B_2_BYTE_VALUE BN_P638_b = {2,{0x01,0x01}};
265     const TPM2B_80_BYTE_VALUE BN_P638_gX = {80,
266             {0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D,
267              0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3,
268              0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E,
269              0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F,
270              0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55,
271              0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B,
272              0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80,
273              0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD,
274              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0,
275              0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66}};
276     const TPM2B_1_BYTE_VALUE BN_P638_gY = {1,{0x10}};
277     const TPM2B_80_BYTE_VALUE BN_P638_n = {80,
278             {0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D,
279              0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3,
280              0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E,
281              0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F,
282              0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55,
283              0x60, 0x00, 0x86, 0x55, 0x00, 0x21, 0xE5, 0x55,
284              0xFF, 0xFF, 0xF5, 0x4F, 0xFF, 0xF4, 0xEA, 0xC0,
285              0x00, 0x00, 0x00, 0x49, 0x80, 0x01, 0x54, 0xD9,
286              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xA0,
287              0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61}};
288     const TPM2B_1_BYTE_VALUE BN_P638_h = {1,{1}};
289     const ECC_CURVE_DATA BN_P638 = {&BN_P638_p.b, &BN_P638_a.b, &BN_P638_b.b,
290                                     &BN_P638_gX.b, &BN_P638_gY.b, &BN_P638_n.b,
```

```
291                                     &BN_P638_h.b};
292     #endif // ECC_BN_P638
293     #if defined ECC_SM2_P256 && ECC_SM2_P256 == YES
294     const TPM2B_32_BYTE_VALUE SM2_P256_p = {32,
295             {0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF,
296              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
297              0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
298              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF}};
299     const TPM2B_32_BYTE_VALUE SM2_P256_a = {32,
300             {0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF,
301              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
302              0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00,
303              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC}};
304     const TPM2B_32_BYTE_VALUE SM2_P256_b = {32,
305             {0x28, 0xE9, 0xFA, 0x9E, 0x9D, 0x9F, 0x5E, 0x34,
306              0x4D, 0x5A, 0x9E, 0x4B, 0xCF, 0x65, 0x09, 0xA7,
307              0xF3, 0x97, 0x89, 0xF5, 0x15, 0xAB, 0x8F, 0x92,
308              0xDD, 0xBC, 0xBD, 0x41, 0x4D, 0x94, 0x0E, 0x93}};
309     const TPM2B_32_BYTE_VALUE SM2_P256_gX = {32,
310             {0x32, 0xC4, 0xAE, 0x2C, 0x1F, 0x19, 0x81, 0x19,
311              0x5F, 0x99, 0x04, 0x46, 0x6A, 0x39, 0xC9, 0x94,
312              0x8F, 0xE3, 0x0B, 0xBF, 0xF2, 0x66, 0x0B, 0xE1,
313              0x71, 0x5A, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7}};
314     const TPM2B_32_BYTE_VALUE SM2_P256_gY = {32,
315             {0xBC, 0x37, 0x36, 0xA2, 0xF4, 0xF6, 0x77, 0x9C,
316              0x59, 0xBD, 0xCE, 0xE3, 0x6B, 0x69, 0x21, 0x53,
317              0xD0, 0xA9, 0x87, 0x7C, 0xC6, 0x2A, 0x47, 0x40,
318              0x02, 0xDF, 0x32, 0xE5, 0x21, 0x39, 0xF0, 0xA0}};
319     const TPM2B_32_BYTE_VALUE SM2_P256_n = {32,
320             {0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF,
321              0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
322              0x72, 0x03, 0xDF, 0x6B, 0x21, 0xC6, 0x05, 0x2B,
323              0x53, 0xBB, 0xF4, 0x09, 0x39, 0xD5, 0x41, 0x23}};
324     const TPM2B_1_BYTE_VALUE SM2_P256_h = {1,{1}};
325     const ECC_CURVE_DATA SM2_P256 = {&SM2_P256_p.b, &SM2_P256_a.b, &SM2_P256_b.b,
326                                      &SM2_P256_gX.b, &SM2_P256_gY.b, &SM2_P256_n.b,
327                                      &SM2_P256_h.b};
328     #endif // ECC_SM2_P256
329     #define comma
330     const ECC_CURVE   eccCurves[] = {
331     #if defined ECC_NIST_P192 && ECC_NIST_P192 == YES
332         comma
333         {TPM_ECC_NIST_P192,
334         192,
335         {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA256},
336         {TPM_ALG_NULL,TPM_ALG_NULL},
337         &NIST_P192 CURVE_NAME("NIST_P192")}
338     #   undef comma
339     #   define comma ,
340     #endif // ECC_NIST_P192
341     #if defined ECC_NIST_P224 && ECC_NIST_P224 == YES
342         comma
343         {TPM_ECC_NIST_P224,
344         224,
345         {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA256},
346         {TPM_ALG_NULL,TPM_ALG_NULL},
347         &NIST_P224 CURVE_NAME("NIST_P224")}
348     #   undef comma
349     #   define comma ,
350     #endif // ECC_NIST_P224
351     #if defined ECC_NIST_P256 && ECC_NIST_P256 == YES
352         comma
353         {TPM_ECC_NIST_P256,
354         256,
355         {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA256},
356         {TPM_ALG_NULL,TPM_ALG_NULL},
```

```
357         &NIST_P256 CURVE_NAME("NIST_P256")}
358     #   undef comma
359     #   define comma ,
360     #endif // ECC_NIST_P256
361     #if defined ECC_NIST_P384 && ECC_NIST_P384 == YES
362         comma
363         {TPM_ECC_NIST_P384,
364         384,
365         {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA384},
366         {TPM_ALG_NULL,TPM_ALG_NULL},
367         &NIST_P384 CURVE_NAME("NIST_P384")}
368     #   undef comma
369     #   define comma ,
370     #endif // ECC_NIST_P384
371     #if defined ECC_NIST_P521 && ECC_NIST_P521 == YES
372         comma
373         {TPM_ECC_NIST_P521,
374         521,
375         {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SHA512},
376         {TPM_ALG_NULL,TPM_ALG_NULL},
377         &NIST_P521 CURVE_NAME("NIST_P521")}
378     #   undef comma
379     #   define comma ,
380     #endif // ECC_NIST_P521
381     #if defined ECC_BN_P256 && ECC_BN_P256 == YES
382         comma
383         {TPM_ECC_BN_P256,
384         256,
385         {TPM_ALG_NULL,TPM_ALG_NULL},
386         {TPM_ALG_NULL,TPM_ALG_NULL},
387         &BN_P256 CURVE_NAME("BN_P256")}
388     #   undef comma
389     #   define comma ,
390     #endif // ECC_BN_P256
391     #if defined ECC_BN_P638 && ECC_BN_P638 == YES
392         comma
393         {TPM_ECC_BN_P638,
394         638,
395         {TPM_ALG_NULL,TPM_ALG_NULL},
396         {TPM_ALG_NULL,TPM_ALG_NULL},
397         &BN_P638 CURVE_NAME("BN_P638")}
398     #   undef comma
399     #   define comma ,
400     #endif // ECC_BN_P638
401     #if defined ECC_SM2_P256 && ECC_SM2_P256 == YES
402         comma
403         {TPM_ECC_SM2_P256,
404         256,
405         {TPM_ALG_KDF1_SP800_56A,TPM_ALG_SM3_256},
406         {TPM_ALG_NULL,TPM_ALG_NULL},
407         &SM2_P256 CURVE_NAME("SM2_P256")}
408     #   undef comma
409     #   define comma ,
410     #endif // ECC_SM2_P256
411     };
412     #endif // TPM_ALG_ECC
```

10.2.9 CryptDes.c

10.2.9.1 Introduction

This file contains the extra functions required for TDES.

10.2.9.2 Includes, Defines, and Typedefs

```
#include "Tpm.h"
#ifdef TPM_ALG_TDES
#define DES_NUM_WEAK 64
const UINT64 DesWeakKeys[DES_NUM_WEAK] = {
    0x0101010101010101, 0xFEFEFEFEFEFEFEFE, 0xE0E0E0E0F1F1F1F1, 0x1F1F1F1F0E0E0E0E,
    0x011F011F010E010E, 0x1F011F010E010E01, 0x01E001E001F101F1, 0xE001E001F101F101,
    0x01FE01FE01FE01FE, 0xFE01FE01FE01FE01, 0x1FE01FE00EF10EF1, 0xE01FE01FF10EF10E,
    0x1FFE1FFE0EFE0EFE, 0xFE1FFE1FFE0EFE0E, 0xE0FEE0FEF1FEF1FE, 0xFEE0FEE0FEF1FEF1,
    0x01011F1F01010E0E, 0x1F1F01010E0E0101, 0xE0E01F1FF1F10E0E, 0x0101E0E00101F1F1,
    0x1F1FE0E00E0EF1F1, 0xE0E0FEFEF1F1FEFE, 0x0101FEFE0101FEFE, 0x1F1FFEFE0E0EFEFE,
    0xE0FE011FF1FE010E, 0x011F1F01010E0E01, 0x1FE001FE0EF101FE, 0xE0FE1F01F1FE0E01,
    0x011FE0FE010EF1FE, 0x1FE0E01F0EF1F10E, 0xE0FEFEE0F1FEFEF1, 0x011FFEE0010EFEF1,
    0x1FE0FE010EF1FE01, 0xFE0101FEFE0101FE, 0x01E01FFE01F10EFE, 0x1FFE01E00EFE01F1,
    0xFE011FE0FE010EF1, 0xFE01E01FFE01F10E, 0x1FFEE0010EFEF101, 0xFE1F01E0FE0E01F1,
    0x01E0E00101F1F101, 0x1FFEFE1F0EFEFE0E, 0xFE1FE001FE0EF101, 0x01E0FE1F01F1FE0E,
    0xE00101E0F10101F1, 0xFE1F1FFEFE0E0EFE, 0x01FE1FE001FE0EF1, 0xE0011FFEF1010EFE,
    0xFEE0011FFEF1010E, 0x01FEE01F01FEF10E, 0xE001FE1FF101FE0E, 0xFEE01F01FEF10E01,
    0x01FEFE0101FEFE01, 0xE01F01FEF10E01FE, 0xFEE0E0FEFEF1F1FE, 0x1F01011F0E01010E,
    0xE01F1FE0F10E0EF1, 0xFEFE0101FEFE0101, 0x1F01E0FE0E01F1FE, 0xE01FFE01F10EFE01,
    0xFEFE1F1FFEFE0E0E, 0x1F01FEE00E01FEF1, 0xE0E00101F1F10101, 0xFEFEE0E0FEFEF1F1};
```

10.2.9.2.1 CryptSetOddByteParity()

This function sets the per byte parity of a 64-bit value. The least-significant bit is of each byte is replaced with the odd parity of the other 7 bits in the byte. With odd parity, no byte will ever be 0x00.

```
UINT64
CryptSetOddByteParity(
    UINT64          k
    )
{
#define PMASK 0x0101010101010101ULL
    UINT64          out;
    k |= PMASK;     // set the parity bit
    out = k;
    k ^= k >> 4;
    k ^= k >> 2;
    k ^= k >> 1;
    k &= PMASK;     // odd parity extracted
    out ^= k;       // out is now even parity because parity bit was already set
    out ^= PMASK;   // out is now even parity
    return out;
}
```

10.2.9.2.2 CryptDesIsWeakKey()

Check to see if a DES key is on the list of weak, semi-weak, or possibly weak keys.

```
static BOOL
CryptDesIsWeakKey(
    UINT64            k
    )
{
    int              i;
//
    for(i = 0; i < DES_NUM_WEAK; i++)
    {
        if(k == DesWeakKeys[i])
            return TRUE;
    }
```



```
50      return FALSE;
51  }
```

10.2.9.2.3 CryptDesValidateKey()

Function to check to see if the input key is a valid DES key where the definition of valid is that none of the elements are on the list of weak, semi-weak, or possibly weak keys; and that for two keys, K1!=K2, and for three keys that K1!=K2 and K2!=K3.

```
52  BOOL
53  CryptDesValidateKey(
54      TPM2B_SYM_KEY       *desKey     // IN: key to validate
55      )
56  {
57      UINT64               k[3];
58      int                  i;
59      int                  keys = (desKey->t.size + 7) / 8;
60      BYTE                *pk = desKey->t.buffer;
61      BOOL                 ok;
62  //
63      // Note: 'keys' is the number of keys, not the maximum index for 'k'
64      ok = ((keys == 2) || (keys == 3)) && ((desKey->t.size % 8) == 0);
65      for(i = 0; ok && i < keys; pk += 8, i++)
66      {
67          k[i] = CryptSetOddByteParity(BYTE_ARRAY_TO_UINT64(pk));
68          ok = !CryptDesIsWeakKey(k[i]);
69      }
70      ok = ok && k[0] != k[1];
71      if(keys == 3)
72          ok = ok && k[1] != k[2];
73      return ok;
74  }
```

10.2.9.2.4 CryptGenerateKeyDes()

This function is used to create a DES key of the appropriate size. The key will have odd parity in the bytes.

```
75  TPM_RC
76  CryptGenerateKeyDes(
77      TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
78                                                  //     for the new key.
79      TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
80      RAND_STATE              *rand               // IN: the "entropy" source for
81      )
82  {
83      // Assume that the publicArea key size has been validated and is a supported
84      // number of bits.
85      sensitive->sensitive.sym.t.size =
86              BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
87      do
88      {
89          BYTE                    *pK = sensitive->sensitive.sym.t.buffer;
90          int                      i = (sensitive->sensitive.sym.t.size + 7) / 8;
91  // Use the random number generator to generate the required number of bits
92          DRBG_Generate(rand, pK, sensitive->sensitive.sym.t.size);
93          for(; i > 0; pK += 8, i--)
94          {
95              UINT64      k = BYTE_ARRAY_TO_UINT64(pK);
96              k = CryptSetOddByteParity(k);
97              UINT64_TO_BYTE_ARRAY(k, pK);
98          }
```

```
 99      } while(!CryptDesValidateKey(&sensitive->sensitive.sym));
100      return TPM_RC_SUCCESS;
101  }
102  #endif
```

10.2.10 CryptEccKeyExchange.c

```
  1  #include "Tpm.h"
  2  #if CC_ZGen_2Phase == YES //%
  3  #ifdef TPM_ALG_ECMQV
```

10.2.10.1.1 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)

```
  4  static BOOL
  5  avf1(
  6      bigNum              bnX,            // IN/OUT: the reduced value
  7      bigNum              bnN             // IN: the order of the curve
  8      )
  9  {
 10  // compute f = 2^(ceil(ceil(log2(n)) / 2))
 11      int                         f = (BnSizeInBits(bnN) + 1) / 2;
 12  // x' = 2^f + (x mod 2^f)
 13      BnMaskBits(bnX, f);   // This is mod 2*2^f but it doesn't matter because
 14                              // the next operation will SET the extra bit anyway
 15      BnSetBit(bnX, f);
 16      return TRUE;
 17  }
```

10.2.10.1.2 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.

Error Returns	Meaning
TPM_RC_SUCCESS	results is valid
TPM_RC_NO_RESULTS	the value for *dsA* does not give a valid point on the curve

```
 18  static TPM_RC
 19  C_2_2_MQV(
 20      TPMS_ECC_POINT        *outZ,          // OUT: the computed point
 21      TPM_ECC_CURVE          curveId,       // IN: the curve for the computations
 22      TPM2B_ECC_PARAMETER   *dsA,           // IN: static private TPM key
 23      TPM2B_ECC_PARAMETER   *deA,           // IN: ephemeral private TPM key
 24      TPMS_ECC_POINT        *QsB,           // IN: static public party B key
 25      TPMS_ECC_POINT        *QeB            // IN: ephemeral public party B key
 26      )
 27  {
 28      CURVE_INITIALIZED(E, curveId);
```

```
29      ECC_CURVE_DATA          *C = AccessCurveData(E);
30      POINT(pQeA);
31      POINT_INITIALIZED(pQeB, QeB);
32      POINT_INITIALIZED(pQsB, QsB);
33      ECC_NUM(bnTa);
34      ECC_INITIALIZED(bnDeA, deA);
35      ECC_INITIALIZED(bnDsA, dsA);
36      ECC_NUM(bnN);
37      ECC_NUM(bnXeB);
38      TPM_RC                  retVal;
39  //
40      // Parameter checks
41      if(E == NULL)
42          ERROR_RETURN(TPM_RC_VALUE);
43      pAssert(outZ != NULL && pQeB != NULL && pQsB != NULL && deA != NULL
44              && dsA != NULL);
45  // Process:
46  //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
47  //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
48  //  3. If P = O, output an error indicator.
49  //  4. Z=xP, where xP is the x-coordinate of P.
50      // Compute the public ephemeral key pQeA = [de,A]G
51      if((retVal = BnPointMult(pQeA, CurveGetG(C), bnDeA, NULL, NULL, E))
52         != TPM_RC_SUCCESS)
53          goto Exit;
54  //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
55  //  tA := (ds,A + de,A  avf(Xe,A)) mod n    (3)
56  //  Compute 'tA' = ('deA' +  'dsA'  avf('XeA')) mod n
57      // Ta = avf(XeA);
58      BnCopy(bnTa, pQeA->x);
59      avf1(bnTa, bnN);
60      // do Ta = ds,A * Ta mod n = dsA * avf(XeA) mod n
61      BnModMult(bnTa, bnDsA, bnTa, bnN);
62      // now Ta = deA + Ta mod n =  deA + dsA * avf(XeA) mod n
63      BnAdd(bnTa, bnTa, bnDeA);
64      BnMod(bnTa, bnN);
65  //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
66  // Put this in because almost every case of h is == 1 so skip the call when
67      // not necessary.
68      if(!BnEqualWord(CurveGetCofactor(C), 1))
69          // Cofactor is not 1 so compute Ta := Ta * h mod n
70          BnModMult(bnTa, bnTa, CurveGetCofactor(C), CurveGetOrder(C));
71      // Now that 'tA' is (h * 'tA' mod n)
72      // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
73      // first, compute XeB = avf(XeB)
74      avf1(bnXeB, bnN);
75      // QsB := [XeB]QsB
76      BnPointMult(pQsB, pQsB, bnXeB, NULL, NULL, E);
77      BnEccAdd(pQeB, pQeB, pQsB, E);
78      // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
79      // If the result is not the point at infinity, return QeB
80      BnPointMult(pQeB, pQeB, bnTa, NULL, NULL, E);
81      if(BnEqualZero(pQeB->z))
82          ERROR_RETURN(TPM_RC_NO_RESULT);
83      // Convert BIGNUM E to TPM2B E
84      BnPointTo2B(outZ, pQeB, E);
85  Exit:
86      CURVE_FREE(E);
87      return retVal;
88  }
89  #endif // TPM_ALG_ECMQV
```


10.2.10.1.3 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).

```
 90  static TPM_RC
 91  C_2_2_ECDH(
 92      TPMS_ECC_POINT          *outZs,         // OUT: Zs
 93      TPMS_ECC_POINT          *outZe,         // OUT: Ze
 94      TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
 95      TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
 96      TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
 97      TPMS_ECC_POINT          *QsB,           // IN: static public party B key
 98      TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
 99      )
100  {
101      CURVE_INITIALIZED(E, curveId);
102      ECC_INITIALIZED(bnAs, dsA);
103      ECC_INITIALIZED(bnAe, deA);
104      POINT_INITIALIZED(ecBs, QsB);
105      POINT_INITIALIZED(ecBe, QeB);
106      POINT(ecZ);
107      TPM_RC            retVal;
108  //
109      // Parameter checks
110      if(E == NULL)
111          ERROR_RETURN(TPM_RC_CURVE);
112      pAssert(outZs != NULL && dsA != NULL && deA != NULL && QsB != NULL
113              && QeB != NULL);
114      // Do the point multiply for the Zs value ([dsA]QsB)
115      retVal = BnPointMult(ecZ, ecBs, bnAs, NULL, NULL, E);
116      if(retVal == TPM_RC_SUCCESS)
117      {
118          // Convert the Zs value.
119          BnPointTo2B(outZs, ecZ, E);
120          // Do the point multiply for the Ze value ([deA]QeB)
121          retVal = BnPointMult(ecZ, ecBe, bnAe, NULL, NULL, E);
122          if(retVal == TPM_RC_SUCCESS)
123              BnPointTo2B(outZe, ecZ, E);
124      }
125  Exit:
126      CURVE_FREE(E);
127      return retVal;
128  }
```

10.2.10.1.4 CryptEcc2PhaseKeyExchange()

This function is the dispatch routine for the EC key exchange functions that use two ephemeral and two static keys.

Error Returns	Meaning
TPM_RC_SCHEME	scheme is not defined

```
129  LIB_EXPORT TPM_RC
130  CryptEcc2PhaseKeyExchange(
131      TPMS_ECC_POINT          *outZ1,         // OUT: a computed point
132      TPMS_ECC_POINT          *outZ2,         // OUT: and optional second point
133      TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
134      TPM_ALG_ID               scheme,        // IN: the key exchange scheme
135      TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
136      TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
137      TPMS_ECC_POINT          *QsB,           // IN: static public party B key
```



```
138         TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
139         )
140     {
141         pAssert(outZ1 != NULL
142                 && dsA != NULL && deA != NULL
143                 && QsB != NULL && QeB != NULL);
144         // Initialize the output points so that they are empty until one of the
145         // functions decides otherwise
146         outZ1->x.b.size = 0;
147         outZ1->y.b.size = 0;
148         if(outZ2 != NULL)
149         {
150             outZ2->x.b.size = 0;
151             outZ2->y.b.size = 0;
152         }
153         switch(scheme)
154         {
155             case TPM_ALG_ECDH:
156                 return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
157                 break;
158     #ifdef  TPM_ALG_ECMQV
159             case TPM_ALG_ECMQV:
160                 return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
161                 break;
162     #endif
163     #ifdef  TPM_ALG_SM2
164             case TPM_ALG_SM2:
165                 return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
166                 break;
167     #endif
168             default:
169                 return TPM_RC_SCHEME;
170         }
171     }
172     #ifdef TPM_ALG_SM2
```

10.2.10.1.5 ComputeWForSM2()

Compute the value for w used by SM2

```
173     static UINT32
174     ComputeWForSM2(
175         bigCurve        E
176         )
177     {
178         //  w := ceil(ceil(log2(n)) / 2) - 1
179         return (BnMsb(CurveGetOrder(AccessCurveData(E))) / 2 - 1);
180     }
```

10.2.10.1.6 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.

```
181     static bigNum
182     avfSm2(
183         bigNum              bn,           // IN/OUT: the reduced value
184         UINT32              w              // IN: the value of w
185         )
```

```
186 {
187     // a)   set w := ceil(ceil(log2(n)) / 2) - 1
188     // b)   set x' := 2^w + ( x & (2^w - 1))
189     // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
190     BnMaskBits(bn, w);   // as with avf1, this is too big by a factor of 2 but
191                          // it doesn't matter because we SET the extra bit
192                          // anyway
193     BnSetBit(bn, w);
194     return bn;
195 }
```

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

Error Returns	Meaning
TPM_RC_SUCCESS	results is valid
TPM_RC_NO_RESULTS	the value for *dsA* does not give a valid point on the curve

```
196 LIB_EXPORT TPM_RC
197 SM2KeyExchange(
198     TPMS_ECC_POINT        *outZ,          // OUT: the computed point
199     TPM_ECC_CURVE          curveId,       // IN: the curve for the computations
200     TPM2B_ECC_PARAMETER   *dsAIn,         // IN: static private TPM key
201     TPM2B_ECC_PARAMETER   *deAIn,         // IN: ephemeral private TPM key
202     TPMS_ECC_POINT        *QsBIn,         // IN: static public party B key
203     TPMS_ECC_POINT        *QeBIn          // IN: ephemeral public party B key
204     )
205 {
206     CURVE_INITIALIZED(E, curveId);
207     const ECC_CURVE_DATA      *C = (E != NULL) ? AccessCurveData(E) : NULL;
208     ECC_INITIALIZED(dsA, dsAIn);
209     ECC_INITIALIZED(deA, deAIn);
210     POINT_INITIALIZED(QsB, QsBIn);
211     POINT_INITIALIZED(QeB, QeBIn);
212     BN_WORD_INITIALIZED(One, 1);
213     POINT(QeA);
214     ECC_NUM(XeB);
215     POINT(Z);
216     ECC_NUM(Ta);
217     UINT32                  w;
218     TPM_RC                 retVal = TPM_RC_NO_RESULT;
219 //
220     // Parameter checks
221     if(E == NULL)
222         ERROR_RETURN(TPM_RC_CURVE);
223     pAssert(outZ != NULL && dsA != NULL && deA != NULL &&  QsB != NULL
224             && QeB != NULL);
225     // Compute the value for w
226     w = ComputeWForSM2(E);
227     // Compute the public ephemeral key pQeA = [de,A]G
228     if(!BnEccModMult(QeA, CurveGetG(C), deA, E))
229         goto Exit;
230     //  tA := (ds,A + de,A  avf(Xe,A)) mod n    (3)
231     //  Compute 'tA' = ('dsA' +  'deA'  avf('XeA')) mod n
232     // Ta = avf(XeA);
233     // do Ta = de,A * Ta = deA * avf(XeA)
234     BnMult(Ta, deA, avfSm2(QeA->x, w));
235     // now Ta = dsA + Ta =  dsA + deA * avf(XeA)
236     BnAdd(Ta, dsA, Ta);
237     BnMod(Ta, CurveGetOrder(C));
```

```
238         //  outZ = [h  tA mod n] (Qs,B + [avf(Xe,B)](Qe,B)) (4)
239         // Put this in because almost every case of h is == 1 so skip the call when
240         // not necessary.
241         if(!BnEqualWord(CurveGetCofactor(C), 1))
242             // Cofactor is not 1 so compute Ta := Ta * h mod n
243             BnModMult(Ta, Ta, CurveGetCofactor(C), CurveGetOrder(C));
244         // Now that 'tA' is (h * 'tA' mod n)
245         // 'outZ' = ['tA'](QsB + [avf(QeB.x)](QeB)).
246         BnCopy(XeB, QeB->x);
247         if(!BnEccModMult2(Z, QsB, One, QeB, avfSm2(XeB, w), E))
248             goto Exit;
249         // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
250         if(!BnEccModMult(Z, Z, Ta, E))
251             goto Exit;
252         // Convert BIGNUM E to TPM2B E
253         BnPointTo2B(outZ, Z, E);
254         retVal = TPM_RC_SUCCESS;
255     Exit:
256         CURVE_FREE(E);
257         return retVal;
258     }
259     #endif
260     #endif //% CC_ZGen_2Phase
```

10.2.11 CryptEccMain.c

10.2.11.1 Includes and Defines

```
1     #include "Tpm.h"
2     #ifdef TPM_ALG_ECC
```

This version requires that the new format for ECC data be used

```
3     #ifndef USE_BN_ECC_DATA
4     #error "Need to define USE_BN_ECC_DATA in Implementaion.h"
5     #endif
```

10.2.11.2 Functions

```
6     #ifdef SIMULATION
7     void
8     EccSimulationEnd(
9         void
10        )
11    {
12    #ifdef SIMULATION
13    // put things to be printed at the end of the simulation here
14    #endif
15    }
16    #endif // SIMULATION
```

10.2.11.2.1 CryptEccInit()

This function is called at _TPM_Init()

```
17    BOOL
18    CryptEccInit(
19        void
20        )
21    {
22        return TRUE;
```

```
}
```

10.2.11.2.2 CryptEccStartup()

This function is called at TPM2_Startup().

```
BOOL
CryptEccStartup(
    void
    )
{
    return TRUE;
}
```

10.2.11.2.3 ClearPoint2B(generic

Initialize the size values of a TPMS_ECC_POINT structure.

```
void
ClearPoint2B(
    TPMS_ECC_POINT      *p          // IN: the point
    )
{
    if(p != NULL)
    {
        p->x.t.size = 0;
        p->y.t.size = 0;
    }
}
```

10.2.11.2.4 CryptEccGetParametersByCurveId()

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. This function is in this module so that it can be called by GetCurve() data.

Return Value	Meaning
NULL	curve with the indicated TPM_ECC_CURVE value is not implemented
non-NULL	pointer to the curve data

```
LIB_EXPORT const ECC_CURVE *
CryptEccGetParametersByCurveId(
    TPM_ECC_CURVE       curveId     // IN: the curveID
    )
{
    int          i;
    for(i = 0; i < ECC_CURVE_COUNT; i++)
    {
        if(eccCurves[i].curveId == curveId)
            return &eccCurves[i];
    }
    return NULL;
}
```

10.2.11.2.5 CryptEccGetKeySizeForCurve()

This function returns the key size in bits of the indicated curve

```
55  LIB_EXPORT UINT16
56  CryptEccGetKeySizeForCurve(
57      TPM_ECC_CURVE            curveId    // IN: the curve
58      )
59  {
60      const ECC_CURVE *curve = CryptEccGetParametersByCurveId(curveId);
61      UINT16           keySizeInBits;
62  //
63      keySizeInBits = (curve != NULL) ? curve->keySizeBits : 0;
64      return keySizeInBits;
65  }
```

10.2.11.2.6 GetCurveData()

This function returns the a pointer for the parameter data associated with a curve.

```
66  const ECC_CURVE_DATA *
67  GetCurveData(
68      TPM_ECC_CURVE        curveId     // IN: the curveID
69      )
70  {
71      const ECC_CURVE      *curve = CryptEccGetParametersByCurveId(curveId);
72      return (curve != NULL) ? curve->curveData : NULL;
73  }
```

10.2.11.2.7 CryptEccGetCurveByIndex()

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.

```
74  LIB_EXPORT TPM_ECC_CURVE
75  CryptEccGetCurveByIndex(
76      UINT16               i
77      )
78  {
79      if(i >= ECC_CURVE_COUNT)
80          return TPM_ECC_NONE;
81      return eccCurves[i].curveId;
82  }
```

10.2.11.2.8 CryptEccGetParameter()

This function returns an ECC curve parameter. The parameter is selected by a single character designator from the set of {PNABXYH}.

Return Value	Meaning
TRUE	curve exists and parameter returned
FALSE	curve does not exist or parameter selector

```
83  LIB_EXPORT BOOL
84  CryptEccGetParameter(
85      TPM2B_ECC_PARAMETER     *out,       // OUT: place to put parameter
86      char                     p,         // IN: the parameter selector
87      TPM_ECC_CURVE            curveId    // IN: the curve id
88      )
89  {
90      const ECC_CURVE_DATA    *curve = GetCurveData(curveId);
91      bigConst                 parameter = NULL;
```

```
92      if(curve != NULL)
93      {
94          switch(p)
95          {
96              case 'p':
97                  parameter = CurveGetPrime(curve);
98                  break;
99              case 'n':
100                 parameter = CurveGetOrder(curve);
101                 break;
102             case 'a':
103                 parameter = CurveGet_a(curve);
104                 break;
105             case 'b':
106                 parameter = CurveGet_b(curve);
107                 break;
108             case 'x':
109                 parameter = CurveGetGx(curve);
110                 break;
111             case 'y':
112                 parameter = CurveGetGy(curve);
113                 break;
114             case 'h':
115                 parameter = CurveGetCofactor(curve);
116                 break;
117             default:
118                 FAIL(FATAL_ERROR_INTERNAL);
119                 break;
120         }
121     }
122     // If not debugging and we get here with parameter still NULL, had better
123     // not try to convert so just return FALSE instead.
124     return (parameter != NULL) ? BnTo2B(parameter, &out->b, 0) : 0;
125 }
```

10.2.11.2.9 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

```
126 TPMI_YES_NO
127 CryptCapGetECCCurve(
128     TPM_ECC_CURVE    curveID,       // IN: the starting ECC curve
129     UINT32           maxCount,      // IN: count of returned curves
130     TPML_ECC_CURVE  *curveList      // OUT: ECC curve list
131     )
132 {
133     TPMI_YES_NO       more = NO;
134     UINT16            i;
135     UINT32            count = ECC_CURVE_COUNT;
136     TPM_ECC_CURVE     curve;
137     // Initialize output property list
138     curveList->count = 0;
139     // The maximum count of curves we may return is MAX_ECC_CURVES
140     if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
141     // Scan the eccCurveValues array
142     for(i = 0; i < count; i++)
143     {
144         curve = CryptEccGetCurveByIndex(i);
```

```
145             // If curveID is less than the starting curveID, skip it
146             if(curve < curveID)
147                 continue;
148             if(curveList->count < maxCount)
149             {
150                 // If we have not filled up the return list, add more curves to
151                 // it
152                 curveList->eccCurves[curveList->count] = curve;
153                 curveList->count++;
154             }
155             else
156             {
157                 // If the return list is full but we still have curves
158                 // available, report this and stop iterating
159                 more = YES;
160                 break;
161             }
162         }
163         return more;
164     }
```

10.2.11.2.10 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```
165     const TPMT_ECC_SCHEME *
166     CryptGetCurveSignScheme(
167         TPM_ECC_CURVE    curveId        // IN: The curve selector
168         )
169     {
170         const ECC_CURVE         *curve = CryptEccGetParametersByCurveId(curveId);
171         if(curve != NULL)
172             return &(curve->sign);
173         else
174             return NULL;
175     }
```

10.2.11.2.11 CryptGenerateR()

This function computes the commit random value for a split signing scheme.

If *c* is NULL, it indicates that *r* is being generated for TPM2_Commit(). If *c* is not NULL, the TPM will validate that the gr.*commitArray* bit associated with the input value of *c* is SET. If not, the TPM returns FALSE and no *r* value is generated.

Return Value	Meaning
TRUE	r value computed
FALSE	no r value computed

```
176     BOOL
177     CryptGenerateR(
178         TPM2B_ECC_PARAMETER     *r,             // OUT: the generated random value
179         UINT16                  *c,             // IN/OUT: count value.
180         TPMI_ECC_CURVE           curveID,       // IN: the curve for the value
181         TPM2B_NAME              *name           // IN: optional name of a key to
182                                                 //     associate with 'r'
183         )
184     {
185         // This holds the marshaled g_commitCounter.
186         TPM2B_TYPE(8B, 8);
187         TPM2B_8B                 cntr = {{8,{0}}};
```

```
188         UINT32                  iterations;
189         TPM2B_ECC_PARAMETER     n;
190         UINT64                  currentCount = gr.commitCounter;
191         UINT16                  t1;
192     //
193         if(!CryptEccGetParameter(&n, 'n', curveID))
194             return FALSE;
195          // If this is the commit phase, use the current value of the commit counter
196         if(c != NULL)
197         {
198             // if the array bit is not set, can't use the value.
199             if(!TEST_BIT((*c & COMMIT_INDEX_MASK), gr.commitArray))
200                 return FALSE;
201             // If it is the sign phase, figure out what the counter value was
202             // when the commitment was made.
203             //
204             // When gr.commitArray has less than 64K bits, the extra
205             // bits of 'c' are used as a check to make sure that the
206             // signing operation is not using an out of range count value
207             t1 = (UINT16)currentCount;
208             // If the lower bits of c are greater or equal to the lower bits of t1
209             // then the upper bits of t1 must be one more than the upper bits
210             // of c
211             if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
212                 // Since the counter is behind, reduce the current count
213                 currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
214             t1 = (UINT16)currentCount;
215             if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
216                 return FALSE;
217             // set the counter to the value that was
218             // present when the commitment was made
219             currentCount = (currentCount & 0xffffffffffff0000) | *c;
220         }
221         // Marshal the count value to a TPM2B buffer for the KDF
222         cntr.t.size = sizeof(currentCount);
223         UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
224         // Now can do the KDF to create the random value for the signing operation
225         // During the creation process, we may generate an r that does not meet the
226         // requirements of the random value.
227         // want to generate a new r.
228         r->t.size = n.t.size;
229         // Arbitrary upper limit on the number of times that we can look for
230         // a suitable random value.  The normally number of tries will be 1.
231         for(iterations = 1; iterations < 1000000;)
232         {
233             int     i;
234             CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, COMMIT_STRING,
235                       &name->b, &cntr.b, n.t.size * 8, r->t.buffer, &iterations, FALSE);
236             // "random" value must be less than the prime
237             if(UnsignedCompareB(r->b.size, r->b.buffer, n.t.size, n.t.buffer) >= 0)
238                 continue;
239             // in this implementation it is required that at least bit
240             // in the upper half of the number be set
241             for(i = n.t.size / 2; i >= 0; i--)
242                 if(r->b.buffer[i] != 0)
243                     return TRUE;
244         }
245         return FALSE;
246     }
```

10.2.11.2.12 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.

```
247  UINT16
248  CryptCommit(
249      void
250      )
251  {
252      UINT16      oldCount = (UINT16)gr.commitCounter;
253      gr.commitCounter++;
254      SET_BIT(oldCount & COMMIT_INDEX_MASK, gr.commitArray);
255      return oldCount;
256  }
```

10.2.11.2.13 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.

```
257  void
258  CryptEndCommit(
259      UINT16          c               // IN: the counter value of the commitment
260      )
261  {
262      ClearBit((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
263  }
```

10.2.11.2.14 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve

Return Value	Meaning
TRUE	Get parameters success
FALSE	Unsupported ECC curve ID

```
264  BOOL
265  CryptEccGetParameters(
266      TPM_ECC_CURVE                curveId,       // IN: ECC curve ID
267      TPMS_ALGORITHM_DETAIL_ECC   *parameters     // OUT: ECC parameters
268      )
269  {
270      const ECC_CURVE             *curve = CryptEccGetParametersByCurveId(curveId);
271      const ECC_CURVE_DATA        *data;
272      BOOL                         found = curve != NULL;
273      if(found)
274      {
275          data = curve->curveData;
276          parameters->curveID = curve->curveId;
277          parameters->keySize = curve->keySizeBits;
278          parameters->kdf = curve->kdf;
279          parameters->sign = curve->sign;
280          BnTo2B(data->prime, &parameters->p.b, 0);
281          BnTo2B(data->a, &parameters->a.b, 0);
282          BnTo2B(data->b, &parameters->b.b, 0);
283          BnTo2B(data->base.x, &parameters->gX.b, parameters->p.t.size);
284          BnTo2B(data->base.y, &parameters->gY.b, parameters->p.t.size);
285          BnTo2B(data->order, &parameters->n.b, 0);
```

```
286             BnTo2B(data->h, &parameters->h.b, 0);
287         }
288         return found;
289     }
```

10.2.11.2.15 BnGetCurvePrime()

This function is used to get just the prime modulus associated with a curve

```
290     const bignum_t *
291     BnGetCurvePrime(
292         TPM_ECC_CURVE            curveId
293         )
294     {
295         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
296         return (C != NULL) ? CurveGetPrime(C) : NULL;
297     }
```

10.2.11.2.16 BnGetCurveOrder()

This function is used to get just the curve order

```
298     const bignum_t *
299     BnGetCurveOrder(
300         TPM_ECC_CURVE            curveId
301         )
302     {
303         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
304         return (C != NULL) ? CurveGetOrder(C) : NULL;
305     }
```

10.2.11.2.17 BnIsOnCurve()

This function checks if a point is on the curve.

```
306     BOOL
307     BnIsOnCurve(
308         pointConst                  Q,
309         const ECC_CURVE_DATA       *C
310         )
311     {
312         BN_VAR(right, (MAX_ECC_KEY_BITS * 3));
313         BN_VAR(left, (MAX_ECC_KEY_BITS * 2));
314         bigConst                  prime = CurveGetPrime(C);
315     //
316         // Show that point is on the curve y^2 = x^3 + ax + b;
317         // Or y^2 = x(x^2 + a) + b
318         // y^2
319         BnMult(left, Q->y, Q->y);
320         BnMod(left, prime);
321     // x^2
322         BnMult(right, Q->x, Q->x);
323         // x^2 + a
324         BnAdd(right, right, CurveGet_a(C));
325     //    BnMod(right, CurveGetPrime(C));
326         // x(x^2 + a)
327         BnMult(right, right, Q->x);
328         // x(x^2 + a) + b
329         BnAdd(right, right, CurveGet_b(C));
330         BnMod(right, prime);
331         if(BnUnsignedCmp(left, right) == 0)
```

```
332         return TRUE;
333     else
334         return FALSE;
335 }
```

10.2.11.2.18 BnIsValidPrivateEcc()

Checks that 0 < x < q

```
336 BOOL
337 BnIsValidPrivateEcc(
338     bigConst                x,          // IN: private key to check
339     bigCurve                E           // IN: the curve to check
340     )
341 {
342     BOOL        retVal;
343     retVal = (!BnEqualZero(x)
344               && (BnUnsignedCmp(x, CurveGetOrder(AccessCurveData(E))) < 0));
345     return retVal;
346 }
347 LIB_EXPORT BOOL
348 CryptEccIsValidPrivateKey(
349     TPM2B_ECC_PARAMETER     *d,
350     TPM_ECC_CURVE           curveId
351     )
352 {
353     BN_INITIALIZED(bnD, MAX_ECC_PARAMETER_BYTES * 8, d);
354     return !BnEqualZero(bnD) && (BnUnsignedCmp(bnD, BnGetCurveOrder(curveId)) < 0);
355 }
```

10.2.11.2.19 BnPointMul()

This function does a point multiply of the form R = [d]S + [u]Q where the parameters are *bigNum* values. If S is NULL and d is not NULL, then it computes R = [d]G + [u]Q or just R = [d]G if u and Q are NULL. If *skipChecks* is TRUE, then the function will not verify that the inputs are correct for the domain. This would be the case when the values were created by the CryptoEngine() code. It will return TPM_RC_NO_RESULTS if the resulting point is the point at infinity.

Error Returns	Meaning
TPM_RC_NO_RESULTS	result of multiplication is a point at infinity
TPM_RC_ECC_POINT	*S* or *Q* is not on the curve
TPM_RC_VALUE	*d* or *u* is not 0 < d < n

```
356 TPM_RC
357 BnPointMult(
358     bigPoint            R,          // OUT: computed point
359     pointConst          S,          // IN: optional point to multiply by 'd'
360     bigConst            d,          // IN: scalar for [d]S or [d]G
361     pointConst          Q,          // IN: optional second point
362     bigConst            u,          // IN: optional second scalar
363     bigCurve            E           // IN: curve parameters
364     )
365 {
366     BOOL                OK;
367 //
368     TEST(TPM_ALG_ECDH);
369     // Need one scalar
370     OK = (d != NULL || u != NULL);
371     // If S is present, then d has to be present. If S is not
```

```
372         // present, then d may or may not be present
373         OK = OK && (((S == NULL) == (d == NULL)) || (d != NULL));
374         // either both u and Q have to be provided or neither can be provided (don't
375         // know what to do if only one is provided.
376         OK = OK && ((u == NULL) == (Q == NULL));
377         OK = OK && (E != NULL);
378         if(!OK)
379             return TPM_RC_VALUE;
380         OK = (S == NULL) || BnIsOnCurve(S, E->C);
381         OK = OK && ((Q == NULL) || BnIsOnCurve(Q, E->C));
382         if(!OK)
383             return TPM_RC_ECC_POINT;
384         if((d != NULL) && (S == NULL))
385             S = CurveGetG(AccessCurveData(E));
386         // If only one scalar, don't need Shamir's trick
387         if((d == NULL) || (u == NULL))
388         {
389             if(d == NULL)
390                 OK = BnEccModMult(R, Q, u, E);
391             else
392                 OK = BnEccModMult(R, S, d, E);
393         }
394         else
395         {
396             OK = BnEccModMult2(R, S, d, Q, u, E);
397         }
398         return  (OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT);
399     }
```

10.2.11.2.20 BnEccGetPrivate()

This function gets random values with no more bits than are in *q* (the curve order) until it finds a value *(d*) such that 1 <= *d* < *q*. This is the method of FIPS 186-3 Section B.1.2 'Key Pair Generation by Testing Candidates' with minor optimizations to reduce the need for a local parameter to hold the value of *q* - 2.

The execution time of this function is non-deterministic. However, the probability that the search will take more than one iteration is very small. As a consequence, the weighted-average run time for this function is significantly less than the method of key pair generation with extra random bits.

```
400     BOOL
401     BnEccGetPrivate(
402         bigNum                   dOut,      // OUT: the qualified random value
403         const ECC_CURVE_DATA    *C,         // IN: curve for which the private key
404                                             //     needs to be appropriate
405         RAND_STATE              *rand       // IN: state for DRBG
406         )
407     {
408     //
409         bigConst                 order = CurveGetOrder(C);
410     //
411         do
412         {
413             BnGetRandomBits(dOut, BnSizeInBits(order), rand);
414             BnAddWord(dOut, dOut, 1);
415         } while(BnUnsignedCmp(dOut, order) >= 0);
416         return TRUE;
417     }
418     BOOL
419     BnEccGenerateKeyPair(
420         bigNum              bnD,            // OUT: private scalar
421         bn_point_t          *ecQ,           // OUT: public point
422         bigCurve             E,             // IN: curve for the point
423         RAND_STATE          *rand           // IN: DRBG state to use
424         )
```

```
425 {
426     BOOL                OK = FALSE;
427     int                 limit;
428     for(limit = 100; (limit > 0) && !OK; limit--)
429     {
430         // Get a private scalar
431         BnEccGetPrivate(bnD, E->C, rand);
432         // Do a point multiply
433         OK = BnEccModMult(ecQ, NULL, bnD, E);
434     }
435     if(!OK)
436         BnSetWord(ecQ->z, 0);
437     else
438         BnSetWord(ecQ->z, 1);
439     return OK;
440 }
```

10.2.11.2.21 CryptEccNewKeyPair

This function creates an ephemeral ECC. It is ephemeral in that is expected that the private part of the key will be discarded

```
441 LIB_EXPORT TPM_RC
442 CryptEccNewKeyPair(
443     TPMS_ECC_POINT          *Qout,      // OUT: the public point
444     TPM2B_ECC_PARAMETER     *dOut,      // OUT: the private scalar
445     TPM_ECC_CURVE            curveId    // IN: the curve for the key
446     )
447 {
448     CURVE_INITIALIZED(E, curveId);
449     POINT(ecQ);
450     ECC_NUM(bnD);
451     BOOL                    OK;
452     if(E == NULL)
453         return TPM_RC_CURVE;
454     TEST(TPM_ALG_ECDH);
455     OK = BnEccGenerateKeyPair(bnD, ecQ, E, NULL);
456     if(OK)
457     {
458         BnPointTo2B(Qout, ecQ, E);
459         BnTo2B(bnD, &dOut->b, Qout->x.t.size);
460     }
461     else
462     {
463         Qout->x.t.size = Qout->y.t.size = dOut->t.size = 0;
464     }
465     CURVE_FREE(E);
466     return OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
467 }
```

10.2.11.2.22 CryptEccPointMultiply()

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 TPM_RC_NO_RESULTS 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.

Error Returns	Meaning
TPM_RC_ECC_POINT	the point *Pin* or *Qin* is not on the curve
TPM_RC_NO_RESULT	the product point is at infinity
TPM_RC_CURVE	bad curve
TPM_RC_VALUE	*dIn* or *uIn* out of range

```
468  LIB_EXPORT TPM_RC
469  CryptEccPointMultiply(
470      TPMS_ECC_POINT      *Rout,              // OUT: the product point R
471      TPM_ECC_CURVE        curveId,           // IN: the curve to use
472      TPMS_ECC_POINT      *Pin,               // IN: first point (can be null)
473      TPM2B_ECC_PARAMETER *dIn,               // IN: scalar value for [dIn]Qin
474                                              //     the Pin
475      TPMS_ECC_POINT      *Qin,               // IN: point Q
476      TPM2B_ECC_PARAMETER *uIn                // IN: scalar value for the multiplier
477                                              //     of Q
478      )
479  {
480      CURVE_INITIALIZED(E, curveId);
481      POINT_INITIALIZED(ecP, Pin);
482      ECC_INITIALIZED(bnD, dIn);      // If dIn is null, then bnD is null
483      ECC_INITIALIZED(bnU, uIn);
484      POINT_INITIALIZED(ecQ, Qin);
485      POINT(ecR);
486      TPM_RC             retVal;
487  //
488      retVal = BnPointMult(ecR, ecP, bnD, ecQ, bnU, E);
489      if(retVal == TPM_RC_SUCCESS)
490          BnPointTo2B(Rout, ecR, E);
491      else
492          ClearPoint2B(Rout);
493      CURVE_FREE(E);
494      return retVal;
495  }
```

10.2.11.2.23 CryptEccIsPointOnCurve()

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

```
496  LIB_EXPORT BOOL
497  CryptEccIsPointOnCurve(
498      TPM_ECC_CURVE           curveId,        // IN: the curve selector
499      TPMS_ECC_POINT         *Qin             // IN: the point.
500      )
501  {
502      const ECC_CURVE_DATA    *C = GetCurveData(curveId);
503      POINT_INITIALIZED(ecQ, Qin);
504      BOOL            OK;
505  //
506      pAssert(Qin != NULL);
```

```
507         OK = (C != NULL && (BnIsOnCurve(ecQ, C)));
508         return OK;
509     }
```

10.2.11.2.24 CryptEccGenerateKey()

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.1.2 "Key Pair Generation by Testing Candidates." 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.

It is a fatal error if *Qout*, *dOut*, is not provided (is NULL).

If the curve is not supported If *seed* is not provided, then a random number will be used for the key

Error Returns	Meaning
TPM_RC_CURVE	curve is not supported
TPM_RC_FAIL	???

```
510     LIB_EXPORT TPM_RC
511     CryptEccGenerateKey(
512         TPMT_PUBLIC         *publicArea,        // IN/OUT: The public area template for
513                                                 //      the new key. The public key
514                                                 //      area will be replaced computed
515                                                 //      ECC public key
516         TPMT_SENSITIVE      *sensitive,         // OUT: the sensitive area will be
517                                                 //      updated to contain the private
518                                                 //      ECC key and the symmetric
519                                                 //      encryption key
520         RAND_STATE          *rand               // IN: if not NULL, the deterministic
521                                                 //     RNG state
522         )
523     {
524         CURVE_INITIALIZED(E, publicArea->parameters.eccDetail.curveID);
525         ECC_NUM(bnD);
526         POINT(ecQ);
527         const UINT32             MaxCount = 100;
528         UINT32                   count = 0;
529         TPM_RC                   retVal = TPM_RC_NO_RESULT;
530         TEST(TPM_ALG_ECDSA); // ECDSA is used to verify each key
531         // Validate parameters
532         if(E == NULL)
533             ERROR_RETURN(TPM_RC_CURVE);
534         publicArea->unique.ecc.x.t.size = 0;
535         publicArea->unique.ecc.y.t.size = 0;
536         sensitive->sensitive.ecc.t.size = 0;
537         // Start search for key (should be quick)
538         for(count = 1; (count < MaxCount) && (retVal != TPM_RC_SUCCESS); count++)
539         {
540             if(!BnEccGenerateKeyPair(bnD, ecQ, E, rand))
541                 FAIL(FATAL_ERROR_INTERNAL);
542             retVal = TPM_RC_SUCCESS;
543     #ifdef FIPS_COMPLIANT
544             // See if PWCT is required
545             if(publicArea->objectAttributes.sign)
546             {
547                 ECC_NUM(bnT);
548                 ECC_NUM(bnS);
549                 TPM2B_DIGEST    digest;
550                 TEST(TPM_ALG_ECDSA);
551                 digest.t.size =
552                     (UINT16)BITS_TO_BYTES(BnSizeInBits(CurveGetPrime(
```

```
                    AccessCurveData(E))));
            // Get a random value to sign using the current DRBG state
            DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
            BnSignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
            // and make sure that we can validate the signature
            retVal = BnValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest);
        }
#endif
    }
// if counter maxed out, put the TPM to failure mode
    if(count == MaxCount)
        FAIL(FATAL_ERROR_INTERNAL);
// Convert results
    BnPointTo2B(&publicArea->unique.ecc, ecQ, E);
    BnTo2B(bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
Exit:
    CURVE_FREE(E);
    return retVal;
}
#endif  // TPM_ALG_ECC
```

10.2.12 CryptEccSignature.c

10.2.12.1 Includes and Defines

```
#include "Tpm.h"
#include "CryptEccSignature_fp.h"
#ifdef TPM_ALG_ECC
```

10.2.12.2 Utility Functions

10.2.12.2.1 EcdsaDigest()

Function to adjust the digest so that it is no larger than the order of the curve. This is used for ECDSA sign and verification.

```
static bigNum
EcdsaDigest(
    bigNum               bnD,           // OUT: the adjusted digest
    const TPM2B_DIGEST  *digest,        // IN: digest to adjust
    bigConst             max            // IN: value that indicates the maximum
                                        //     number of bits in the results
    )
{
    int              bitsInMax = BnSizeInBits(max);
    int              shift;
//
    if(digest == NULL)
        BnSetWord(bnD, 0);
    else
    {
        BnFromBytes(bnD, digest->t.buffer,
                    (NUMBYTES)MIN(digest->t.size, BITS_TO_BYTES(bitsInMax)));
        shift = BnSizeInBits(bnD) - bitsInMax;
        if(shift > 0)
            BnShiftRight(bnD, bnD, shift);
    }
    return bnD;
}
```

10.2.12.2.2 BnSchnorrSign()

This contains the Schnorr signature computation. It is used by both ECDSA and Schnorr signing. The result is computed as: [s = k + r * d (mod n)] where

a) *s* is the signature

b) *k* is a random value

c) *r* is the value to sign

d) *d* is the private EC key

e) *n* is the order of the curve

Error Returns	Meaning
TPM_RC_NO_RESULT	the result of the operation was zero or r (mod n) is zero

```
27  static TPM_RC
28  BnSchnorrSign(
29      bigNum                  bnS,            // OUT: s component of the signature
30      bigConst                bnK,            // IN: a random value
31      bigNum                  bnR,            // IN: the signature 'r' value
32      bigConst                bnD,            // IN: the private key
33      bigConst                bnN             // IN: the order of the curve
34      )
35  {
36      // Need a local temp value to store the intermediate computation because product
37      // size can be larger than will fit in bnS.
38      BN_VAR(bnT1, MAX_ECC_PARAMETER_BYTES * 2 * 8);
39  //
40      // Reduce bnR without changing the input value
41      BnDiv(NULL, bnT1, bnR, bnN);
42      if(BnEqualZero(bnT1))
43          return TPM_RC_NO_RESULT;
44      // compute s = (k + r * d)(mod n)
45          // r * d
46      BnMult(bnT1, bnT1, bnD);
47          // k * r * d
48      BnAdd(bnT1, bnT1, bnK);
49          // k + r * d (mod n)
50      BnDiv(NULL, bnS, bnT1, bnN);
51      return (BnEqualZero(bnS)) ? TPM_RC_NO_RESULT : TPM_RC_SUCCESS;
52  }
```

10.2.12.3 Signing Functions

10.2.12.3.1 BnSignEcdsa()

This function implements the ECDSA signing algorithm. The method is described in the comments below. This version works with internal numbers.

```
53  TPM_RC
54  BnSignEcdsa(
55      bigNum                  bnR,            // OUT: r component of the signature
56      bigNum                  bnS,            // OUT: s component of the signature
57      bigCurve                E,              // IN: the curve used in the signature
58                                              //     process
59      bigNum                  bnD,            // IN: private signing key
60      const TPM2B_DIGEST     *digest,         // IN: the digest to sign
61      RAND_STATE             *rand            // IN: used in debug of signing
62      )
```

```
63  {
64      ECC_NUM(bnK);
65      ECC_NUM(bnIk);
66      BN_VAR(bnE, MAX_DIGEST_SIZE * 8);
67      POINT(ecR);
68      bigConst                order = CurveGetOrder(AccessCurveData(E));
69      TPM_RC                  retVal = TPM_RC_SUCCESS;
70      INT32                   tries = 10;
71      BOOL                    OK = FALSE;
72  //
73      pAssert(digest != NULL);
74      // The algorithm as described in "Suite B Implementer's Guide to FIPS
75      // 186-3(ECDSA)"
76      // 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a
77      //    per-message secret number and its inverse modulo n. Since n is prime,
78      //    the output will be invalid only if there is a failure in the RBG.
79      // 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
80      //    multiplication (see [Routines]), where G is the base point included in
81      //    the set of domain parameters.
82      // 3. Compute r = xR mod n. If r = 0, then return to Step 1. 1.
83      // 4. Use the selected hash function to compute H = Hash(M).
84      // 5. Convert the bit string H to an integer e as described in Appendix B.2.
85      // 6. Compute s = (k^-1 *  (e + d *  r)) mod q. If s = 0, return to Step 1.2.
86      // 7. Return (r, s).
87      // In the code below, q is n (that it, the order of the curve is p)
88      do // This implements the loop at step 6. If s is zero, start over.
89      {
90          for(; tries > 0; tries--)
91          {
92              // Step 1 and 2 -- generate an ephemeral key and the modular inverse
93              // of the private key.
94              if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
95                  continue;
96           // x coordinate is mod p.  Make it mod q
97              BnMod(ecR->x, order);
98              // Make sure that it is not zero;
99              if(BnEqualZero(ecR->x))
100                 continue;
101             // write the modular reduced version of r as part of the signature
102             BnCopy(bnR, ecR->x);
103             // Make sure that a modular inverse exists and try again if not
104             OK = (BnModInverse(bnIk, bnK, order));
105             if(OK)
106                 break;
107         }
108         if(!OK)
109             goto Exit;
110         EcdsaDigest(bnE, digest, order);
111         // now have inverse of K (bnIk), e (bnE), r (bnR),  d (bnD) and
112         // CurveGetOrder(E)
113         // Compute s = k^-1 (e + r*d)(mod q)
114         //  first do s = r*d mod q
115         BnModMult(bnS, bnR, bnD, order);
116         // s = e + s = e + r * d
117         BnAdd(bnS, bnE, bnS);
118         // s = k^(-1)s (mod n) = k^(-1)(e + r * d)(mod n)
119         BnModMult(bnS, bnIk, bnS, order);
120         // If S is zero, try again
121     } while(BnEqualZero(bnS));
122 Exit:
123     return retVal;
124 }
125 #if ALG_ECDAA
```

10.2.12.3.2 BnSignEcdaa()

This function performs *s* = *r* + *T* * *d* mod q where

a) 'r is a random, or pseudo-random value created in the commit phase

b) *nonceK* is a TPM-generated, random value 0 < *nonceK* < n

c) *T* is mod q of Hash*(nonceK* || digest), and

d) *d* is a private key.

The signature is the tuple *(nonceK*, s)

Regrettably, the parameters in this function kind of collide with the parameter names used in ECSCHNORR making for a lot of confusion. In particular, the *k* value in this function is value in this function u

Error Returns	Meaning
TPM_RC_SCHEME	unsupported hash algorithm

```
126  static TPM_RC
127  BnSignEcdaa(
128      TPM2B_ECC_PARAMETER     *nonceK,        // OUT: nonce component of the signature
129      bigNum                   bnS,           // OUT: s component of the signature
130      bigCurve                 E,             // IN: the curve used in signing
131      bigNum                   bnD,           // IN: the private key
132      const TPM2B_DIGEST      *digest,        // IN: the value to sign (mod q)
133      TPMT_ECC_SCHEME         *scheme,        // IN: signing scheme (contains the
134                                              //      commit count value).
135      OBJECT                  *eccKey,        // IN: The signing key
136      RAND_STATE              *rand           // IN: a random number state
137      )
138  {
139      TPM_RC                   retVal;
140      TPM2B_ECC_PARAMETER      r;
141      HASH_STATE               state;
142      TPM2B_DIGEST             T;
143      BN_MAX(bnT);
144  //
145      NOT_REFERENCED(rand);
146      if(!CryptGenerateR(&r, &scheme->details.ecdaa.count,
147                         eccKey->publicArea.parameters.eccDetail.curveID,
148                         &eccKey->name))
149          retVal = TPM_RC_VALUE;
150      else
151      {
152          // This allocation is here because k is not defined until CrypGenerateR()
153          // is done.
154          ECC_INITIALIZED(bnR, &r);
155          do
156          {
157              // generate nonceK such that 0 < nonceK < n
158              // use bnT as a temp.
159              BnEccGetPrivate(bnT, AccessCurveData(E), rand);
160              BnTo2B(bnT, &nonceK->b, 0);
161              T.t.size = CryptHashStart(&state, scheme->details.ecdaa.hashAlg);
162              if(T.t.size == 0)
163              {
164                  retVal = TPM_RC_SCHEME;
165              }
166              else
167              {
168                  CryptDigestUpdate2B(&state, &nonceK->b);
169                  CryptDigestUpdate2B(&state, &digest->b);
```

```
                CryptHashEnd2B(&state, &T.b);
                BnFrom2B(bnT, &T.b);
                // Watch out for the name collisions in this call!!
                retVal = BnSchnorrSign(bnS, bnR, bnT, bnD,
                                       AccessCurveData(E)->order);
            }
        } while(retVal == TPM_RC_NO_RESULT);
        // Because the rule is that internal state is not modified if the command
        // fails, only end the commit if the command succeeds.
        // NOTE that if the result of the Schnorr computation was zero
        // it will probably not be worthwhile to run the same command again because
        // the result will still be zero. This means that the Commit command will
        // need to be run again to get a new commit value for the signature.
        if(retVal == TPM_RC_SUCCESS)
            CryptEndCommit(scheme->details.ecdaa.count);
    }
    return retVal;
}
#endif // ALG_ECDAA
#if ALG_ECSCHNORR //%
```

10.2.12.3.3 SchnorrReduce()

Function to reduce a hash result if it's magnitude is to large. The size of *number* is set so that it has no more bytes of significance than the reference value. If the resulting number can have more bits of significance than the reference.

```
static void
SchnorrReduce(
    TPM2B       *number,        // IN/OUT: Value to reduce
    bigConst     reference      // IN: the reference value
    )
{
    UINT16      maxBytes = (UINT16)BITS_TO_BYTES(BnSizeInBits(reference));
    if(number->size > maxBytes)
        number->size = maxBytes;
}
```

10.2.12.3.4 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*(xR* || P)(mod q)

c) *rT* = truncated *r*

d) *s*= *k* + *rT* * *ds* (mod q)

e) return the tuple *rT*, s

Error Returns	Meaning
TPM_RC_NO_RESULT	failure in the Schnorr sign process
TPM_RC_SCHEME	*hashAlg* can't produce zero-length digest

```
static TPM_RC
BnSignEcSchnorr(
    bigNum                  bnR,            // OUT: r component of the signature
    bigNum                  bnS,            // OUT: s component of the signature
```

```
204         bigCurve                E,              // IN: the curve used in signing
205         bigNum                  bnD,            // IN: the signing key
206         const TPM2B_DIGEST      *digest,        // IN: the digest to sign
207         TPM_ALG_ID              hashAlg,        // IN: signing scheme (contains a hash)
208         RAND_STATE              *rand           // IN: non-NULL when testing
209         )
210     {
211         HASH_STATE              hashState;
212         UINT16                  digestSize
213             = CryptHashGetDigestSize(hashAlg);
214         TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_KEY_BYTES));
215         TPM2B_T                 T2b;
216         TPM2B                   *e = &T2b.b;
217         TPM_RC                  retVal = TPM_RC_NO_RESULT;
218         const ECC_CURVE_DATA    *C;
219         bigConst                order;
220         bigConst                prime;
221         ECC_NUM(bnK);
222         POINT(ecR);
223     //
224         // Parameter checks
225         if(E == NULL)
226             ERROR_RETURN(TPM_RC_VALUE);
227         C = AccessCurveData(E);
228         order = CurveGetOrder(C);
229         prime = CurveGetOrder(C);
230         // If the digest does not produce a hash, then null the signature and return
231         // a failure.
232         if(digestSize == 0)
233         {
234             BnSetWord(bnR, 0);
235             BnSetWord(bnS, 0);
236             ERROR_RETURN(TPM_RC_SCHEME);
237         }
238         do
239         {
240             // Generate a random key pair
241             if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
242                 break;
243             // Convert R.x to a string
244             BnTo2B(ecR->x, e, (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(prime)));
245             // f) compute r = Hash(e || P) (mod n)
246             CryptHashStart(&hashState, hashAlg);
247             CryptDigestUpdate2B(&hashState, e);
248             CryptDigestUpdate2B(&hashState, &digest->b);
249             e->size = CryptHashEnd(&hashState, digestSize, e->buffer);
250             // Reduce the hash size if it is larger than the curve order
251             SchnorrReduce(e, order);
252             // Convert hash to number
253             BnFrom2B(bnR, e);
254             // Do the Schnorr computation
255             retVal = BnSchnorrSign(bnS, bnK, bnR, bnD, CurveGetOrder(C));
256         } while(retVal == TPM_RC_NO_RESULT);
257     Exit:
258         return retVal;
259     }
260     #endif // ALG_ECSCHNORR
261     #if ALG_SM2
262     #ifdef  _SM2_SIGN_DEBUG
263     static BOOL
264     BnHexEqual(
265         bigNum          bn,         //IN: big number value
266         const char      *c          //IN: character string number
267         )
268     {
269         ECC_NUM(bnC);
```

```
270         BnFromHex(bnC, c);
271         return (BnUnsignedCmp(bn, bnC) == 0);
272     }
273     #endif // _SM2_SIGN_DEBUG
```

10.2.12.3.5 BnSignEcSm2()

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.

Error Returns	Meaning
TPM_RC_VALUE	bad curve

```
274     static TPM_RC
275     BnSignEcSm2(
276         bigNum                  bnR,        // OUT: r component of the signature
277         bigNum                  bnS,        // OUT: s component of the signature
278         bigCurve                E,          // IN: the curve used in signing
279         bigNum                  bnD,        // IN: the private key
280         const TPM2B_DIGEST      *digest,    // IN: the digest to sign
281         RAND_STATE              *rand       // IN: random number generator (mostly for
282                                             //     debug)
283         )
284     {
285         BN_MAX_INITIALIZZED(bnE, digest);   // Don't know how big digest might be
286         ECC_NUM(bnN);
287         ECC_NUM(bnK);
288         ECC_NUM(bnX1);
289         ECC_NUM(bnT);   // temp
290         POINT(Q1);
291         bigConst                  order = (E != NULL)
292             ? CurveGetOrder(AccessCurveData(E)) : NULL;
293     //
294     #ifdef _SM2_SIGN_DEBUG
295         BnFromHex(bnE,
296                   "B524F552CD82B8B028476E005C377FB19A87E6FC682D48BB5D42E3D9B9EFFE76");
297         BnFromHex(bnD,
298                   "128B2FA8BD433C6C068C8D803DFF79792A519A55171B1B650C23661D15897263");
299     #endif
300         // A3: Use random number generator to generate random number 1 <= k <= n-1;
301         // NOTE: Ax: numbers are from the SM2 standard
302     loop:
303         {
304             // Get a random number 0 < k < n
305             BnGenerateRandomInRange(bnK, order, rand);
306     #ifdef _SM2_SIGN_DEBUG
307             BnFromHex(bnK,
308                       "6CB28D99385C175C94F94E934817663FC176D925DD72B727260DBAAE1FB2F96F");
309     #endif
310             // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
311             // to details specified in 4.2.7 in Part 1 of this document, transform the
312             // data type of x1 into an integer;
313             if(BnEccModMult(Q1, NULL, bnK, E) != TPM_RC_SUCCESS)
314                 goto loop;
315             // A5: Figure out r = (e + x1) mod n,
316             BnAdd(bnR, bnE, bnX1);
317             BnMod(bnR, order);
318     #ifdef _SM2_SIGN_DEBUG
319             pAssert(BnHexEqual(bnR,
320
    "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1"));
321     #endif
```

```
322             // if r=0 or r+k=n, return to A3;
323             if(BnEqualZero(bnR))
324                 goto loop;
325             BnAdd(bnT, bnK, bnR);
326             if(BnUnsignedCmp(bnT, bnN) == 0)
327                 goto loop;
328             // A6: Figure out s = ((1 + dA)^-1  (k - r  dA)) mod n,
329             // if s=0, return to A3;
330             // compute t = (1+dA)^-1
331             BnAddWord(bnT, bnD, 1);
332             BnModInverse(bnT, bnT, order);
333     #ifdef _SM2_SIGN_DEBUG
334             pAssert(BnHexEqual(bnT,
335     
        "79BFCF3052C80DA7B939E0C6914A18CBB2D96D8555256E83122743A7D4F5F956"));
336     #endif
337             // compute s = t * (k - r * dA) mod n
338             BnModMult(bnS, bnR, bnD, order);
339             // k - r * dA mod n = k + n - ((r * dA) mod n)
340             BnSub(bnS, order, bnS);
341             BnAdd(bnS, bnK, bnS);
342             BnModMult(bnS, bnS, bnT, order);
343     #ifdef _SM2_SIGN_DEBUG
344             pAssert(BnHexEqual(bnS,
345     
        "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7"));
346     #endif
347             if(BnEqualZero(bnS))
348                 goto loop;
349         }
350         // A7: According to details specified in 4.2.1 in Part 1 of this document,
351         // transform the data type of r, s into bit strings, signature of message M
352         // is (r, s).
353         // This is handled by the common return code
354     #ifdef _SM2_SIGN_DEBUG
355         pAssert(BnHexEqual(bnR,
356     
        "40F1EC59F793D9F49E09DCEF49130D4194F79FB1EED2CAA55BACDB49C4E755D1"));
357         pAssert(BnHexEqual(bnS,
358     
        "6FC6DAC32C5D5CF10C77DFB20F7C2EB667A457872FB09EC56327A67EC7DEEBE7"));
359     #endif
360         return TPM_RC_SUCCESS;
361     }
362     #endif // ALG_SM2
```

10.2.12.3.6 CryptEccSign()

This function is the dispatch function for the various ECC-based signing schemes. There is a bit of ugliness to the parameter passing. In order to test this, we sometime would like to use a deterministic RNG so that we can get the same signatures during testing. The easiest way to do this for most schemes is to pass in a deterministic RNG and let it return canned values during testing. There is a competing need for a canned parameter to use in ECDAA. To accommodate both needs with minimal fuss, a special type of RAND_STATE is defined to carry the address of the commit value. The setup and handling of this is not very different for the caller than what was in previous versions of the code.

Error Returns	Meaning
TPM_RC_SCHEME	*scheme* is not supported

```
363     LIB_EXPORT TPM_RC
364     CryptEccSign(
365         TPMT_SIGNATURE          *signature,     // OUT: signature
```

```
366         OBJECT                  *signKey,       // IN: ECC key to sign the hash
367         const TPM2B_DIGEST      *digest,        // IN: digest to sign
368         TPMT_ECC_SCHEME         *scheme,        // IN: signing scheme
369         RAND_STATE              *rand
370         )
371     {
372         CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
373         ECC_INITIALIZED(bnD, &signKey->sensitive.sensitive.ecc.b);
374         ECC_NUM(bnR);
375         ECC_NUM(bnS);
376         const ECC_CURVE_DATA   *C = AccessCurveData(E);
377         TPM_RC                  retVal;
378     //
379         NOT_REFERENCED(scheme);
380         if(E == NULL)
381             ERROR_RETURN(TPM_RC_VALUE);
382         signature->signature.ecdaa.signatureR.t.size
383             = sizeof(signature->signature.ecdaa.signatureR.t.buffer);
384         signature->signature.ecdaa.signatureS.t.size
385             = sizeof(signature->signature.ecdaa.signatureS.t.buffer);
386         // Prime order for this implementation needs to be a multiple of a byte
387         // so, no P521 for now.
388         pAssert((BnSizeInBits(CurveGetPrime(C)) & 7) == 0);
389         pAssert((BnSizeInBits(CurveGetOrder(C)) & 7) == 0);
390         TEST(signature->sigAlg);
391         switch(signature->sigAlg)
392         {
393             case ALG_ECDSA_VALUE:
394                 retVal = BnSignEcdsa(bnR, bnS, E, bnD, digest, rand);
395                 break;
396     #if ALG_ECDAA
397             case ALG_ECDAA_VALUE:
398                 retVal = BnSignEcdaa(&signature->signature.ecdaa.signatureR, bnS, E,
399                                      bnD, digest, scheme, signKey, rand);
400                 bnR = NULL;
401                 break;
402     #endif
403     #if ALG_ECSCHNORR
404             case ALG_ECSCHNORR_VALUE:
405                 retVal = BnSignEcSchnorr(bnR, bnS, E, bnD, digest,
406                                          signature->signature.ecschnorr.hash,
407                                          rand);
408                 break;
409     #endif
410     #if ALG_SM2
411             case ALG_SM2_VALUE:
412                 retVal = BnSignEcSm2(bnR, bnS, E, bnD, digest, rand);
413                 break;
414     #endif
415             default:
416                 return TPM_RC_SCHEME;
417         }
418         // If signature generation worked, convert the results.
419         if(retVal == TPM_RC_SUCCESS)
420         {
421             NUMBYTES     orderBytes =
422                 (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(C)));
423             if(bnR != NULL)
424                 BnTo2B(bnR, &signature->signature.ecdaa.signatureR.b, orderBytes);
425             if(bnS != NULL)
426                 BnTo2B(bnS, &signature->signature.ecdaa.signatureS.b, orderBytes);
427         }
428     Exit:
429         CURVE_FREE(E);
430         return retVal;
431     }
```



```
#if ALG_ECDSA     //%
```

10.2.12.3.7 BnValidateSignatureEcdsa()

This function validates an ECDSA signature. *rIn* and *sIn* should have been checked to make sure that they are in the range 0 < v < n

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```
TPM_RC
BnValidateSignatureEcdsa(
    bigNum                  bnR,            // IN: r component of the signature
    bigNum                  bnS,            // IN: s component of the signature
    bigCurve                E,              // IN: the curve used in the signature
                                            //     process
    bn_point_t             *ecQ,            // IN: the public point of the key
    const TPM2B_DIGEST     *digest          // IN: the digest that was signed
    )
{
    // Make sure that the allocation for the digest is big enough for a maximum
    // digest
    BN_VAR(bnE, MAX_DIGEST_SIZE * 8);
    POINT(ecR);
    ECC_NUM(bnU1);
    ECC_NUM(bnU2);
    ECC_NUM(bnW);
    bigConst                order = CurveGetOrder(AccessCurveData(E));
    TPM_RC                  retVal = TPM_RC_SIGNATURE;
    // Get adjusted digest
    EcdsaDigest(bnE, digest, order);
    // 1. If r and s are not both integers in the interval [1, n - 1], output
    //    INVALID.
    //  bnR  and bnS were validated by the caller
    // 2. Use the selected hash function to compute H0 = Hash(M0).
    // This is an input parameter
    // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
    // Done at entry
    // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
    if(!BnModInverse(bnW, bnS, order))
        goto Exit;
    // 5. Compute u1 = (e' *   w) mod n, and compute u2 = (r' *  w) mod n.
    BnModMult(bnU1, bnE, bnW, order);
    BnModMult(bnU2, bnR, bnW, order);
    // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
    //    scalar multiplication and EC addition (see [Routines]). If R is equal to
    //    the point at infinity O, output INVALID.
    if(BnPointMult(ecR, CurveGetG(AccessCurveData(E)), bnU1, ecQ, bnU2, E)
       != TPM_RC_SUCCESS)
        goto Exit;
    // 7. Compute v = Rx mod n.
    BnMod(ecR->x, order);
    // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
    if(BnUnsignedCmp(ecR->x, bnR) != 0)
        goto Exit;
    retVal = TPM_RC_SUCCESS;
Exit:
    return retVal;
}
#endif      //% ALG_ECDSA
#if ALG_SM2
```

10.2.12.3.8 BnValidateSignatureEcSm2()

This function is used to validate an SM2 signature.

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```
484  static TPM_RC
485  BnValidateSignatureEcSm2(
486      bigNum                  bnR,        // IN: r component of the signature
487      bigNum                  bnS,        // IN: s component of the signature
488      bigCurve                E,          // IN: the curve used in the signature
489                                          //     process
490      bigPoint                ecQ,        // IN: the public point of the key
491      const TPM2B_DIGEST      *digest     // IN: the digest that was signed
492      )
493  {
494      POINT(P);
495      ECC_NUM(bnRp);
496      ECC_NUM(bnT);
497      BN_MAX_INITIALIZED(bnE, digest);
498      BOOL                    OK;
499      bigConst                order = CurveGetOrder(AccessCurveData(E));
500  #ifdef _SM2_SIGN_DEBUG
501      // Make sure that the input signature is the test signature
502      pAssert(BnHexEqual(bnR,
503                         "40F1EC59F793D9F49E09DCEF49130D41"
504                         "94F79FB1EED2CAA55BACDB49C4E755D1"));
505      pAssert(BnHexEqual(bnS,
506                         "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
507                         "67A457872FB09EC56327A67EC7DEEBE7"));
508  #endif
509      // b)   compute t  := (r + s) mod n
510      BnAdd(bnT, bnT, bnS);
511      BnMod(bnT, order);
512  #ifdef _SM2_SIGN_DEBUG
513      pAssert(BnHexEqual(bnT,
514                         "2B75F07ED7ECE7CCC1C8986B991F441A"
515                         "D324D6D619FE06DD63ED32E0C997C801"));
516  #endif
517      // c)   verify that t > 0
518      OK = !BnEqualZero(bnT);
519      if(!OK)
520          // set T to a value that should allow rest of the computations to run
521          // without trouble
522          BnCopy(bnT, bnS);
523      // d)   compute (x, y) := [s]G + [t]Q
524      OK = BnEccModMult2(P, NULL, bnS, ecQ, bnT, E);
525  #ifdef  _SM2_SIGN_DEBUG
526      pAssert(OK && BnHexEqual(P->x,
527                               "110FCDA57615705D5E7B9324AC4B856D"
528                               "23E6D9188B2AE47759514657CE25D112"));
529  #endif
530      // e)   compute r' := (e + x) mod n (the x coordinate is in bnT)
531      BnAdd(bnRp, bnE, P->x);
532      BnMod(bnRp, order);
533      // f)   verify that r' = r
534      OK = (BnUnsignedCmp(bnR, bnRp) != 0) & OK;
535      if(!OK)
536          return TPM_RC_SIGNATURE;
537      else
538          return TPM_RC_SUCCESS;
539  }
540  #endif  //% ALG_SM2
```



```
#if ALG_ECSCHNORR
```

10.2.12.3.9 BnValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```
static TPM_RC
BnValidateSignatureEcSchnorr(
    bigNum               bnR,       // IN: r component of the signature
    bigNum               bnS,       // IN: s component of the signature
    TPM_ALG_ID           hashAlg,   // IN: hash algorithm of the signature
    bigCurve             E,         // IN: the curve used in the signature
                                    //     process
    bigPoint             ecQ,       // IN: the public point of the key
    const TPM2B_DIGEST  *digest     // IN: the digest that was signed
    )
{
    BN_MAX(bnRn);
    POINT(ecE);
    BN_MAX(bnEx);
    const ECC_CURVE_DATA    *C = AccessCurveData(E);
    bigConst                 order = CurveGetOrder(C);
    UINT16                   digestSize = CryptHashGetDigestSize(hashAlg);
    HASH_STATE               hashState;
    TPM2B_TYPE(BUFFER, MAX(MAX_ECC_PARAMETER_BYTES, MAX_DIGEST_SIZE));
    TPM2B_BUFFER             Ex2 = {{sizeof(Ex2.t.buffer),{ 0 }}};
    BOOL                     OK;
//
    // E = [s]G - [r]Q
    BnMod(bnR, order);
    // Make -r = n - r
    BnSub(bnRn, order, bnR);
    // E = [s]G + [-r]Q
    OK = BnPointMult(ecE, CurveGetG(C), bnS, ecQ, bnRn, E) == TPM_RC_SUCCESS;
//   // reduce the x portion of E mod q
//    OK = OK && BnMod(ecE->x, order);
    // Convert to byte string
    OK = OK && BnTo2B(ecE->x, &Ex2.b,
                      (NUMBYTES)(BITS_TO_BYTES(BnSizeInBits(order))));
    if(OK)
    {
// Ex = h(pE.x || digest)
        CryptHashStart(&hashState, hashAlg);
        CryptDigestUpdate(&hashState, Ex2.t.size, Ex2.t.buffer);
        CryptDigestUpdate(&hashState, digest->t.size, digest->t.buffer);
        Ex2.t.size = CryptHashEnd(&hashState, digestSize, Ex2.t.buffer);
        SchnorrReduce(&Ex2.b, order);
        BnFrom2B(bnEx, &Ex2.b);
        // see if Ex matches R
        OK = BnUnsignedCmp(bnEx, bnR) == 0;
    }
    return (OK) ? TPM_RC_SUCCESS : TPM_RC_SIGNATURE;
}
#endif  // ALG_ECSCHNORR
```

10.2.12.3.10 CryptEccValidateSignature()

This function validates an EcDsa() or EcSchnorr() signature. The point *Qin* needs to have been validated to be on the curve of *curveId*.

Error Returns	Meaning
TPM_RC_SIGNATURE	not a valid signature

```
590   LIB_EXPORT TPM_RC
591   CryptEccValidateSignature(
592       TPMT_SIGNATURE          *signature,     // IN: signature to be verified
593       OBJECT                  *signKey,       // IN: ECC key signed the hash
594       const TPM2B_DIGEST      *digest         // IN: digest that was signed
595       )
596   {
597       CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
598       ECC_NUM(bnR);
599       ECC_NUM(bnS);
600       POINT_INITIALIZED(ecQ, &signKey->publicArea.unique.ecc);
601       bigConst                order;
602       TPM_RC                  retVal;
603       if(E == NULL)
604           ERROR_RETURN(TPM_RC_VALUE);
605       order = CurveGetOrder(AccessCurveData(E));
606   //    // Make sure that the scheme is valid
607       switch(signature->sigAlg)
608       {
609           case ALG_ECDSA_VALUE:
610   #if ALG_ECSCHNORR
611           case ALG_ECSCHNORR_VALUE:
612   #endif
613   #if ALG_SM2
614           case ALG_SM2_VALUE:
615   #endif
616               break;
617           default:
618               ERROR_RETURN(TPM_RC_SCHEME);
619               break;
620       }
621       // Can convert r and s after determining that the scheme is an ECC scheme. If
622       // this conversion doesn't work, it means that the unmarshaling code for
623       // an ECC signature is broken.
624       BnFrom2B(bnR, &signature->signature.ecdsa.signatureR.b);
625       BnFrom2B(bnS, &signature->signature.ecdsa.signatureS.b);
626       // r and s have to be greater than 0 but less than the curve order
627       if(BnEqualZero(bnR) || BnEqualZero(bnS))
628           ERROR_RETURN(TPM_RC_SIGNATURE);
629       if((BnUnsignedCmp(bnS, order) >= 0)
630          || (BnUnsignedCmp(bnR, order) >= 0))
631           ERROR_RETURN(TPM_RC_SIGNATURE);
632       switch(signature->sigAlg)
633       {
634           case ALG_ECDSA_VALUE:
635               retVal = BnValidateSignatureEcdsa(bnR, bnS, E, ecQ, digest);
636               break;
637   #if ALG_ECSCHNORR
638           case ALG_ECSCHNORR_VALUE:
639               retVal = BnValidateSignatureEcSchnorr(bnR, bnS,
640                                                     signature->signature.any.hashAlg,
641                                                     E, ecQ, digest);
642               break;
643   #endif
644   #if ALG_SM2
645           case ALG_SM2_VALUE:
646               retVal = BnValidateSignatureEcSm2(bnR, bnS, E, ecQ, digest);
647               break;
648   #endif
649           default:
650               FAIL(FATAL_ERROR_INTERNAL);
```

```
651         }
652     Exit:
653         CURVE_FREE(E);
654         return retVal;
655     }
```

10.2.12.3.11 CryptEccCommitCompute()

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* is NULL. If *B* is not NULL, then it is a fatal error if *d* is NULL or if *K* and *L* are both NULL. If *M* is not NULL, then it is a fatal error if *E* is NULL.

Error Returns	Meaning
TPM_RC_SUCCESS	computations completed normally
TPM_RC_NO_RESULT	if *K*, *L* or *E* was computed to be the point at infinity
TPM_RC_CANCELED	a cancel indication was asserted during this function

```
656     LIB_EXPORT TPM_RC
657     CryptEccCommitCompute(
658         TPMS_ECC_POINT          *K,             // OUT: [d]B or [r]Q
659         TPMS_ECC_POINT          *L,             // OUT: [r]B
660         TPMS_ECC_POINT          *E,             // OUT: [r]M
661         TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
662         TPMS_ECC_POINT          *M,             // IN: M (optional)
663         TPMS_ECC_POINT          *B,             // IN: B (optional)
664         TPM2B_ECC_PARAMETER     *d,             // IN: d (optional)
665         TPM2B_ECC_PARAMETER     *r              // IN: the computed r value (required)
666         )
667     {
668         CURVE_INITIALIZED(curve, curveId);
669         ECC_INITIALIZED(bnR, r);
670         TPM_RC               retVal = TPM_RC_SUCCESS;
671     //
672         // Validate that the required parameters are provided.
673         // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
674         // E := [r]Q if both M and B are NULL.
675         pAssert(r != NULL && E != NULL);
676         // Initialize the output points in case they are not computed
677         ClearPoint2B(K);
678         ClearPoint2B(L);
679         ClearPoint2B(E);
680         // Sizes of the r parameter may not be zero
681         pAssert(r->t.size > 0);
682         // If B is provided, compute K=[d]B and L=[r]B
683         if(B != NULL)
684         {
685             ECC_INITIALIZED(bnD, d);
686             POINT_INITIALIZED(pB, B);
687             POINT(pK);
688             POINT(pL);
689     //
690             pAssert(d != NULL && K != NULL && L != NULL);
691             if(!BnIsOnCurve(pB, AccessCurveData(curve)))
692                 ERROR_RETURN(TPM_RC_VALUE);
693              // do the math for K = [d]B
694             if((retVal = BnPointMult(pK, pB, bnD, NULL, NULL, curve)) != TPM_RC_SUCCESS)
695                 goto Exit;
696             // Convert BN K to TPM2B K
```



```
697             BnPointTo2B(K, pK, curve);
698             //  compute L= [r]B after checking for cancel
699             if(_plat__IsCanceled())
700                 ERROR_RETURN(TPM_RC_CANCELED);
701             // compute L = [r]B
702             if(!BnIsValidPrivateEcc(bnR, curve))
703                 ERROR_RETURN(TPM_RC_VALUE);
704             if((retVal = BnPointMult(pL, pB, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
705                 goto Exit;
706             // Convert BN L to TPM2B L
707             BnPointTo2B(L, pL, curve);
708         }
709         if((M != NULL) || (B == NULL))
710         {
711             POINT_INITIALIZED(pM, M);
712             POINT(pE);
713     //
714             // Make sure that a place was provided for the result
715             pAssert(E != NULL);
716             // if this is the third point multiply, check for cancel first
717             if((B != NULL) && _plat__IsCanceled())
718                 ERROR_RETURN(TPM_RC_CANCELED);
719             // If M provided, then pM will not be NULL and will compute E = [r]M.
720             // However, if M was not provided, then pM will be NULL and E = [r]G
721             // will be computed
722             if((retVal = BnPointMult(pE, pM, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
723                 goto Exit;
724             // Convert E to 2B format
725             BnPointTo2B(E, pE, curve);
726         }
727     Exit:
728         CURVE_FREE(curve);
729         return retVal;
730     }
731     #endif  // TPM_ALG_ECC
```

10.2.13 CryptHash.c

10.2.13.1 Description

This file contains implementation of cryptographic functions for hashing.

10.2.13.2 Includes, Defines, and Types

```
1   #define _CRYPT_HASH_C_
2   #include "Tpm.h"
3   #define HASH_TABLE_SIZE     (HASH_COUNT + 1)
4   extern const HASH_INFO   g_hashData[HASH_COUNT + 1];
5   #ifdef  TPM_ALG_SHA1
6   HASH_DEF_TEMPLATE(SHA1);
7   #endif
8   #ifdef      TPM_ALG_SHA256
9   HASH_DEF_TEMPLATE(SHA256);
10  #endif
11  #ifdef      TPM_ALG_SHA384
12  HASH_DEF_TEMPLATE(SHA384);
13  #endif
14  #ifdef      TPM_ALG_SHA512
15  HASH_DEF_TEMPLATE(SHA512);
16  #endif
17  HASH_DEF nullDef = {{0}};
```

10.2.13.3 Obligatory Initialization Functions

```
18  BOOL
19  CryptHashInit(
20      void
21      )
22  {
23      LibHashInit();
24      return TRUE;
25  }
26  BOOL
27  CryptHashStartup(
28      void
29      )
30  {
31      return TRUE;
32  }
```

10.2.13.4 Hash Information Access Functions

10.2.13.4.1 Introduction

These functions provide access to the hash algorithm description information.

10.2.13.4.2 CryptGetHashDef()

This function accesses the hash descriptor associated with a hash a algorithm. The function returns NULL for TPM_ALG_NULL and fails if *hashAlg* is not a hash algorithm.

```
33  const PHASH_DEF
34  CryptGetHashDef(
35      TPM_ALG_ID       hashAlg
36      )
37  {
38      PHASH_DEF       retVal;
39      switch(hashAlg)
40      {
41  #ifdef TPM_ALG_SHA1
42          case TPM_ALG_SHA1:
43              return &SHA1_Def;
44              break;
45  #endif
46  #ifdef TPM_ALG_SHA256
47          case TPM_ALG_SHA256:
48              retVal = &SHA256_Def;
49              break;
50  #endif
51  #ifdef  TPM_ALG_SHA384
52          case TPM_ALG_SHA384:
53              retVal = &SHA384_Def;
54              break;
55  #endif
56  #ifdef  TPM_ALG_SHA512
57          case TPM_ALG_SHA512:
58              retVal = &SHA512_Def;
59              break;
60  #endif
61          default:
62              retVal = &nullDef;
63              break;
64      }
65      return retVal;
```



```
66  }
```

10.2.13.4.3 CryptHashIsImplemented()

This function tests to see if an algorithm ID is a valid hash algorithm. If flag is true, then TPM_ALG_NULL is a valid hash.

Return Value	Meaning
TRUE	*hashAlg* is a valid, implemented hash on this TPM.
FALSE	not valid

```
67  BOOL
68  CryptHashIsImplemented(
69      TPM_ALG_ID       hashAlg,
70      BOOL             flag
71      )
72  {
73      switch(hashAlg)
74      {
75  #ifdef TPM_ALG_SHA1
76          case TPM_ALG_SHA1:
77  #endif
78  #ifdef TPM_ALG_SHA256
79          case TPM_ALG_SHA256:
80  #endif
81  #ifdef  TPM_ALG_SHA384
82          case TPM_ALG_SHA384:
83  #endif
84  #ifdef  TPM_ALG_SHA512
85          case TPM_ALG_SHA512:
86  #endif
87  #ifdef TPM_ALG_SM3_256
88          case TPM_ALG_SHA256:
89  #endif
90              return TRUE;
91              break;
92          case TPM_ALG_NULL:
93              return flag;
94              break;
95          default:
96              break;
97      }
98      return FALSE;
99  }
```

10.2.13.4.4 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.

NOTE: The data structure must have a digest size of 0 for TPM_ALG_NULL.

```
100 static
101 const HASH_INFO *
102 GetHashInfoPointer(
103     TPM_ALG_ID       hashAlg
104     )
105 {
106     UINT32              i;
107 //
```

```
108         // TPM_ALG_NULL is the stop value so search up to it
109         for(i = 0; i < HASH_COUNT; i++)
110         {
111             if(g_hashData[i].alg == hashAlg)
112                 return &g_hashData[i];
113         }
114         // either the input was TPM_ALG_NUL or we didn't find the requested algorithm
115         // in either case return a pointer to the TPM_ALG_NULL "hash" descriptor
116         return &g_hashData[HASH_COUNT];
117     }
```

10.2.13.4.5 CryptHashGetAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and an *index* of 2 will return the last. All other index values will return TPM_ALG_NULL.

Return Value	Meaning
TPM_ALG_xxx()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
118     LIB_EXPORT TPM_ALG_ID
119     CryptHashGetAlgByIndex(
120         UINT32          index           // IN: the index
121         )
122     {
123         if(index >= HASH_COUNT)
124             return TPM_ALG_NULL;
125         return g_hashData[index].alg;
126     }
```

10.2.13.4.6 CryptHashGetDigestSize()

Returns the size of the digest produced by the hash. If *hashAlg* is not a hash algorithm, the TPM will FAIL.

Return Value	Meaning
0	TPM_ALG_NULL
> 0	the digest size

```
127     LIB_EXPORT UINT16
128     CryptHashGetDigestSize(
129         TPM_ALG_ID      hashAlg         // IN: hash algorithm to look up
130         )
131     {
132         return CryptGetHashDef(hashAlg)->digestSize;
133     }
```

10.2.13.4.7 CryptHashGetBlockSize()

Returns the size of the block used by the hash. If *hashAlg* is not a hash algorithm, the TPM will FAIL.

Return Value	Meaning
0	TPM_ALG_NULL
> 0	the digest size

```
134 LIB_EXPORT UINT16
135 CryptHashGetBlockSize(
136     TPM_ALG_ID       hashAlg        // IN: hash algorithm to look up
137     )
138 {
139     return CryptGetHashDef(hashAlg)->blockSize;
140 }
```

10.2.13.4.8 CryptHashGetDer

This function returns a pointer to the DER string for the algorithm and indicates its size.

```
141 LIB_EXPORT UINT16
142 CryptHashGetDer(
143     TPM_ALG_ID       hashAlg,       // IN: the algorithm to look up
144     const BYTE      **p
145     )
146 {
147     const HASH_INFO       *q;
148     q = GetHashInfoPointer(hashAlg);
149     *p = &q->der[0];
150     return q->derSize;
151 }
```

10.2.13.4.9 CryptHashGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```
152 TPM_ALG_ID
153 CryptHashGetContextAlg(
154     PHASH_STATE      state          // IN: the context to check
155     )
156 {
157     return state->hashAlg;
158 }
```

10.2.13.5 State Import and Export

```
159 #if 1
```

10.2.13.5.1 CryptHashCopyState

This function is used to **clone** a HASH_STATE.

```
160 LIB_EXPORT void
161 CryptHashCopyState(
162     HASH_STATE          *out,           // OUT: destination of the state
163     const HASH_STATE    *in             // IN: source of the state
164     )
165 {
166     pAssert(out->type == in->type);
167     out->hashAlg = in->hashAlg;
168     out->def = in->def;
169     if(in->hashAlg != TPM_ALG_NULL)
```



```
170         {
171             // Just verify that the hashAlg makes sense (is implemented)
172             CryptGetHashDef(in->hashAlg);
173             // ... and copy.
174              HASH_STATE_COPY(out, in);
175         }
176         if(in->type == HASH_STATE_HMAC)
177         {
178             const HMAC_STATE    *hIn = (HMAC_STATE *)in;
179             HMAC_STATE          *hOut = (HMAC_STATE *)out;
180             hOut->hmacKey = hIn->hmacKey;
181         }
182         return;
183     }
184     #endif //0
```

10.2.13.5.2 CryptHashExportState()

This function is used to export a hash or HMAC hash state. This function would be called when preparing to context save a sequence object.

```
185     void
186     CryptHashExportState(
187         PCHASH_STATE         internalFmt,   // IN: the hash state formatted for use by
188                                             //     library
189         PEXPORT_HASH_STATE   externalFmt    // OUT: the exported hash state
190         )
191     {
192         BYTE                    *outBuf = (BYTE *)externalFmt;
193     //
194         cAssert(sizeof(HASH_STATE) <= sizeof(EXPORT_HASH_STATE));
195     #define CopyToOffset(value)                                                 \
196             memcpy(&outBuf[offsetof(HASH_STATE,value)], &internalFmt->value,    \
197                    sizeof(internalFmt->value))
198         CopyToOffset(hashAlg);
199         CopyToOffset(type);
200         if(internalFmt->type == HASH_STATE_HMAC)
201         {
202             HMAC_STATE              *from = (HMAC_STATE *)internalFmt;
203             memcpy(&outBuf[offsetof(HMAC_STATE, hmacKey)], &from->hmacKey,
204                    sizeof(from->hmacKey));
205         }
206         if(internalFmt->hashAlg != TPM_ALG_NULL)
207              HASH_STATE_EXPORT(externalFmt, internalFmt);
208     }
```

10.2.13.5.3 CryptHashImportState()

This function is used to import the hash state. This function would be called to import a hash state when the context of a sequence object was being loaded.

```
209     void
210     CryptHashImportState(
211         PHASH_STATE          internalFmt,   // OUT: the hash state formatted for use by
212                                             //     the library
213         PCEXPORT_HASH_STATE  externalFmt    // IN: the exported hash state
214         )
215     {
216         BYTE                    *inBuf = (BYTE *)externalFmt;
217     //
218     #define CopyFromOffset(value)                                               \
219         memcpy(&internalFmt->value, &inBuf[offsetof(HASH_STATE,value)],         \
```

```
220             sizeof(internalFmt->value))
221     // Copy the hashAlg of the byte-aligned input structure to the structure-aligned
222     // internal structure.
223     CopyFromOffset(hashAlg);
224     CopyFromOffset(type);
225     if(internalFmt->hashAlg != TPM_ALG_NULL)
226     {
227         internalFmt->def = CryptGetHashDef(internalFmt->hashAlg);
228         HASH_STATE_IMPORT(internalFmt, inBuf);
229         if(internalFmt->type == HASH_STATE_HMAC)
230         {
231             HMAC_STATE              *to = (HMAC_STATE *)internalFmt;
232             memcpy(&to->hmacKey, &inBuf[offsetof(HMAC_STATE, hmacKey)],
233                    sizeof(to->hmacKey));
234         }
235     }
236 }
```

10.2.13.6 State Modification Functions

10.2.13.6.1 HashEnd()

Local function to complete a hash that uses the *hashDef* instead of an algorithm ID. This function is used to complete the hash and only return a partial digest. The return value is the size of the data copied.

```
237 static UINT16
238 HashEnd(
239     PHASH_STATE      hashState,     // IN: the hash state
240     UINT32           dOutSize,      // IN: the size of receive buffer
241     PBYTE            dOut           // OUT: the receive buffer
242     )
243 {
244     BYTE                temp[MAX_DIGEST_SIZE];
245     if(hashState->hashAlg == TPM_ALG_NULL)
246         dOutSize = 0;
247     if(dOutSize > 0)
248     {
249         hashState->def = CryptGetHashDef(hashState->hashAlg);
250         // Set the final size
251         dOutSize = MIN(dOutSize, hashState->def->digestSize);
252         // Complete into the temp buffer and then copy
253         HASH_END(hashState, temp);
254         // Don't want any other functions calling the HASH_END method
255         // directly.
256 #undef HASH_END
257         memcpy(dOut, &temp, dOutSize);
258     }
259     hashState->type = HASH_STATE_EMPTY;
260 //    hashState->hashAlg = TPM_ALG_ERROR;
261     return (UINT16)dOutSize;
262 }
```

10.2.13.6.2 CryptHashStart()

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.

This function does not use the sequence parameter. If it is necessary to import or export context, this will start the sequence in a local state and export the state to the input buffer. Will need to add a flag to the state structure to indicate that it needs to be imported before it can be used. (BLEH).

Return Value	Meaning
0	hash is TPM_ALG_NULL
>0	digest size

```
263  LIB_EXPORT UINT16
264  CryptHashStart(
265      PHASH_STATE      hashState,      // OUT: the running hash state
266      TPM_ALG_ID       hashAlg         // IN: hash algorithm
267      )
268  {
269      UINT16               retVal;
270      TEST(hashAlg);
271      hashState->hashAlg = hashAlg;
272      if(hashAlg == TPM_ALG_NULL)
273      {
274          retVal = 0;
275      }
276      else
277      {
278          hashState->def = CryptGetHashDef(hashAlg);
279          HASH_START(hashState);
280          retVal = hashState->def->digestSize;
281      }
282  #undef HASH_START
283      hashState->type = HASH_STATE_HASH;
284      return retVal;
285  }
```

10.2.13.6.3 CryptDigestUpdate()

Add data to a hash or HMAC stack.

```
286  LIB_EXPORT void
287  CryptDigestUpdate(
288      PHASH_STATE      hashState,      // IN: the hash context information
289      UINT32           dataSize,       // IN: the size of data to be added
290      const BYTE      *data            // IN: data to be hashed
291      )
292  {
293      if(hashState->hashAlg != TPM_ALG_NULL)
294      {
295          pAssert((hashState->type == HASH_STATE_HASH)
296                  || (hashState->type == HASH_STATE_HMAC));
297          hashState->def = CryptGetHashDef(hashState->hashAlg);
298          HASH_DATA(hashState, dataSize, (BYTE *)data);
299      }
300      return;
301  }
```

10.2.13.6.4 CryptHashEnd()

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 or *dOutSize*, whichever is smaller

```
302  LIB_EXPORT UINT16
303  CryptHashEnd(
304      PHASH_STATE      hashState,      // IN: the state of hash stack
305      UINT32           dOutSize,       // IN: size of digest buffer
306      BYTE            *dOut            // OUT: hash digest
307      )
308  {
309      pAssert(hashState->type == HASH_STATE_HASH);
310      return HashEnd(hashState, dOutSize, dOut);
311  }
```

10.2.13.6.5 CryptHashBlock()

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 placed in *dOut*

```
312  LIB_EXPORT UINT16
313  CryptHashBlock(
314      TPM_ALG_ID       hashAlg,        // IN: The hash algorithm
315      UINT32           dataSize,       // IN: size of buffer to hash
316      const BYTE      *data,           // IN: the buffer to hash
317      UINT32           dOutSize,       // IN: size of the digest buffer
318      BYTE            *dOut            // OUT: digest buffer
319      )
320  {
321      HASH_STATE          state;
322      CryptHashStart(&state, hashAlg);
323      CryptDigestUpdate(&state, dataSize, data);
324      return HashEnd(&state, dOutSize, dOut);
325  }
```

10.2.13.6.6 CryptDigestUpdate2B()

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.

```
326  LIB_EXPORT void
327  CryptDigestUpdate2B(
328      PHASH_STATE      state,          // IN: the digest state
329      const TPM2B     *bIn             // IN: 2B containing the data
330      )
331  {
332      // Only compute the digest if a pointer to the 2B is provided.
333      // In CryptDigestUpdate(), if size is zero or buffer is NULL, then no change
334      // to the digest occurs. This function should not provide a buffer if bIn is
335      // not provided.
336      pAssert(bIn != NULL);
337      CryptDigestUpdate(state, bIn->size, bIn->buffer);
```



```
338         return;
339     }
```

10.2.13.6.7 CryptHashEnd2B()

This function is the same as CypteCompleteHash() but the digest is placed in a TPM2B. This is the most common use and this is provided for specification clarity. 'digest.size' should be set to indicate the number of bytes to place in the buffer

Return Value	Meaning
>=0	the number of bytes placed in 'digest.buffer'

```
340     LIB_EXPORT UINT16
341     CryptHashEnd2B(
342         PHASH_STATE      state,          // IN: the hash state
343         P2B              digest          // IN: the size of the buffer Out: requested
344                                          //     number of bytes
345         )
346     {
347         return CryptHashEnd(state, digest->size, digest->buffer);
348     }
```

10.2.13.6.8 CryptDigestUpdateInt()

This function is used to include an integer value to a hash stack. The function marshals the integer into its canonical form before calling CryptDigestUpdate().

```
349     LIB_EXPORT void
350     CryptDigestUpdateInt(
351         void            *state,          // IN: the state of hash stack
352         UINT32           intSize,        // IN: the size of 'intValue' in bytes
353         UINT64           intValue        // IN: integer value to be hashed
354         )
355     {
356     #if LITTLE_ENDIAN_TPM == YES
357         intValue = REVERSE_ENDIAN_64(intValue);
358     #endif
359         CryptDigestUpdate(state, intSize, &((BYTE *)&intValue)[8 - intSize]);
360     }
```

10.2.13.7 HMAC Functions

10.2.13.7.1 CryptHmacStart

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)

```
361     LIB_EXPORT UINT16
362     CryptHmacStart(
363         PHMAC_STATE      state,          // IN/OUT: the state buffer
364         TPM_ALG_ID       hashAlg,        // IN: the algorithm to use
365         UINT16           keySize,        // IN: the size of the HMAC key
```



```
366         const BYTE      *key            // IN: the HMAC key
367         )
368     {
369         PHASH_DEF               hashDef;
370         BYTE *                  pb;
371         UINT32                  i;
372     //
373         hashDef = CryptGetHashDef(hashAlg);
374         if(hashDef->digestSize != 0)
375         {
376         // If the HMAC key is larger than the hash block size, it has to be reduced
377         // to fit. The reduction is a digest of the hashKey.
378             if(keySize > hashDef->blockSize)
379             {
380                 // if the key is too big, reduce it to a digest of itself
381                 state->hmacKey.t.size = CryptHashBlock(hashAlg, keySize, key,
382                                                        hashDef->digestSize,
383                                                        state->hmacKey.t.buffer);
384             }
385             else
386             {
387                 memcpy(state->hmacKey.t.buffer, key, keySize);
388                 state->hmacKey.t.size = keySize;
389             }
390             // XOR the key with iPad (0x36)
391             pb = state->hmacKey.t.buffer;
392             for(i = state->hmacKey.t.size; i > 0; i--)
393                 *pb++ ^= 0x36;
394             // if the keySize is smaller than a block, fill the rest with 0x36
395             for(i = hashDef->blockSize - state->hmacKey.t.size; i > 0; i--)
396                 *pb++ = 0x36;
397             // Increase the oPadSize to a full block
398             state->hmacKey.t.size = hashDef->blockSize;
399             // Start a new hash with the HMAC key
400             // This will go in the caller's state structure and may be a sequence or not
401             CryptHashStart((PHASH_STATE)state, hashAlg);
402             CryptDigestUpdate((PHASH_STATE)state, state->hmacKey.t.size,
403                               state->hmacKey.t.buffer);
404             // XOR the key block with 0x5c ^ 0x36
405             for(pb = state->hmacKey.t.buffer, i = hashDef->blockSize; i > 0; i--)
406                 *pb++ ^= (0x5c ^ 0x36);
407         }
408         // Set the hash algorithm
409         state->hashState.hashAlg = hashAlg;
410         // Set the hash state type
411         state->hashState.type = HASH_STATE_HMAC;
412         return hashDef->digestSize;
413     }
```

10.2.13.7.2 CryptHmacEnd()

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)

```
414     LIB_EXPORT UINT16
415     CryptHmacEnd(
416         PHMAC_STATE      state,         // IN: the hash state buffer
417         UINT32           dOutSize,      // IN: size of digest buffer
418         BYTE            *dOut           // OUT: hash digest
```

```
419         )
420     {
421         BYTE                temp[MAX_DIGEST_SIZE];
422         PHASH_STATE         hState = (PHASH_STATE)&state->hashState;
423         pAssert(hState->type == HASH_STATE_HMAC);
424         hState->def = CryptGetHashDef(hState->hashAlg);
425         // Change the state type for completion processing
426         hState->type = HASH_STATE_HASH;
427         if(hState->hashAlg == TPM_ALG_NULL)
428             dOutSize = 0;
429         else
430         {
431         // Complete the current hash
432             HashEnd(hState, hState->def->digestSize, temp);
433             // Do another hash starting with the oPad
434             CryptHashStart(hState, hState->hashAlg);
435             CryptDigestUpdate(hState, state->hmacKey.t.size, state->hmacKey.t.buffer);
436             CryptDigestUpdate(hState, hState->def->digestSize, temp);
437         }
438         return HashEnd(hState, dOutSize, dOut);
439     }
```

10.2.13.7.3 CryptHmacStart2B()

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

```
440     LIB_EXPORT UINT16
441     CryptHmacStart2B(
442         PHMAC_STATE      hmacState,     // OUT: the state of HMAC stack. It will be used
443                                         //      in HMAC update and completion
444         TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
445         P2B              key            // IN: HMAC key
446         )
447     {
448         return CryptHmacStart(hmacState, hashAlg, key->size, key->buffer);
449     }
```

10.2.13.7.4 CryptHmacEnd2B()

This function is the same as CryptHmacEnd() 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*

```
450     LIB_EXPORT UINT16
451     CryptHmacEnd2B(
452         PHMAC_STATE      hmacState,     // IN: the state of HMAC stack
453         P2B              digest         // OUT: HMAC
454         )
455     {
```

```
456         return CryptHmacEnd(hmacState, digest->size, digest->buffer);
457     }
```

10.2.13.8 Mask and Key Generation Functions

10.2.13.8.1 _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

Error Returns	Meaning
0	hash algorithm was TPM_ALG_NULL
> 0	should be the same as *mSize*

```
458     LIB_EXPORT UINT16
459     CryptMGF1(
460         UINT32           mSize,         // IN: length of the mask to be produced
461         BYTE            *mask,          // OUT: buffer to receive the mask
462         TPM_ALG_ID       hashAlg,       // IN: hash to use
463         UINT32           seedSize,      // IN: size of the seed
464         BYTE            *seed           // IN: seed size
465         )
466     {
467         HASH_STATE           hashState;
468         PHASH_DEF            hDef = CryptGetHashDef(hashAlg);
469         UINT32               remaining;
470         UINT32               counter = 0;
471         BYTE                 swappedCounter[4];
472         // If there is no digest to compute return
473         if((hashAlg == TPM_ALG_NULL) || (mSize == 0))
474             return 0;
475         for(remaining = mSize; ; remaining -= hDef->digestSize)
476         {
477             // Because the system may be either Endian...
478             UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
479             // Start the hash and include the seed and counter
480             CryptHashStart(&hashState, hashAlg);
481             CryptDigestUpdate(&hashState, seedSize, seed);
482             CryptDigestUpdate(&hashState, 4, swappedCounter);
483             // Handling the completion depends on how much space remains in the mask
484             // buffer. If it can hold the entire digest, put it there. If not
485             // put the digest in a temp buffer and only copy the amount that
486             // will fit into the mask buffer.
487             HashEnd(&hashState, remaining, mask);
488             if(remaining <= hDef->digestSize)
489                 break;
490             mask = &mask[hDef->digestSize];
491             counter++;
492         }
493         return (UINT16)mSize;
494     }
```

10.2.13.8.2 CryptKDFa()

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.

Error Returns	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
495  LIB_EXPORT UINT16
496  CryptKDFa(
497      TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
498      const TPM2B     *key,           // IN: HMAC key
499      const TPM2B     *label,         // IN: a label for the KDF
500      const TPM2B     *contextU,      // IN: context U
501      const TPM2B     *contextV,      // IN: context V
502      UINT32           sizeInBits,    // IN: size of generated key in bits
503      BYTE            *keyStream,     // OUT: key buffer
504      UINT32          *counterInOut,  // IN/OUT: caller may provide the iteration
505                                      //     counter for incremental operations to
506                                      //     avoid large intermediate buffers.
507      BOOL             once           // IN: TRUE - only 1 iteration is performed
508                                      //     FALSE if iteration count determined by
509                                      //     "sizeInBits"
510      )
511  {
512      UINT32                  counter = 0;    // counter value
513      INT16                   bytes;          // number of bytes to produce
514      BYTE                   *stream = keyStream;
515      HMAC_STATE              hState;
516      UINT16                  digestSize = CryptHashGetDigestSize(hashAlg);
517      pAssert(key != NULL && keyStream != NULL);
518      pAssert(once == FALSE || (sizeInBits & 7) == 0);
519      if(digestSize == 0)
520          return 0;
521      if(counterInOut != NULL)
522          counter = *counterInOut;
523      // If the size of the request is larger than the numbers will handle,
524      // it is a fatal error.
525      pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);
526      bytes = once ? digestSize : (INT16)((sizeInBits + 7) / 8);
527      // Generate required bytes
528      for(; bytes > 0; bytes -= digestSize)
529      {
530          counter++;
531          if(bytes < digestSize)
532              digestSize = bytes;
533          // Start HMAC
534          if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
535              return 0;
536          // Adding counter
537          CryptDigestUpdateInt(&hState.hashState, 4, counter);
538          // Adding label
539          if(label != NULL)
540              HASH_DATA(&hState.hashState, label->size, (BYTE *)label->buffer);
541          // Add a null. SP108 is not very clear about when the 0 is needed but to
542          // make this like the previous version that did not add an 0x00 after
```

```
543             // a null-terminated string, this version will only add a null byte
544             // if the label parameter did not end in a null byte, or if no label
545             // is present.
546             if((label == NULL)
547                || (label->size == 0)
548                || (label->buffer[label->size - 1] != 0))
549                 CryptDigestUpdateInt(&hState.hashState, 1, 0);
550             // Adding contextU
551             if(contextU != NULL)
552                 HASH_DATA(&hState.hashState, contextU->size, contextU->buffer);
553             // Adding contextV
554             if(contextV != NULL)
555                 HASH_DATA(&hState.hashState, contextV->size, contextV->buffer);
556             // Adding size in bits
557             CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);
558             CryptHmacEnd(&hState, digestSize, stream);
559             stream = &stream[digestSize];
560         }
561         // Mask off bits if the required bits is not a multiple of byte size
562         if((sizeInBits % 8) != 0)
563             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
564         if(counterInOut != NULL)
565             *counterInOut = counter;
566         return (UINT16)((sizeInBits + 7) / 8);
567     }
```

10.2.13.8.3 CryptKDFe()

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.

Error Returns	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
568     LIB_EXPORT UINT16
569     CryptKDFe(
570         TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
571         TPM2B           *Z,             // IN: Z
572         const TPM2B     *label,         // IN: a label value for the KDF
573         TPM2B           *partyUInfo,    // IN: PartyUInfo
574         TPM2B           *partyVInfo,    // IN: PartyVInfo
575         UINT32           sizeInBits,    // IN: size of generated key in bits
576         BYTE            *keyStream      // OUT: key buffer
577         )
578     {
579         HASH_STATE       hashState;
580         PHASH_DEF        hashDef = CryptGetHashDef(hashAlg);
581         UINT32           counter = 0;       // counter value
582         UINT16           hLen;
583         BYTE            *stream = keyStream;
584         INT16            bytes;             // number of bytes to generate
585         pAssert(keyStream != NULL && Z != NULL && ((sizeInBits + 7) / 8) < INT16_MAX);
586     //
587         hLen = hashDef->digestSize;
588         bytes = (INT16)((sizeInBits + 7) / 8);
589         if(hashAlg == TPM_ALG_NULL || bytes == 0)
590             return 0;
```

```
591         // Generate required bytes
592         //The inner loop of that KDF uses:
593         //  Hash[i] := H(counter | Z | OtherInfo) (5)
594         // Where:
595         //  Hash[i]   the hash generated on the i-th iteration of the loop.
596         //  H()     an approved hash function
597         //  counter a 32-bit counter that is initialized to 1 and incremented
598         //          on each iteration
599         //  Z       the X coordinate of the product of a public ECC key and a
600         //          different private ECC key.
601         //  OtherInfo   a collection of qualifying data for the KDF defined below.
602         //  In this specification, OtherInfo will be constructed by:
603         //      OtherInfo := Use | PartyUInfo  | PartyVInfo
604         for(; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
605         {
606             if(bytes < hLen)
607                 hLen = bytes;
608             counter++;
609             // Do the hash
610             CryptHashStart(&hashState, hashAlg);
611             // Add counter
612             CryptDigestUpdateInt(&hashState, 4, counter);
613             // Add Z
614             if(Z != NULL)
615                 CryptDigestUpdate2B(&hashState, Z);
616             // Add label
617             if(label != NULL)
618                 CryptDigestUpdate2B(&hashState, label);
619             // Add a null. SP108 is not very clear about when the 0 is needed but to
620             // make this like the previous version that did not add an 0x00 after
621             // a null-terminated string, this version will only add a null byte
622             // if the label parameter did not end in a null byte, or if no label
623             // is present.
624             if((label == NULL)
625                || (label->size == 0)
626                || (label->buffer[label->size - 1] != 0))
627                 CryptDigestUpdateInt(&hashState, 1, 0);
628             // Add PartyUInfo
629             if(partyUInfo != NULL)
630                 CryptDigestUpdate2B(&hashState, partyUInfo);
631             // Add PartyVInfo
632             if(partyVInfo != NULL)
633                 CryptDigestUpdate2B(&hashState, partyVInfo);
634             // Compute Hash. hLen was changed to be the smaller of bytes or hLen
635             // at the start of each iteration.
636             CryptHashEnd(&hashState, hLen, stream);
637         }
638         // Mask off bits if the required bits is not a multiple of byte size
639         if((sizeInBits % 8) != 0)
640             keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
641         return (UINT16)((sizeInBits + 7) / 8);
642     }
```

10.2.14 CryptHashData.c

```
1   #include "Tpm.h"
2   const HASH_INFO   g_hashData[HASH_COUNT + 1] = {
3   #ifdef  TPM_ALG_SHA1
4       {TPM_ALG_SHA1,    SHA1_DIGEST_SIZE,   SHA1_BLOCK_SIZE,
5       SHA1_DER_SIZE,   {SHA1_DER}},
6   #endif
7   #ifdef  TPM_ALG_SHA256
8       {TPM_ALG_SHA256,    SHA256_DIGEST_SIZE,   SHA256_BLOCK_SIZE,
9       SHA256_DER_SIZE,   {SHA256_DER}},
10  #endif
```

```
#ifdef  TPM_ALG_SHA384
    {TPM_ALG_SHA384,    SHA384_DIGEST_SIZE,   SHA384_BLOCK_SIZE,
    SHA384_DER_SIZE,   {SHA384_DER}},
#endif
#ifdef  TPM_ALG_SHA512
    {TPM_ALG_SHA512,    SHA512_DIGEST_SIZE,   SHA512_BLOCK_SIZE,
    SHA512_DER_SIZE,   {SHA512_DER}},
#endif
#ifdef  TPM_ALG_SM3_256
    {TPM_ALG_SM3_256,    SM3_256_DIGEST_SIZE,   SM3_256_BLOCK_SIZE,
    SM3_256_DER_SIZE,   {SM3_256_DER}},
#endif
        {TPM_ALG_NULL,0,0,0,{0}}
    };
```

10.2.15 CryptPrime.c

```
#include "Tpm.h"
#include "CryptPrime_fp.h"
//#define CPRI_PRIME
//#include "PrimeTable.h"
#include "CryptPrimeSieve_fp.h"
extern const uint32_t      s_LastPrimeInTable;
extern const uint32_t      s_PrimeTableSize;
extern const uint32_t      s_PrimesInTable;
extern const unsigned char s_PrimeTable[];
extern bigConst            s_CompositeOfSmallPrimes;
```

10.2.15.1 Root2()

This finds ceil(sqrt(n)) to use as a stopping point for searching the prime table.

```
static uint32_t
Root2(
    uint32_t            n
    )
{
    int32_t             last = (int32_t)(n >> 2);
    int32_t             next = (int32_t)(n >> 1);
    int32_t             diff;
    int32_t             stop = 10;
//
    // get a starting point
    for(; next != 0; last >>= 1, next >>= 2);
    last++;
    do
    {
        next = (last + (n / last)) >> 1;
        diff = next - last;
        last = next;
        if(stop-- == 0)
            FAIL(FATAL_ERROR_INTERNAL);
    } while(diff < -1 || diff > 1);
    if((n / next) > (unsigned)next)
        next++;
    pAssert(next != 0);
    pAssert(((n / next) <= (unsigned)next) && (n / (next + 1) < (unsigned)next));
    return next;
}
```


10.2.15.2 IsPrimeInt()

This will do a test of an word up to 32-bits in size.

```
38  BOOL
39  IsPrimeInt(
40      uint32_t            n
41      )
42  {
43      uint32_t            i;
44      uint32_t            stop;
45      if(n < 3 || ((n & 1) == 0))
46          return (n == 2);
47      if(n <= s_LastPrimeInTable)
48      {
49          n >>= 1;
50          return ((s_PrimeTable[n >> 3] >> (n & 7)) & 1);
51      }
52      // Need to search
53      stop = Root2(n) >> 1;
54      // starting at 1 is equivalent to staring at  (1 << 1) + 1 = 3
55      for(i = 1; i < stop; i++)
56      {
57          if((s_PrimeTable[i >> 3] >> (i & 7)) & 1)
58              // see if this prime evenly divides the number
59              if((n % ((i << 1) + 1)) == 0)
60                  return FALSE;
61      }
62      return TRUE;
63  }
```

10.2.15.3 BnIsPrime()

This function is used when the key sieve is not implemented. This function Will try to eliminate some of the obvious things before going on to perform MillerRabin() as a final verification of primeness.

```
64  BOOL
65  BnIsProbablyPrime(
66      bigNum          prime,          // IN:
67      RAND_STATE      *rand           // IN: the random state just
68                                      //     in case Miller-Rabin is required
69      )
70  {
71  #if RADIX_BITS > 32
72      if(BnUnsignedCmpWord(prime, UINT32_MAX) <= 0)
73  #else
74      if(BnGetSize(prime) == 1)
75  #endif
76          return IsPrimeInt(prime->d[0]);
77      if(BnIsEven(prime))
78          return FALSE;
79      if(BnUnsignedCmpWord(prime, s_LastPrimeInTable) <= 0)
80      {
81          crypt_uword_t       temp = prime->d[0] >> 1;
82          return ((s_PrimeTable[temp >> 3] >> (temp & 7)) & 1);
83      }
84      {
85          BN_VAR(n, LARGEST_NUMBER_BITS);
86          BnGcd(n, prime, s_CompositeOfSmallPrimes);
87          if(!BnEqualWord(n, 1))
88              return FALSE;
89      }
90      return MillerRabin(prime, rand);
```

```
}
```

10.2.15.4 MillerRabinRounds()

Function returns the number of Miller-Rabin rounds necessary to give an error probability equal to the security strength of the prime. These values are from FIPS 186-3.

```
UINT32
MillerRabinRounds(
    UINT32          bits            // IN: Number of bits in the RSA prime
    )
{
    if(bits < 511) return 8;    // don't really expect this
    if(bits < 1536) return 5;   // for 512 and 1K primes
    return 4;                   // for 3K public modulus and greater
}
```

10.2.15.5 MillerRabin()

This function performs a Miller-Rabin test from FIPS 186-3. It does *iterations* trials on the number. In all likelihood, if the number is not prime, the first test fails.

Return Value	Meaning
TRUE	probably prime
FALSE	composite

```
BOOL
MillerRabin(
    bigNum          bnW,
    RAND_STATE      *rand
    )
{
    BN_MAX(bnWm1);
    BN_PRIME(bnM);
    BN_PRIME(bnB);
    BN_PRIME(bnZ);
    BOOL            ret = FALSE;   // Assumed composite for easy exit
    unsigned int    a;
    unsigned int    j;
    int             wLen;
    int             i;
    int             iterations = MillerRabinRounds(BnSizeInBits(bnW));
//
   INSTRUMENT_INC(MillerRabinTrials[PrimeIndex]);
    pAssert(bnW->size > 1);
    // Let a be the largest integer such that 2^a divides w1.
    BnSubWord(bnWm1, bnW, 1);
    pAssert(bnWm1->size != 0);
    // Since w is odd (w-1) is even so start at bit number 1 rather than 0
    // Get the number of bits in bnWm1 so that it doesn't have to be recomputed
    // on each iteration.
    i = bnWm1->size * RADIX_BITS;
  // Now find the largest power of 2 that divides w1
    for(a = 1;
    (a < (bnWm1->size * RADIX_BITS)) &&
        (BnTestBit(bnWm1, a) == 0);
        a++);
 // 2. m = (w1) / 2^a
        BnShiftRight(bnM, bnWm1, a);
      // 3. wlen = len (w).
    wLen = BnSizeInBits(bnW);
```

```
136       // 4. For i = 1 to iterations do
137         for(i = 0; i < iterations; i++)
138         {
139           // 4.1 Obtain a string b of wlen bits from an RBG.
140           // Ensure that 1 < b < w1.
141             do
142             {
143                 BnGetRandomBits(bnB, wLen, rand);
144                 // 4.2 If ((b <= 1) or (b >= w1)), then go to step 4.1.
145             } while((BnUnsignedCmpWord(bnB, 1) <= 0)
146                     || (BnUnsignedCmp(bnB, bnWm1) >= 0));
147          // 4.3 z = b^m mod w.
148          // if ModExp fails, then say this is not
149          // prime and bail out.
150             BnModExp(bnZ, bnB, bnM, bnW);
151          // 4.4 If ((z == 1) or (z = w == 1)), then go to step 4.7.
152             if((BnUnsignedCmpWord(bnZ, 1) == 0)
153                || (BnUnsignedCmp(bnZ, bnWm1) == 0))
154                 goto step4point7;
155          // 4.5 For j = 1 to a  1 do.
156             for(j = 1; j < a; j++)
157             {
158               // 4.5.1 z = z^2 mod w.
159                 BnModMult(bnZ, bnZ, bnZ, bnW);
160               // 4.5.2 If (z = w1), then go to step 4.7.
161                 if(BnUnsignedCmp(bnZ, bnWm1) == 0)
162                     goto step4point7;
163               // 4.5.3 If (z = 1), then go to step 4.6.
164                 if(BnEqualWord(bnZ, 1))
165                     goto step4point6;
166             }
167             // 4.6 Return COMPOSITE.
168     step4point6:
169             INSTRUMENT_INC(failedAtIteration[i]);
170             goto end;
171           // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
172     step4point7:
173             continue;
174         }
175       // 5. Return PROBABLY PRIME
176         ret = TRUE;
177     end:
178         return ret;
179     }
180     #ifdef TPM_ALG_RSA  //%
```

10.2.15.6 RsaCheckPrime()

This will check to see if a number is prime and appropriate for an RSA prime.

This has different functionality based on whether we are using key sieving or not. If not, the number checked to see if it is divisible by the public exponent, then the number is adjusted either up or down in order to make it a better candidate. It is then checked for being probably prime.

If sieving is used, the number is used to root a sieving process.

```
181     TPM_RC
182     RsaCheckPrime(
183         bigNum           prime,
184         UINT32           exponent,
185         RAND_STATE      *rand
186         )
187     {
188     #ifndef RSA_KEY_SIEVE
```

```
189         TPM_RC          retVal = TPM_RC_SUCCESS;
190         UINT32          modE = BnModWord(prime, exponent);
191         NOT_REFERENCED(rand);
192         if(modE == 0)
193             // evenly divisible so add two keeping the number odd
194             BnAddWord(prime, prime, 2);
195         // want 0 != (p - 1) mod e
196         // which is 1 != p mod e
197         else if(modE == 1)
198             // subtract 2 keeping number odd and insuring that
199             // 0 != (p - 1) mod e
200             BnSubWord(prime, prime, 2);
201         if(BnIsProbablyPrime(prime, rand) == 0)
202             ERROR_RETURN(TPM_RC_VALUE);
203     Exit:
204         return retVal;
205     #else
206         return PrimeSelectWithSieve(prime, exponent, rand);
207     #endif
208     }
```

10.2.15.7 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.

```
209     LIB_EXPORT void
210     RsaAdjustPrimeCandidate(
211         bigNum          prime
212         )
213     {
214         UINT16  highBytes;
215         crypt_uword_t       *msw = &prime->d[prime->size - 1];
216     #define MASK (MAX_CRYPT_UWORD >> (RADIX_BITS - 16))
217         highBytes = *msw >> (RADIX_BITS - 16);
218         // This is fixed point arithmetic on 16-bit values
219         highBytes = ((UINT32)highBytes * (UINT32)0x4AFB) >> 16;
220         highBytes += 0xB505;
221         *msw = ((crypt_uword_t)(highBytes) << (RADIX_BITS - 16)) + (*msw & MASK);
222         prime->d[0] |= 1;
223     }
```

10.2.15.8 BnGeneratePrimeForRSA()

Function to generate a prime of the desired size with the proper attributes for an RSA prime.

```
224     void
225     BnGeneratePrimeForRSA(
226         bigNum          prime,
227         UINT32          bits,
228         UINT32          exponent,
229         RAND_STATE      *rand
230         )
231     {
232         BOOL            found = FALSE;
233     //
```



```
234         // Make sure that the prime is large enough
235         pAssert(prime->allocated >= BITS_TO_CRYPT_WORDS(bits));
236         // Only try to handle specific sizes of keys in order to save overhead
237         pAssert((bits % 32) == 0);
238         prime->size = BITS_TO_CRYPT_WORDS(bits);
239         while(!found)
240         {
241             DRBG_Generate(rand, (BYTE *)prime->d, (UINT16)BITS_TO_BYTES(bits));
242             RsaAdjustPrimeCandidate(prime);
243             found = RsaCheckPrime(prime, exponent, rand) == TPM_RC_SUCCESS;
244         }
245     }
246     #endif //% TPM_ALG_RSA
```

10.2.16 CryptPrimeSieve.c

10.2.16.1 Includes and defines

```
1     #include "Tpm.h"
2     #define RSA_KEY_SIEVE
3     #if defined RSA_KEY_SIEVE   //%
4     #include "CryptPrimeSieve_fp.h"
```

This determines the number of bits in the largest sieve field.

```
5     #define MAX_FIELD_SIZE  2048
6     extern const uint32_t      s_LastPrimeInTable;
7     extern const uint32_t      s_PrimeTableSize;
8     extern const uint32_t      s_PrimesInTable;
9     extern const unsigned char s_PrimeTable[];
```

This table is set of prime markers. Each entry is the prime value for the ((n + 1) * 1024) prime. That is, the entry in *s_PrimeMarkers*[1] is the value for the 2,048th prime. This is used in the PrimeSieve() to adjust the limit for the prime search. When processing smaller prime candidates, fewer primes are checked directly before going to Miller-Rabin. As the prime grows, it is worth spending more time eliminating primes as, a) the density is lower, and b) the cost of Miller-Rabin is higher.

```
10    const uint32_t      s_PrimeMarkersCount = 6;
11    const uint32_t      s_PrimeMarkers[] = {
12        8167, 17881, 28183, 38891, 49871, 60961 };
13    uint32_t   primeLimit;
```

10.2.16.2 Functions

10.2.16.2.1 RsaAdjustPrimeLimit()

This used during the sieve process. The iterator for getting the next prime *(RsaNextPrime*()) will return primes until it hits the limit *(primeLimit*) set up by this function. This causes the sieve process to stop when an appropriate number of primes have been sieved.

```
14    LIB_EXPORT void
15    RsaAdjustPrimeLimit(
16        uint32_t        requestedPrimes
17        )
18    {
19        if(requestedPrimes == 0 || requestedPrimes > s_PrimesInTable)
20            requestedPrimes = s_PrimesInTable;
21        requestedPrimes = (requestedPrimes - 1) / 1024;
22        if(requestedPrimes < s_PrimeMarkersCount)
```

```
23          primeLimit = s_PrimeMarkers[requestedPrimes];
24      else
25          primeLimit = s_LastPrimeInTable;
26      primeLimit >>= 1;
27  }
```

10.2.16.2.2 RsaNextPrime()

This the iterator used during the sieve process. The input is the last prime returned (or any starting point) and the output is the next higher prime. The function returns 0 when the *primeLimit* is reached.

```
28  LIB_EXPORT uint32_t
29  RsaNextPrime(
30      uint32_t    lastPrime
31      )
32  {
33      if(lastPrime == 0)
34          return 0;
35      lastPrime >>= 1;
36      for(lastPrime += 1; lastPrime <= primeLimit; lastPrime++)
37      {
38          if(((s_PrimeTable[lastPrime >> 3] >> (lastPrime & 0x7)) & 1) == 1)
39              return ((lastPrime << 1) + 1);
40      }
41      return 0;
42  }
```

This table contains a previously 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.

```
43  const BYTE   seedValues[] = {
44      0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c,
45      0x99, 0x96, 0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96,
46      0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
47      0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
48      0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
49      0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
50      0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
51      0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0x69, 0xc9, 0x34,
52      0x25, 0xda, 0x22, 0x65, 0x99, 0xb4, 0x4c, 0x1b,
53      0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
54      0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6,
55      0x25, 0xd3, 0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2,
56      0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
57      0xd1};
58  #define USE_NIBBLE
59  #ifndef USE_NIBBLE
60  static const BYTE bitsInByte[256] = {
61      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
62      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
63      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
64      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
65      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
66      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
67      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
68      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
69      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
70      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
71      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
72      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
73      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
74      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
75      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
```



```
76      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
77      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
78      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
79      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
80      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
81      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
82      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
83      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
84      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
85      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
86      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
87      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
88      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
89      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
90      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
91      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
92      0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08
93  };
94  #define BitsInByte(x)   bitsInByte[(unsigned char)x]
95  #else
96  const BYTE bitsInNibble[16] = {
97      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
98      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04};
99  #define BitsInByte(x)                                           \
100             (bitsInNibble[(unsigned char)(x) & 0xf]             \
101         +   bitsInNibble[((unsigned char)(x) >> 4) & 0xf])
102 #endif
```

10.2.16.2.3 BitsInArry()

This function counts the number of bits set in an array of bytes.

```
103 static int
104 BitsInArray(
105     const unsigned char     *a,             // IN: A pointer to an array of bytes
106     unsigned int             aSize          // IN: the number of bytes to sum
107     )
108 {
109     int     j = 0;
110     for(; aSize; a++, aSize--)
111         j += BitsInByte(*a);
112     return j;
113 }
```

10.2.16.2.4 FindNthSetBit()

This function finds the nth SET bit in a bit array. The *n* parameter is between 1 and the number of bits in the array (always a multiple of 8). If called when the array does not have n bits set, it will return -1

Return Value	Meaning
<0	no bit is set or no bit with the requested number is set
>=0	the number of the bit in the array that is the nth set

```
114 LIB_EXPORT int
115 FindNthSetBit(
116     const UINT16     aSize,         // IN: the size of the array to check
117     const BYTE      *a,             // IN: the array to check
118     const UINT32     n              // IN, the number of the SET bit
119     )
120 {
121     UINT16       i;
```

```
122         int             retValue;
123         UINT32          sum = 0;
124         BYTE            sel;
125         //find the bit
126         for(i = 0; (i < (int)aSize) && (sum < n); i++)
127             sum += BitsInByte(a[i]);
128         i--;
129         // The chosen bit is in the byte that was just accessed
130         // Compute the offset to the start of that byte
131         retValue = i * 8 - 1;
132         sel = a[i];
133         // Subtract the bits in the last byte added.
134         sum -= BitsInByte(sel);
135         // Now process the byte, one bit at a time.
136         for(; (sel != 0) && (sum != n); retValue++, sel = sel >> 1)
137             sum += (sel & 1) != 0;
138         return (sum == n) ? retValue : -1;
139     }
140     typedef struct
141     {
142         UINT16      prime;
143         UINT16      count;
144     } SIEVE_MARKS;
145     const SIEVE_MARKS sieveMarks[5] = {
146         {31, 7}, {73, 5}, {241, 4}, {1621, 3}, {UINT16_MAX, 2}};
```

10.2.16.2.5 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 precomputed 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 precomputed field to be used without modification.

To get better performance, one could address the issue of developing the composite numbers. When the size of the prime gets large, the time for doing the divisions goes up, noticeably. It could be better to develop larger composite numbers even if they need to be *bigNum*'s themselves. The object would be to reduce the number of times that the large prime is divided into a few large divides and then use smaller divides to get to the final 16 bit (or smaller) remainders.

```
147     LIB_EXPORT UINT32
148     PrimeSieve(
149         bigNum           bnN,       // IN/OUT: number to sieve
150         UINT32           fieldSize, // IN: size of the field area in bytes
151         BYTE            *field      // IN: field
152         )
153     {
154         INT32            i;
155         UINT32           j;
156         UINT32           fieldBits = fieldSize * 8;
157         UINT32           r;
158         BYTE            *pField;
159         INT32            iter;
160         UINT32           adjust;
161         UINT32           mark = 0;
162         UINT32           count = sieveMarks[0].count;
163         UINT32           stop = sieveMarks[0].prime;
164         UINT32           composite;
165         UINT32           pList[8];
166         UINT32           next;
167         pAssert(field != NULL && bnN != NULL);
168         // If the remainder is odd, then subtracting the value will give an even number,
169         // but we want an odd number, so subtract the 105+rem. Otherwise, just subtract
170         // the even remainder.
```

```
171         adjust = BnModWord(bnN, 105);
172         if(adjust & 1)
173             adjust += 105;
174         // Adjust the input number so that it points to the first number in a
175         // aligned field.
176         BnSubWord(bnN, bnN, adjust);
177     //    pAssert(BnModWord(bnN, 105) == 0);
178         pField = field;
179         for(i = fieldSize; i >= sizeof(seedValues);
180             pField += sizeof(seedValues), i -= sizeof(seedValues))
181         {
182             memcpy(pField, seedValues, sizeof(seedValues));
183         }
184         if(i != 0)
185             memcpy(pField, seedValues, i);
186         // Cycle through the primes, clearing bits
187         // Have already done 3, 5, and 7
188         iter = 7;
189     #define NEXT_PRIME(iter)    (iter = RsaNextPrime(iter))
190         // Get the next N primes where N is determined by the mark in the sieveMarks
191         while((composite = NEXT_PRIME(iter)) != 0)
192         {
193             next = 0;
194             i = count;
195             pList[i--] = composite;
196             for(; i > 0; i--)
197             {
198                 next = NEXT_PRIME(iter);
199                 pList[i] = next;
200                 if(next != 0)
201                     composite *= next;
202             }
203             // Get the remainder when dividing the base field address
204             // by the composite
205             composite = BnModWord(bnN, composite);
206             // 'composite' is divisible by the composite components. for each of the
207             // composite components, divide 'composite'. That remainder (r) is used to
208             // pick a starting point for clearing the array. The stride is equal to the
209             // composite component. Note, the field only contains odd numbers. If the
210             // field were expanded to contain all numbers, then half of the bits would
211             // have already been cleared. We can save the trouble of clearing them a
212             // second time by having a stride of 2*next. Or we can take all of the even
213             // numbers out of the field and use a stride of 'next'
214             for(i = count; i > 0; i--)
215             {
216                 next = pList[i];
217                 if(next == 0)
218                     goto done;
219                 r = composite % next;
220             // these computations deal with the fact that the field starts at some
221             // arbitrary offset within the number space. If the field were all numbers,
222             // then we would have gone through some number of bit clearings before we
223             // got to the start of this range. We don't know how many there were before,
224             // but we can tell from the remainder whether we are on an even or odd
225             // stride when we hit the beginning of the table. If we are on an odd stride
226             // (r & 1), we would start half a stride in (next - r)/2. If we are on an
227             // even stride, we need 1.5 strides (next + r/2) because the table only has
228             // odd numbers. If the remainder happens to be zero, then the start of the
229             // table is on stride so no adjustment is necessary.
230                 if(r & 1)           j = (next - r) / 2;
231                 else if(r == 0)     j = 0;
232                 else                 j = next - r / 2;
233                 for(; j < fieldBits; j += next)
234                     ClearBit(j, field, fieldSize);
235             }
236             if(next >= stop)
```

```
237             {
238                 mark++;
239                 count = sieveMarks[mark].count;
240                 stop = sieveMarks[mark].prime;
241             }
242         }
243     done:
244         INSTRUMENT_INC(totalFieldsSieved[PrimeIndex]);
245         i = BitsInArray(field, fieldSize);
246         INSTRUMENT_ADD(bitsInFieldAfterSieve[PrimeIndex], i);
247         INSTRUMENT_ADD(emptyFieldsSieved[PrimeIndex], (i == 0));
248         return i;
249     }
250     #ifdef SIEVE_DEBUG
251     static uint32_t fieldSize = 210;
```

10.2.16.2.6 SetFieldSize()

Function to set the field size used for prime generation. Used for tuning.

```
252     LIB_EXPORT uint32_t
253     SetFieldSize(
254         uint32_t        newFieldSize
255         )
256     {
257         if(newFieldSize == 0 || newFieldSize > MAX_FIELD_SIZE)
258             fieldSize = MAX_FIELD_SIZE;
259         else
260             fieldSize = newFieldSize;
261         return fieldSize;
262     }
263     #endif // SIEVE_DEBUG
```

10.2.16.2.7 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.

Error Returns	Meaning
TPM_RC_SUCCESS	candidate is probably prime
TPM_RC_NO_RESULT	candidate is not prime and couldn't find and alternative in in the field

```
264     LIB_EXPORT TPM_RC
265     PrimeSelectWithSieve(
266         bigNum          candidate,          // IN/OUT: The candidate to filter
267         UINT32          e,                  // IN: the exponent
268         RAND_STATE      *rand               // IN: the random number generator state
269         )
270     {
271         BYTE            field[MAX_FIELD_SIZE];
272         UINT32          first;
273         UINT32          ones;
274         INT32           chosen;
275         BN_PRIME(test);
276         UINT32          modE;
277     #ifndef SIEVE_DEBUG
```

```
278         UINT32          fieldSize = MAX_FIELD_SIZE;
279     #endif
280         UINT32          primeSize;
281     //
282         // Adjust the field size and prime table list to fit the size of the prime
283         // being tested. This is done to try to optimize the trade-off between the
284         // dividing done for sieving and the time for Miller-Rabin. When the size
285         // of the prime is large, the cost of Miller-Rabin is fairly high, as is the
286         // cost of the sieving. However, the time for Miller-Rabin goes up considerably
287         // faster than the cost of dividing by a number of primes.
288         primeSize = BnSizeInBits(candidate);
289         if(primeSize <= 512)
290         {
291             RsaAdjustPrimeLimit(1024); // Use just the first 1024 primes
292         }
293         else if(primeSize <= 1024)
294         {
295             RsaAdjustPrimeLimit(4096); // Use just the first 4K primes
296         }
297         else
298         {
299             RsaAdjustPrimeLimit(0);     // Use all available
300         }
301         // Save the low-order word to use as a search generator and make sure that
302         // it has some interesting range to it
303         first = candidate->d[0] | 0x80000000;
304         // Sieve the field
305         ones = PrimeSieve(candidate, fieldSize, field);
306         pAssert(ones > 0 && ones < (fieldSize * 8));
307         for(; ones > 0; ones--)
308         {
309             // Decide which bit to look at and find its offset
310             chosen = FindNthSetBit((UINT16)fieldSize, field, ((first % ones) + 1));
311             if((chosen < 0) || (chosen >= (INT32)(fieldSize * 8)))
312                 FAIL(FATAL_ERROR_INTERNAL);
313             // Set this as the trial prime
314             BnAddWord(test, candidate, (crypt_uword_t)(chosen * 2));
315             // The exponent might not have been one of the tested primes so
316             // make sure that it isn't divisible and make sure that 0 != (p-1) mod e
317             // Note: This is the same as 1 != p mod e
318             modE = BnModWord(test, e);
319             if((modE != 0) && (modE != 1) && MillerRabin(test, rand))
320             {
321                 BnCopy(candidate, test);
322                 return TPM_RC_SUCCESS;
323             }
324             // Clear the bit just tested
325             ClearBit(chosen, field, fieldSize);
326         }
327         // Ran out of bits and couldn't find a prime in this field
328         INSTRUMENT_INC(noPrimeFields[PrimeIndex]);
329         return TPM_RC_NO_RESULT;
330     }
331     #ifdef RSA_INSTRUMENT
332     static char             a[256];
333     char *
334     PrintTuple(
335         UINT32      *i
336         )
337     {
338         sprintf(a, "{%d, %d, %d}", i[0], i[1], i[2]);
339         return a;
340     }
341     #define CLEAR_VALUE(x)    memset(x, 0, sizeof(x))
342     void
343     RsaSimulationEnd(
```

```
344         void
345         )
346     {
347         int         i;
348         UINT32      averages[3];
349         UINT32      nonFirst = 0;
350         if((PrimeCounts[0] + PrimeCounts[1] + PrimeCounts[2]) != 0)
351         {
352             printf("Primes generated = %s\n", PrintTuple(PrimeCounts));
353             printf("Fields sieved = %s\n", PrintTuple(totalFieldsSieved));
354             printf("Fields with no primes = %s\n", PrintTuple(noPrimeFields));
355             printf("Primes checked with Miller-Rabin = %s\n",
356                    PrintTuple(MillerRabinTrials));
357             for(i = 0; i < 3; i++)
358                 averages[i] = (totalFieldsSieved[i]
359                                != 0 ? bitsInFieldAfterSieve[i] / totalFieldsSieved[i]
360                                : 0);
361             printf("Average candidates in field %s\n", PrintTuple(averages));
362             for(i = 1; i < (sizeof(failedAtIteration) / sizeof(failedAtIteration[0]));
363             i++)
364                 nonFirst += failedAtIteration[i];
365             printf("Miller-Rabin failures not in first round = %d\n", nonFirst);
366         }
367         CLEAR_VALUE(PrimeCounts);
368         CLEAR_VALUE(totalFieldsSieved);
369         CLEAR_VALUE(noPrimeFields);
370         CLEAR_VALUE(MillerRabinTrials);
371         CLEAR_VALUE(bitsInFieldAfterSieve);
372     }
373     LIB_EXPORT void
374     GetSieveStats(
375         uint32_t        *trials,
376         uint32_t        *emptyFields,
377         uint32_t        *averageBits
378         )
379     {
380         uint32_t        totalBits;
381         uint32_t        fields;
382         *trials = MillerRabinTrials[0] + MillerRabinTrials[1] + MillerRabinTrials[2];
383         *emptyFields = noPrimeFields[0] + noPrimeFields[1] + noPrimeFields[2];
384         fields = totalFieldsSieved[0] + totalFieldsSieved[1]
385             + totalFieldsSieved[2];
386         totalBits = bitsInFieldAfterSieve[0] + bitsInFieldAfterSieve[1]
387             + bitsInFieldAfterSieve[2];
388         if(fields != 0)
389             *averageBits = totalBits / fields;
390         else
391             *averageBits = 0;
392         CLEAR_VALUE(PrimeCounts);
393         CLEAR_VALUE(totalFieldsSieved);
394         CLEAR_VALUE(noPrimeFields);
395         CLEAR_VALUE(MillerRabinTrials);
396         CLEAR_VALUE(bitsInFieldAfterSieve);
397     }
398     #endif
399     #endif //% RSA_KEY_SIEVE
400     #ifndef RSA_INSTRUMENT
401     void
402     RsaSimulationEnd(
403         void
404         )
405     {
406     }
407     #endif
```


10.2.17 CryptRand.c

10.2.17.1 Introduction

This file implements a DRBG with a behavior according to SP800-90A using a block cypher. This is also compliant to ISO/IEC 18031:2011(E) C.3.2.

A state structure is created for use by TPM.lib and functions within the CryptoEngine() my use their own state structures when they need to have deterministic values.

A debug mode is available that allows the random numbers generated for TPM.lib to be repeated during runs of the simulator. The switch for it is in TpmBuildSwitches.h. It is USE_DEBUG_RNG.

```
1   #include "Tpm.h"
```

10.2.17.2 Random Number Generation

This is the implementation layer of CTR DRGB mechanism. The structure of the functions attempts to be maximally close to SP 800-90A. It is intended to be compiled as a separate module that is linked with a secure application so that both reside inside the same boundary [SP 800-90A 8.5]. The secure application in particular manages or accesses protected storage for the state of the DRBG instantiations, and supplies the implementation functions here with a valid pointer to the working state of the given instantiations (as a DRBG_STATE structure).

This DRBG mechanism implementation does not support prediction resistance. Thus *prediction_resistance_flag* is omitted from Instantiate_function(), Reseed_function(), Generate_function() argument lists [SP 800-90A 9.1, 9.2, 9.3], as well as from the working state data structure DRBG_STATE [SP 800-90A 9.1].

This DRBG mechanism implementation always uses the highest security strength of available in the block ciphers. Thus *requested_security_strength* parameter is omitted from Instantiate_function() and Generate_function() argument lists [SP 800-90A 9.1, 9.2, 9.3], as well as from the working state data structure DRBG_STATE [SP 800-90A 9.1].

Internal functions (ones without Crypt prefix) expect validated arguments and therefore use assertions instead of runtime parameter checks and mostly return void instead of a status value. Pull in the test vector definitions and define the space

```
2   #include    "PRNG_TestVectors.h"
3   const BYTE DRBG_NistTestVector_Entropy[] = {DRBG_TEST_INITIATE_ENTROPY};
4   const BYTE DRBG_NistTestVector_GeneratedInterm[] =
5                                   {DRBG_TEST_GENERATED_INTERM};
6   const BYTE DRBG_NistTestVector_EntropyReseed[] =
7                                   {DRBG_TEST_RESEED_ENTROPY};
8   const BYTE DRBG_NistTestVector_Generated[] = {DRBG_TEST_GENERATED};
```

10.2.17.3 Derivation Functions

10.2.17.3.1 Description

The functions in this section are used to reduce the personalization input values to make them usable as input for reseeding and instantiation. The overall behavior is intended to produce the same results as described in SP800-90A, section 10.4.2 "Derivation Function Using a Block Cipher Algorithm *(Block_Cipher_df*())." The code is broken into several subroutines to deal with the fact that the data used for personalization may come in several separate blocks such as a Template hash and a proof value and a primary seed.

10.2.17.3.2 Derivation Function Defines and Structures

```
#define     DF_COUNT (DRBG_KEY_SIZE_WORDS / DRBG_IV_SIZE_WORDS + 1)
#if DRBG_KEY_SIZE_BITS != 128 && DRBG_KEY_SIZE_BITS != 256
#   error "CryptRand.c only written for AES with 128- or 256-bit keys."
#endif
typedef struct
{
    DRBG_KEY_SCHEDULE   keySchedule;
    DRBG_IV             iv[DF_COUNT];
    DRBG_IV             out1;
    DRBG_IV             buf;
    int                 contents;
} DF_STATE, *PDF_STATE;
```

10.2.17.3.3 DfCompute()

This function does the incremental update of the derivation function state. It encrypts the *iv* value and XOR's the results into each of the blocks of the output. This is equivalent to processing all of input data for each output block.

```
static void
DfCompute(
    PDF_STATE        dfState
    )
{
    int              i;
    int              iv;
    crypt_uword_t   *pIv;
    crypt_uword_t    temp[DRBG_IV_SIZE_WORDS] = {0};
//
    for(iv = 0; iv < DF_COUNT; iv++)
    {
        pIv = (crypt_uword_t *)&dfState->iv[iv].words[0];
        for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
        {
            temp[i] ^= pIv[i] ^ dfState->buf.words[i];
        }
        DRBG_ENCRYPT(&dfState->keySchedule, &temp, pIv);
    }
    for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
        dfState->buf.words[i] = 0;
    dfState->contents = 0;
}
```

10.2.17.3.4 DfStart()

This initializes the output blocks with an encrypted counter value and initializes the key schedule.

```
static void
DfStart(
    PDF_STATE        dfState,
    uint32_t         inputLength
    )
{
    BYTE            init[8];
    int             i;
    UINT32          drbgSeedSize = sizeof(DRBG_SEED);
    const BYTE dfKey[] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
    #if DRBG_KEY_SIZE_WORDS > 4
```

```
57          ,0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
58          0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
59      #endif
60      };
61      memset(dfState, 0, sizeof(DF_STATE));
62      DRBG_ENCRYPT_SETUP(&dfKey[0], DRBG_KEY_SIZE_BITS, &dfState->keySchedule);
63      // Create the first chaining values
64      for(i = 0; i < DF_COUNT; i++)
65          ((BYTE *)&dfState->iv[i])[3] = (BYTE)i;
66      DfCompute(dfState);
67      // initialize the first 64 bits of the IV in a way that doesn't depend
68      // on the size of the words used.
69      UINT32_TO_BYTE_ARRAY(inputLength, init);
70      UINT32_TO_BYTE_ARRAY(drbgSeedSize, &init[4]);
71      memcpy(&dfState->iv[0], init, 8);
72      dfState->contents = 4;
73  }
```

10.2.17.3.5 DfUpdate()

This updates the state with the input data. A byte at a time is moved into the state buffer until it is full and then that block is encrypted by DfCompute().

```
74  static void
75  DfUpdate(
76      PDF_STATE        dfState,
77      int              size,
78      const BYTE      *data
79      )
80  {
81      while(size > 0)
82      {
83          int         toFill = DRBG_IV_SIZE_BYTES - dfState->contents;
84          if(size < toFill)
85              toFill = size;
86          // Copy as many bytes as there are or until the state buffer is full
87          memcpy(&dfState->buf.bytes[dfState->contents], data, toFill);
88          // Reduce the size left by the amount copied
89          size -= toFill;
90          // Advance the data pointer by the amount copied
91          data += toFill;
92          // increase the buffer contents count by the amount copied
93          dfState->contents += toFill;
94          pAssert(dfState->contents <= DRBG_IV_SIZE_BYTES);
95          // If we have a full buffer, do a computation pass.
96          if(dfState->contents == DRBG_IV_SIZE_BYTES)
97              DfCompute(dfState);
98      }
99  }
```

10.2.17.3.6 DfEnd()

This function is called to get the result of the derivation function computation. If the buffer is not full, it is padded with zeros. The output buffer is structured to be the same as a DRBG_SEED value so that the function can return a pointer to the DRBG_SEED value in the DF_STATE structure.

```
100 static DRBG_SEED *
101 DfEnd(
102     PDF_STATE        dfState
103     )
104 {
105     // Since DfCompute is always called when a buffer is full, there is always
```

```
106         // space in the buffer for the terminator
107         dfState->buf.bytes[dfState->contents++] = 0x80;
108         // If the buffer is not full, pad with zeros
109         while(dfState->contents < DRBG_IV_SIZE_BYTES)
110             dfState->buf.bytes[dfState->contents++] = 0;
111         // Do a final state update
112         DfCompute(dfState);
113         return (DRBG_SEED *)&dfState->iv;
114     }
```

10.2.17.3.7 DfBuffer()

Function to take an input buffer and do the derivation function to produce a DRBG_SEED value that can be used in DRBG_Reseed();

```
115     static DRBG_SEED *
116     DfBuffer(
117         DRBG_SEED       *output,        // OUT: receives the result
118         int              size,          // IN: size of the buffer to add
119         BYTE            *buf            // IN: address of the buffer
120         )
121     {
122         DF_STATE        dfState;
123         if(size == 0 || buf == NULL)
124             return NULL;
125         // Initialize the derivation function
126         DfStart(&dfState, size);
127         DfUpdate(&dfState, size, buf);
128         DfEnd(&dfState);
129         memcpy(output, &dfState.iv[0], sizeof(DRBG_SEED));
130         return output;
131     }
```

10.2.17.3.8 DRBG_GetEntropy()

Even though this implementation never fails, it may get blocked indefinitely long in the call to get entropy from the platform *(DRBG_GetEntropy32*()). This function is only used during instantiation of the DRBG for manufacturing and on each start-up after an non-orderly shutdown.

Return Value	Meaning
TRUE	Requested entropy returned
FALSE	Entropy Failure

```
132     BOOL
133     DRBG_GetEntropy(
134         UINT32           requiredEntropy,   // IN: requested number of bytes of full
135                                             //     entropy
136         BYTE            *entropy            // OUT: buffer to return collected entropy
137         )
138     {
139     #ifndef USE_DEBUG_RNG
140         UINT32       obtainedEntropy;
141         INT32        returnedEntropy;
142     // If in debug mode, always use the self-test values for initialization
143         if(IsSelfTest())
144         {
145     #endif
146             // If doing simulated DRBG, then check to see if the
147             // entropyFailure condition is being tested
148             if(!IsEntropyBad())
```

```
149         {
150             // In self-test, the caller should be asking for exactly the seed
151             // size of entropy.
152             pAssert(requiredEntropy == sizeof(DRBG_NistTestVector_Entropy));
153             memcpy(entropy, DRBG_NistTestVector_Entropy,
154                    sizeof(DRBG_NistTestVector_Entropy));
155         }
156 #ifndef USE_DEBUG_RNG
157     }
158     else if(!IsEntropyBad())
159     {
160         // Collect entropy
161         // Note: In debug mode, the only "entropy" value ever returned
162         // is the value of the self-test vector.
163         for(returnedEntropy = 1, obtainedEntropy = 0;
164         obtainedEntropy < requiredEntropy && !IsEntropyBad();
165             obtainedEntropy += returnedEntropy)
166         {
167             returnedEntropy = _plat__GetEntropy(&entropy[obtainedEntropy],
168                                                 requiredEntropy - obtainedEntropy);
169             if(returnedEntropy <= 0)
170                 SetEntropyBad();
171         }
172     }
173 #endif
174     return !IsEntropyBad();
175 }
```

10.2.17.3.9 IncrementIv()

Used by EncryptDRBG()

```
176 void
177 IncrementIv(
178     DRBG_IV         *iv
179     )
180 {
181     BYTE      *ivP = ((BYTE *)iv) + DRBG_IV_SIZE_BYTES;
182     while((--ivP >= (BYTE *)iv) && ((*ivP = ((*ivP + 1) & 0xFF)) == 0));
183 }
```

10.2.17.3.10 EncryptDRBG()

This does the encryption operation for the DRBG. It will encrypt the input state counter (IV) using the state key. Into the output buffer for as many times as it takes to generate the required number of bytes.

```
184 void
185 EncryptDRBG(
186     BYTE                *dOut,
187     UINT32               dOutBytes,
188     DRBG_KEY_SCHEDULE   *keySchedule,
189     DRBG_IV             *iv,
190     UINT32              *lastValue      // Points to the last output value
191     )
192 {
193 #ifdef FIPS_COMPLIANT
194 // For FIPS compliance, the DRBG has to do a continuous self-test to make sure that
195 // no two consecutive values are the same. This overhead is not incurred if the TPM
196 // is not required to be FIPS compliant
197 //
198     UINT32           temp[DRBG_IV_SIZE_BYTES / sizeof(UINT32)];
199     int              i;
200     BYTE            *p;
```

```
201         for(; dOutBytes > 0;)
202         {
203             // Increment the IV before each encryption (this is what makes this
204             // different from normal counter-mode encryption
205             IncrementIv(iv);
206             DRBG_ENCRYPT(keySchedule, iv, temp);
207     // Expect a 16 byte block
208     #if DRBG_IV_SIZE_BITS != 128
209     #error  "Unsuppored IV size in DRBG"
210     #endif
211             if((lastValue[0] == temp[0])
212                && (lastValue[1] == temp[1])
213                && (lastValue[2] == temp[2])
214                && (lastValue[3] == temp[3])
215                )
216                 FAIL(FATAL_ERROR_DRBG);
217             lastValue[0] = temp[0];
218             lastValue[1] = temp[1];
219             lastValue[2] = temp[2];
220             lastValue[3] = temp[3];
221             i = MIN(dOutBytes, DRBG_IV_SIZE_BYTES);
222             dOutBytes -= i;
223             for(p = (BYTE *)temp; i > 0; i--)
224                 *dOut++ = *p++;
225         }
226     #else // version without continuous self-test
227         NOT_REFERENCED(lastValue);
228         for(; dOutBytes >= DRBG_IV_SIZE_BYTES;
229         dOut = &dOut[DRBG_IV_SIZE_BYTES], dOutBytes -= DRBG_IV_SIZE_BYTES)
230         {
231             // Increment the IV
232             IncrementIv(iv);
233             DRBG_ENCRYPT(keySchedule, iv, dOut);
234         }
235         // If there is a partial, generate into a block-sized
236         // temp buffer and copy to the output.
237         if(dOutBytes != 0)
238         {
239             BYTE        temp[DRBG_IV_SIZE_BYTES];
240             // Increment the IV
241             IncrementIv(iv);
242             DRBG_ENCRYPT(keySchedule, iv, temp);
243             memcpy(dOut, temp, dOutBytes);
244         }
245     #endif
246     }
```

10.2.17.3.11 DRBG_Update()

This function performs the state update function. According to SP800-90A, a temp value is created by doing CTR mode encryption of *providedData* and replacing the key and IV with these values. The one difference is that, with counter mode, the IV is incremented after each block is encrypted and in this operation, the counter is incremented before each block is encrypted. This function implements an *optimized* version of the algorithm in that it does the update of the *drbgState*->seed in place and then *providedData* is XORed() into *drbgState*->seed to complete the encryption of *providedData*. This works because the IV is the last thing that gets encrypted.

```
247     void
248     DRBG_Update(
249         DRBG_STATE          *drbgState,     // IN:OUT state to update
250         DRBG_KEY_SCHEDULE   *keySchedule,   // IN: the key schedule (optional)
251         DRBG_SEED           *providedData   // IN: additional data
252         )
```

```
{
    UINT32               i;
    BYTE                *temp = (BYTE *)&drbgState->seed;
    DRBG_KEY            *key = pDRBG_KEY(&drbgState->seed);
    DRBG_IV             *iv = pDRBG_IV(&drbgState->seed);
    DRBG_KEY_SCHEDULE    localKeySchedule;
//
    pAssert(drbgState->magic == DRBG_MAGIC);
    // If an key schedule was not provided, make one
    if(keySchedule == NULL)
    {
        if(DRBG_ENCRYPT_SETUP((BYTE *)key,
                              DRBG_KEY_SIZE_BITS, &localKeySchedule) != 0)
            FAIL(FATAL_ERROR_INTERNAL);
        keySchedule = &localKeySchedule;
    }
    // Encrypt the temp value
    EncryptDRBG(temp, sizeof(DRBG_SEED), keySchedule, iv,
                drbgState->lastValue);
    if(providedData != NULL)
    {
        BYTE        *pP = (BYTE *)providedData;
        for(i = DRBG_SEED_SIZE_BYTES; i != 0; i--)
            *temp++ ^= *pP++;
    }
    // Since temp points to the input key and IV, we are done and
    // don't need to copy the resulting 'temp' to drbgState->seed
}
```

10.2.17.3.12 DRBG_Reseed()

This function is used when reseeding of the DRBG is required. If entropy is provided, it is used in lieu of using hardware entropy.

NOTE: the provided entropy must be the required size.

Return Value	Meaning
TRUE	reseed succeeded
FALSE	reseed failed, probably due to the entropy generation

```
BOOL
DRBG_Reseed(
    DRBG_STATE          *drbgState,         // IN: the state to update
    DRBG_SEED           *providedEntropy,   // IN: entropy
    DRBG_SEED           *additionalData     // IN:
    )
{
    DRBG_SEED            seed;
    BYTE                *pSeed = (BYTE *)&seed;
    pAssert((drbgState != NULL) && (drbgState->magic == DRBG_MAGIC));
    if(providedEntropy == NULL)
    {
        if(!DRBG_GetEntropy(sizeof(DRBG_SEED), pSeed))
            return FALSE;
        providedEntropy = &seed;
    }
    if(additionalData != NULL)
    {
        BYTE        *in1 = (BYTE *)providedEntropy; // This might be seed
        BYTE        *in2 = (BYTE *)additionalData;
        int          i;
        // XOR the provided data with the seed
```

```
303             for(i = sizeof(DRBG_SEED); i > 0; i--)
304                 *pSeed++ = *in1++ ^ *in2++;
305         }
306         DRBG_Update(drbgState, NULL, providedEntropy);
307         drbgState->reseedCounter = 1;
308         return TRUE;
309     }
```

10.2.17.3.13 DRBG_SelfTest()

This is run when the DRBG is instantiated and at startup

Return Value	Meaning
FALSE	test failed
TRUE	test OK

```
310     BOOL
311     DRBG_SelfTest(
312         void
313         )
314     {
315         BYTE             buf[sizeof(DRBG_NistTestVector_Generated)];
316         DRBG_SEED        seed;
317         UINT32           i;
318         BYTE            *p;
319         DRBG_STATE       testState;
320     //
321         pAssert(!IsSelfTest());
322         SetSelfTest();
323         SetDrbgTested();
324         // Do an instantiate
325         if(!DRBG_Instantiate(&testState, 0, NULL))
326             return FALSE;
327     #if defined DRBG_DEBUG_PRINT && defined DEBUG
328         dbgDumpMemBlock(pDRBG_KEY(&testState), DRBG_KEY_SIZE_BYTES,
329                         "Key after Instantiate");
330         dbgDumpMemBlock(pDRBG_IV(&testState), DRBG_IV_SIZE_BYTES,
331                         "Value after Instantiate");
332     #endif
333         if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
334             return FALSE;
335     #if defined DRBG_DEBUG_PRINT && defined DEBUG
336         dbgDumpMemBlock(pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
337                         "Key after 1st Generate");
338         dbgDumpMemBlock(pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
339                         "Value after 1st Generate");
340     #endif
341         if(memcmp(buf, DRBG_NistTestVector_GeneratedInterm, sizeof(buf)) != 0)
342             return FALSE;
343         memcpy(seed.bytes, DRBG_NistTestVector_EntropyReseed, sizeof(seed));
344         DRBG_Reseed(&testState, &seed, NULL);
345     #if defined DRBG_DEBUG_PRINT && defined DEBUG
346         dbgDumpMemBlock((BYTE *)pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
347                         "Key after 2nd Generate");
348         dbgDumpMemBlock((BYTE *)pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
349                         "Value after 2nd Generate");
350         dbgDumpMemBlock(buf, sizeof(buf), "2nd Generated");
351     #endif
352         if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
353             return FALSE;
354         if(memcmp(buf, DRBG_NistTestVector_Generated, sizeof(buf)) != 0)
355             return FALSE;
```



```
356         ClearSelfTest();
357         DRBG_Uninstantiate(&testState);
358         for(p = (BYTE *)&testState, i = 0; i < sizeof(DRBG_STATE); i++)
359         {
360             if(*p++)
361                 return FALSE;
362         }
363         // Simulate hardware failure to make sure that we get an error when
364         // trying to instantiate
365         SetEntropyBad();
366         if(DRBG_Instantiate(&testState, 0, NULL))
367            return FALSE;
368         ClearEntropyBad();
369         return TRUE;
370     }
```

10.2.17.4 Public Interface

10.2.17.4.1 Description

The functions in this section are the interface to the RNG. These are the functions that are used by TPM.lib. Other functions are only visible to programs in the LtcCryptoEngine().

10.2.17.4.2 CryptRandomStir()

This function is used to cause a reseed. A DRBG_SEED amount of entropy is collected from the hardware and then additional data is added.

Error Returns	Meaning
TPM_RC_NO_RESULT	S failure of the entropy generator

```
371     LIB_EXPORT TPM_RC
372     CryptRandomStir(
373         INT32            additionalDataSize,
374         BYTE            *additionalData
375         )
376     {
377         DRBG_SEED        tmpBuf;
378         DRBG_SEED        dfResult;
379     //
380     #if defined USE_DEBUG_RNG && 1
381         // If doing debug, use the input data as the initial setting for the RNG state
382         // so that the test can be reset at any time.
383         NOT_REFERENCED(dfResult);
384         if(additionalDataSize < 0)
385             additionalDataSize = 0;
386         else if(additionalDataSize > sizeof(tmpBuf))
387             additionalDataSize = sizeof(tmpBuf);
388         else
389             memset(&tmpBuf.bytes[additionalDataSize], 0, sizeof(tmpBuf) -
    additionalDataSize);
390         memcpy(tmpBuf.bytes, additionalData, additionalDataSize);
391         memcpy(drbgDefault.seed.bytes, tmpBuf.bytes, sizeof(drbgDefault.seed.bytes));
392     #else
393         // All reseed with outside data starts with a buffer full of entropy
394         if(!DRBG_GetEntropy(sizeof(tmpBuf), (BYTE *)&tmpBuf))
395             return TPM_RC_NO_RESULT;
396         DRBG_Reseed(&drbgDefault, &tmpBuf,
397                     DfBuffer(&dfResult, additionalDataSize, additionalData));
398     #endif
399         drbgDefault.reseedCounter = 1;
```

```
400         return TPM_RC_SUCCESS;
401     }
```

10.2.17.4.3 CryptRandomGenerate()

Generate a *randomSize* number or random bytes.

```
402     LIB_EXPORT UINT16
403     CryptRandomGenerate(
404         INT32            randomSize,
405         BYTE            *buffer
406         )
407     {
408         if(randomSize > UINT16_MAX)
409             randomSize = UINT16_MAX;
410         return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, (UINT16)randomSize);
411     }
```

10.2.17.4.4 DRBG_InstantiateSeededKdf()

Function used to instantiate a KDF-based RNG. This is used for derivations

```
412     LIB_EXPORT BOOL
413     DRBG_InstantiateSeededKdf(
414         KDF_STATE       *state,         // IN: buffer to hold the state
415         TPM_ALG_ID       hashAlg,       // IN: hash algorithm
416         TPM_ALG_ID       kdf,           // IN: the KDF to use
417         TPM2B           *seed,          // IN: the seed to use
418         const TPM2B     *label,         // IN: a label for the generation process.
419         TPM2B           *context        // IN: the context value
420         )
421     {
422         state->magic = KDF_MAGIC;
423         state->seed = seed;
424         state->hash = hashAlg;
425         state->kdf = kdf;
426         state->label = label;
427         state->context = context;
428         state->counter = 1;
429         return TRUE;
430     }
```

10.2.17.4.5 DRBG_AdditionalData()

Function to reseed the DRBG with additional **entropy**. This is normally called before computing the protection value of a primary key in the Endorsement hierarchy.

```
431     LIB_EXPORT void
432     DRBG_AdditionalData(
433         DRBG_STATE      *drbgState,     // IN:OUT state to update
434         TPM2B           *additionalData // IN: value to incorporate
435         )
436     {
437         DRBG_SEED        dfResult;
438         if(drbgState->magic == DRBG_MAGIC)
439         {
440             DfBuffer(&dfResult, additionalData->size, additionalData->buffer);
441             DRBG_Reseed(drbgState, &dfResult, NULL);
442         }
443     }
```

10.2.17.4.6 DRBG_InstantiateSeeded()

This function is used to instantiate a random number generator from seed values. The nominal use of this generator is to create sequences of pseudo-random numbers from a seed value.

```
444 LIB_EXPORT BOOL
445 DRBG_InstantiateSeeded(
446     DRBG_STATE      *drbgState,     // IN: buffer to hold the state
447     const TPM2B     *seed,          // IN: the seed to use
448     const TPM2B     *purpose,       // IN: a label for the generation process.
449     const TPM2B     *name,          // IN: name of the object
450     const TPM2B     *additional     // IN: additional data
451     )
452 {
453     DF_STATE         dfState;
454     int              totalInputSize;
455     // DRBG should have been tested, but...
456     if(!IsDrbgTested() && !DRBG_SelfTest())
457         FAIL(FATAL_ERROR_SELF_TEST);
458     // Initialize the DRBG state
459     memset(drbgState, 0, sizeof(DRBG_STATE));
460     drbgState->magic = DRBG_MAGIC;
461     // Size all of the values
462     totalInputSize = (seed != NULL) ? seed->size : 0;
463     totalInputSize += (purpose != NULL) ? purpose->size : 0;
464     totalInputSize += (name != NULL) ? name->size : 0;
465     totalInputSize += (additional != NULL) ? additional->size : 0;
466     // Initialize the derivation
467     DfStart(&dfState, totalInputSize);
468     // Run all the input strings through the derivation function
469     if(seed != NULL)
470         DfUpdate(&dfState, seed->size, seed->buffer);
471     if(purpose != NULL)
472         DfUpdate(&dfState, purpose->size, purpose->buffer);
473     if(name != NULL)
474         DfUpdate(&dfState, name->size, name->buffer);
475     if(additional != NULL)
476         DfUpdate(&dfState, additional->size, additional->buffer);
477     // Used the derivation function output as the "entropy" input. This is not
478     // how it is described in SP800-90A but this is the equivalent function
479     DRBG_Reseed(((DRBG_STATE *)drbgState), DfEnd(&dfState), NULL);
480     return TRUE;
481 }
```

10.2.17.4.7 CryptRandStartup()

This function is called when TPM_Startup() is executed.

```
482 LIB_EXPORT BOOL
483 CryptRandStartup(
484     void
485     )
486 {
487 #ifndef _DRBG_STATE_SAVE
488     // If not saved in NV, re-instantiate on each startup
489     DRBG_Instantiate(&drbgDefault, 0, NULL);
490 #else
491     // If the running state is saved in NV, NV has to be loaded before it can
492     // be updated
493     if(go.drbgState.magic == DRBG_MAGIC)
494         DRBG_Reseed(&go.drbgState, NULL, NULL);
495     else
496         DRBG_Instantiate(&go.drbgState, 0, NULL);
```

```
497 #endif
498     return TRUE;
499 }
```

10.2.17.4.8 CryptRandInit()

This function is called when _TPM_Init() is being processed

```
500 LIB_EXPORT BOOL
501 CryptRandInit(
502     void
503     )
504 {
505     return DRBG_SelfTest();
506 }
```

10.2.17.4.9 DRBG_Generate()

This function generates a random sequence according SP800-90A. If *random* is not NULL, then *randomSize* bytes of random values are generated. If *random* is NULL or *randomSize* is zero, then the function returns TRUE without generating any bits or updating the reseed counter. This function returns 0 if a reseed is required. Otherwise, it returns the number of bytes produced which could be less than the number requested if the request is too large.

```
507 LIB_EXPORT UINT16
508 DRBG_Generate(
509     RAND_STATE      *state,
510     BYTE            *random,        // OUT: buffer to receive the random values
511     UINT16           randomSize     // IN: the number of bytes to generate
512     )
513 {
514 //
515     if(state == NULL)
516         state = (RAND_STATE *)&drbgDefault;
517     // If the caller used a KDF state, generate a sequence from the KDF
518     if(state->kdf.magic == KDF_MAGIC)
519     {
520         KDF_STATE       *kdf = (KDF_STATE *)state;
521         UINT32           count = (UINT32)kdf->counter;
522         if((randomSize != 0) && (random != NULL))
523             CryptKDFa(kdf->hash, kdf->seed, kdf->label, kdf->context, NULL,
524                       randomSize * 8, random, &count, 0);
525         kdf->counter = count;
526         return randomSize;
527     }
528     else if(state->drbg.magic == DRBG_MAGIC)
529     {
530         DRBG_STATE          *drbgState = (DRBG_STATE *)state;
531         DRBG_KEY_SCHEDULE    keySchedule;
532         DRBG_SEED           *seed = &drbgState->seed;
533         if(drbgState->reseedCounter >= CTR_DRBG_MAX_REQUESTS_PER_RESEED)
534         {
535             if(drbgState == &drbgDefault)
536             {
537                 DRBG_Reseed(drbgState, NULL, NULL);
538                 if(IsEntropyBad() && !IsSelfTest())
539                     return 0;
540             }
541             else
542                 // If this is a PRNG then the only way to get
543                 // here is if the SW has run away.
544                 FAIL(FATAL_ERROR_INTERNAL);
```

```
545         }
546         // if the allowed number of bytes in a request is larger than the
547         // less than the number of bytes that can be requested, then check
548 #if UINT16_MAX >=  CTR_DRBG_MAX_BYTES_PER_REQUEST
549         if(randomSize > CTR_DRBG_MAX_BYTES_PER_REQUEST)
550             randomSize = CTR_DRBG_MAX_BYTES_PER_REQUEST;
551 #endif
552         // Create  encryption schedule
553         if(DRBG_ENCRYPT_SETUP((BYTE *)pDRBG_KEY(seed),
554                               DRBG_KEY_SIZE_BITS, &keySchedule) != 0)
555             FAIL(FATAL_ERROR_INTERNAL);
556         // Generate the random data
557         EncryptDRBG(random, randomSize, &keySchedule, pDRBG_IV(seed),
558                     drbgState->lastValue);
559         // Do a key update
560         DRBG_Update(drbgState, &keySchedule, NULL);
561         // Increment the reseed counter
562         drbgState->reseedCounter += 1;
563     }
564     else
565     {
566         FAIL(FATAL_ERROR_INTERNAL);
567     }
568     return randomSize;
569 }
```

10.2.17.4.10 DRBG_Instantiate()

This is CTR_DRBG_Instantiate_algorithm() from [SP 800-90A 10.2.1.3.1]. This is called when a the TPM DRBG is to be instantiated. This is called to instantiate a DRBG used by the TPM for normal operations.

Return Value	Meaning
TRUE	instantiation succeeded
FALSE	instantiation failed

```
570 LIB_EXPORT BOOL
571 DRBG_Instantiate(
572     DRBG_STATE      *drbgState,         // OUT: the instantiated value
573     UINT16           pSize,             // IN: Size of personalization string
574     BYTE            *personalization    // IN: The personalization string
575     )
576 {
577     DRBG_SEED        seed;
578     DRBG_SEED        dfResult;
579 //
580     pAssert((pSize == 0) || (pSize <= sizeof(seed)) || (personalization != NULL));
581     // If the DRBG has not been tested, test when doing an instantiation. Since
582     // Instantiation is called during self test, make sure we don't get stuck in a
583     // loop.
584     if(!IsDrbgTested() && !IsSelfTest() && !DRBG_SelfTest())
585         return FALSE;
586     // If doing a self test, DRBG_GetEntropy will return the NIST
587     // test vector value.
588     if(!DRBG_GetEntropy(sizeof(seed), (BYTE *)&seed))
589         return FALSE;
590     // set everything to zero
591     memset(drbgState, 0, sizeof(DRBG_STATE));
592     drbgState->magic = DRBG_MAGIC;
593     // Steps 1, 2, 3, 6, 7 of SP 800-90A 10.2.1.3.1 are exactly what
594     // reseeding does. So, do a reduction on the personalization value (if any)
595     // and do a reseed.
596     DRBG_Reseed(drbgState, &seed, DfBuffer(&dfResult, pSize, personalization));
```

```
597         return TRUE;
598     }
```

10.2.17.4.11 DRBG_Uninstantiate()

This is Uninstantiate_function() from [SP 800-90A 9.4].

Error Returns	Meaning

```
599     LIB_EXPORT TPM_RC
600     DRBG_Uninstantiate(
601         DRBG_STATE      *drbgState      // IN/OUT: working state to erase
602         )
603     {
604         if((drbgState == NULL) || (drbgState->magic != DRBG_MAGIC))
605             return TPM_RC_VALUE;
606         memset(drbgState, 0, sizeof(DRBG_STATE));
607         return TPM_RC_SUCCESS;
608     }
```

10.2.17.4.12 CryptRandMinMax()

This function generates a value that as not larger than (2^*max*) - 1 and no smaller than 2^(*min* - 1). For example, if *max* == 4 and *min* == 2, then the number will be between 0x0010 and 0x1111 inclusively. If *max* == 4 and *min* == 4 then the number will be between 0x1000 and 0x1111.

```
609     LIB_EXPORT NUMBYTES
610     CryptRandMinMax(
611         BYTE            *out,
612         UINT32           max,
613         UINT32           min,
614         RAND_STATE      *rand
615         )
616     {
617         BN_VAR(bn, LARGEST_NUMBER_BITS);
618         NUMBYTES            size = (NUMBYTES)BITS_TO_BYTES(max);
619         pAssert(max <= LARGEST_NUMBER_BITS);
620         do
621         {
622             BnGetRandomBits(bn, max, rand);
623         } while(BnSizeInBits(bn) < min);
624         BnToBytes(bn, out, &size);
625         return size;
626     }
```

10.2.18 CryptRsa.c

10.2.18.1 Introduction

This file contains implementation of cryptographic primitives for RSA. Vendors may replace the implementation in this file with their own library functions.

10.2.18.2 Includes

Need this define to get the *private* defines for this function

```
1   #define CRYPT_RSA_C
2   #include "Tpm.h"
```

```
#ifdef  TPM_ALG_RSA
```

10.2.18.3 Obligatory Initialization Functions

10.2.18.3.1 CryptRsaInit()

Function called at _TPM_Init().

```
BOOL
CryptRsaInit(
    void
    )
{
    return TRUE;
}
```

10.2.18.3.2 CryptRsaStartup()

Function called at TPM2_Startup()

```
BOOL
CryptRsaStartup(
    void
    )
{
    return TRUE;
}
```

10.2.18.4 Internal Functions

```
void
RsaInitializeExponent(
    privateExponent_t      *pExp
    )
{
#if CRT_FORMAT_RSA == NO
    BN_INIT(pExp->D);
#else
    BN_INIT(pExp->Q);
    BN_INIT(pExp->dP);
    BN_INIT(pExp->dQ);
    BN_INIT(pExp->qInv);
#endif
}
```

10.2.18.4.1 ComputePrivateExponent()

```
static BOOL
ComputePrivateExponent(
    bigNum               P,             // IN: first prime (size is 1/2 of bnN)
    bigNum               Q,             // IN: second prime (size is 1/2 of bnN)
    bigNum               E,             // IN: the public exponent
    bigNum               N,             // IN: the public modulus
    privateExponent_t   *pExp           // OUT:
    )
{
    BOOL                pOK;
    BOOL                qOK;
#if CRT_FORMAT_RSA == NO
    BN_RSA(bnPhi);
```

```
45  //
46      RsaInitializeExponent(pExp);
47      // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
48      pOK = BnCopy(bnPhi, N);
49      pOK = pOK && BnSub(bnPhi, bnPhi, P);
50      pOK = pOK && BnSub(bnPhi, bnPhi, Q);
51      pOK = pOK && BnAddWord(bnPhi, bnPhi, 1);
52      // Compute the multiplicative inverse d = 1/e mod Phi
53      pOK = pOK && BnModInverse((bigNum)&pExp->D, E, bnPhi);
54      qOK = pOK;
55  #else
56      BN_PRIME(temp);
57      bigNum              pT;
58  //
59      NOT_REFERENCED(N);
60      RsaInitializeExponent(pExp);
61      BnCopy((bigNum)&pExp->Q, Q);
62      // make p the larger value so that m2 is always less than p
63      if(BnUnsignedCmp(P, Q) < 0)
64      {
65          pT = P;
66          P = Q;
67          Q = pT;
68      }
69      //dP = (1/e) mod (p-1) = d mod (p-1)
70      pOK = BnSubWord(temp, P, 1);
71      pOK = pOK && BnModInverse((bigNum)&pExp->dP, E, temp);
72      //dQ = (1/e) mod (q-1) = d mod (q-1)
73      qOK = BnSubWord(temp, Q, 1);
74      qOK = qOK && BnModInverse((bigNum)&pExp->dQ, E, temp);
75      // qInv = (1/q) mod p
76      if(pOK && qOK)
77          pOK = qOK = BnModInverse((bigNum)&pExp->qInv, Q, P);
78  #endif
79      if(!pOK)
80          BnSetWord(P, 0);
81      if(!qOK)
82          BnSetWord(Q, 0);
83      return pOK && qOK;
84  }
```

10.2.18.4.2 RsaPrivateKeyOp()

This function is called to do the exponentiation with the private key. Compile options allow use of the simple (but slow) private exponent, or the more complex but faster CRT method.

```
85  static BOOL
86  RsaPrivateKeyOp(
87      bigNum               inOut, // IN/OUT: number to be exponentiated
88      bigNum               N,     // IN: public modulus (can be NULL if CRT)
89      bigNum               P,     // IN: one of the primes (can be NULL if not CRT)
90      privateExponent_t   *pExp
91      )
92  {
93      BOOL                 OK;
94  #if CRT_FORMAT_RSA == NO
95      (P);
96      OK = BnModExp(inOut, inOut, (bigNum)&pExp->D, N);
97  #else
98      BN_RSA(M1);
99      BN_RSA(M2);
100     BN_RSA(M);
101     BN_RSA(H);
102     bigNum              Q = (bigNum)&pExp->Q;
```

```
103         NOT_REFERENCED(N);
104         // Make P the larger prime.
105         // NOTE that when the CRT form of the private key is created, dP will always
106         // be computed using the larger of p and q so the only thing needed here is that
107         // the primes be selected so that they agree with dP.
108         if(BnUnsignedCmp(P, Q) < 0)
109         {
110             bigNum      T = P;
111             P = Q;
112             Q = T;
113         }
114         // m1 = cdP mod p
115         OK = BnModExp(M1, inOut, (bigNum)&pExp->dP, P);
116         // m2 = cdQ mod q
117         OK = OK && BnModExp(M2, inOut, (bigNum)&pExp->dQ, Q);
118         // h = qInv * (m1 - m2) mod p = qInv * (m1 + P - m2) mod P because Q < P
119         // so m2 < P
120         OK = OK && BnSub(H, P, M2);
121         OK = OK && BnAdd(H, H, M1);
122         OK = OK && BnModMult(H, H, (bigNum)&pExp->qInv, P);
123         // m = m2 + h * q
124         OK = OK && BnMult(M, H, Q);
125         OK = OK && BnAdd(inOut, M2, M);
126     #endif
127         return OK;
128     }
```

10.2.18.4.3 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*).

Error Returns	Meaning
TPM_RC_VALUE	number to exponentiate is larger than the modulus

```
129     static TPM_RC
130     RSAEP(
131         TPM2B       *dInOut,        // IN: size of the encrypted block and the size of
132                                     //     the encrypted value. It must be the size of
133                                     //     the modulus.
134                                     // OUT: the encrypted data. Will receive the
135                                     //      decrypted value
136         OBJECT      *key            // IN: the key to use
137         )
138     {
139         TPM2B_TYPE(4BYTES, 4);
140         TPM2B_4BYTES(e) = {{4, {(BYTE)((RSA_DEFAULT_PUBLIC_EXPONENT >> 24) & 0xff),
141                                 (BYTE)((RSA_DEFAULT_PUBLIC_EXPONENT >> 16) & 0xff),
142                                 (BYTE)((RSA_DEFAULT_PUBLIC_EXPONENT >> 8) & 0xff),
143                                 (BYTE)((RSA_DEFAULT_PUBLIC_EXPONENT)& 0xff)}}};
144     //
145         if(key->publicArea.parameters.rsaDetail.exponent != 0)
146             UINT32_TO_BYTE_ARRAY(key->publicArea.parameters.rsaDetail.exponent,
147                                  e.t.buffer);
148         return ModExpB(dInOut->size, dInOut->buffer, dInOut->size, dInOut->buffer,
149                        e.t.size, e.t.buffer, key->publicArea.unique.rsa.t.size,
150                        key->publicArea.unique.rsa.t.buffer);
151     }
```

10.2.18.4.4 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.

Error Returns	Meaning
TPM_RC_SIZE	the value to decrypt is larger than the modulus

```
152  static TPM_RC
153  RSADP(
154      TPM2B           *inOut,         // IN/OUT: the value to encrypt
155      OBJECT          *key            // IN: the key
156      )
157  {
158      BN_RSA_INITIALIZED(bnM, inOut);
159      BN_RSA_INITIALIZED(bnN, &key->publicArea.unique.rsa);
160      BN_RSA_INITIALIZED(bnP, &key->sensitive.sensitive.rsa);
161      if(BnUnsignedCmp(bnM, bnN) >= 0)
162          return TPM_RC_SIZE;
163      // private key operation requires that private exponent be loaded
164      // During self-test, this might not be the case so load it up if it hasn't
165      // already done
166      // been done
167      if(!key->attributes.privateExp)
168          CryptRsaLoadPrivateExponent(key);
169      if(!RsaPrivateKeyOp(bnM, bnN, bnP, &key->privateExponent))
170          FAIL(FATAL_ERROR_INTERNAL);
171      BnTo2B(bnM, inOut, inOut->size);
172      return TPM_RC_SUCCESS;
173  }
```

10.2.18.4.5 OaepEncode()

This function performs OAEP padding. The size of the buffer to receive the OAEP padded data must equal the size of the modulus

Error Returns	Meaning
TPM_RC_VALUE	*hashAlg* is not valid or message size is too large

```
174  static TPM_RC
175  OaepEncode(
176      TPM2B       *padded,        // OUT: the pad data
177      TPM_ALG_ID   hashAlg,       // IN: algorithm to use for padding
178      const TPM2B *label,         // IN: null-terminated string (may be NULL)
179      TPM2B       *message,       // IN: the message being padded
180      RAND_STATE  *rand           // IN: the random number generator to use
181      )
182  {
183      INT32        padLen;
184      INT32        dbSize;
185      INT32        i;
186      BYTE         mySeed[MAX_DIGEST_SIZE];
187      BYTE        *seed = mySeed;
188      INT32        hLen = CryptHashGetDigestSize(hashAlg);
189      BYTE         mask[MAX_RSA_KEY_BYTES];
190      BYTE        *pp;
191      BYTE        *pm;
192      TPM_RC       retVal = TPM_RC_SUCCESS;
```



```
193         pAssert(padded != NULL && message != NULL);
194         // A value of zero is not allowed because the KDF can't produce a result
195         // if the digest size is zero.
196         if(hLen <= 0)
197             return TPM_RC_VALUE;
198         // Basic size checks
199         //  make sure digest isn't too big for key size
200         if(padded->size < (2 * hLen) + 2)
201             ERROR_RETURN(TPM_RC_HASH);
202         // and that message will fit messageSize <= k - 2hLen - 2
203         if(message->size > (padded->size - (2 * hLen) - 2))
204             ERROR_RETURN(TPM_RC_VALUE);
205         // Hash L even if it is null
206         // Offset into padded leaving room for masked seed and byte of zero
207         pp = &padded->buffer[hLen + 1];
208         if(CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
209                           hLen, pp) != hLen)
210             ERROR_RETURN(TPM_RC_FAILURE);
211         // concatenate PS of k  mLen  2hLen  2
212         padLen = padded->size - message->size - (2 * hLen) - 2;
213         MemorySet(&pp[hLen], 0, padLen);
214         pp[hLen + padLen] = 0x01;
215         padLen += 1;
216         memcpy(&pp[hLen + padLen], message->buffer, message->size);
217         // The total size of db = hLen + pad + mSize;
218         dbSize = hLen + padLen + message->size;
219         // If testing, then use the provided seed. Otherwise, use values
220         // from the RNG
221         CryptRandomGenerate(hLen, mySeed);
222         DRBG_Generate(rand, mySeed, (UINT16)hLen);
223         // mask = MGF1 (seed, nSize  hLen  1)
224         CryptMGF1(dbSize, mask, hashAlg, hLen, seed);
225         // Create the masked db
226         pm = mask;
227         for(i = dbSize; i > 0; i--)
228             *pp++ ^= *pm++;
229         pp = &padded->buffer[hLen + 1];
230         // Run the masked data through MGF1
231         if(CryptMGF1(hLen, &padded->buffer[1], hashAlg, dbSize, pp) != (unsigned)hLen)
232             ERROR_RETURN(TPM_RC_VALUE);
233     // Now XOR the seed to create masked seed
234         pp = &padded->buffer[1];
235         pm = seed;
236         for(i = hLen; i > 0; i--)
237             *pp++ ^= *pm++;
238         // Set the first byte to zero
239         padded->buffer[0] = 0x00;
240     Exit:
241         return retVal;
242     }
```

10.2.18.4.6 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 TPM_RC_VALUE.

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 TPM_RC_VALUE.

Error Returns	Meaning
TPM_RC_VALUE	the value to decode was larger than the modulus, or the padding is wrong or the buffer to receive the results is too small

```
243  static TPM_RC
244  OaepDecode(
245      TPM2B           *dataOut,       // OUT: the recovered data
246      TPM_ALG_ID       hashAlg,       // IN: algorithm to use for padding
247      const TPM2B     *label,         // IN: null-terminated string (may be NULL)
248      TPM2B           *padded         // IN: the padded data
249      )
250  {
251      UINT32       i;
252      BYTE         seedMask[MAX_DIGEST_SIZE];
253      UINT32       hLen = CryptHashGetDigestSize(hashAlg);
254      BYTE         mask[MAX_RSA_KEY_BYTES];
255      BYTE        *pp;
256      BYTE        *pm;
257      TPM_RC       retVal = TPM_RC_SUCCESS;
258      // Strange size (anything smaller can't be an OAEP padded block)
259      // Also check for no leading 0
260      if((padded->size < (unsigned)((2 * hLen) + 2)) || (padded->buffer[0] != 0))
261          ERROR_RETURN(TPM_RC_VALUE);
262  // Use the hash size to determine what to put through MGF1 in order
263  // to recover the seedMask
264      CryptMGF1(hLen, seedMask, hashAlg, padded->size - hLen - 1,
265                &padded->buffer[hLen + 1]);
266      // Recover the seed into seedMask
267      pAssert(hLen <= sizeof(seedMask));
268      pp = &padded->buffer[1];
269      pm = seedMask;
270      for(i = hLen; i > 0; i--)
271          *pm++ ^= *pp++;
272      // Use the seed to generate the data mask
273      CryptMGF1(padded->size - hLen - 1, mask, hashAlg, hLen, seedMask);
274      // Use the mask generated from seed to recover the padded data
275      pp = &padded->buffer[hLen + 1];
276      pm = mask;
277      for(i = (padded->size - hLen - 1); i > 0; i--)
278          *pm++ ^= *pp++;
279      // Make sure that the recovered data has the hash of the label
280      // Put trial value in the seed mask
281      if((CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
282                         hLen, seedMask)) < 0)
283          FAIL(FATAL_ERROR_INTERNAL);
284      if(memcmp(seedMask, mask, hLen) != 0)
285          ERROR_RETURN(TPM_RC_VALUE);
286      // find the start of the data
287      pm = &mask[hLen];
288      for(i = (UINT32)padded->size - (2 * hLen) - 1; i > 0; i--)
289      {
290          if(*pm++ != 0)
291              break;
292      }
293      // If we ran out of data or didn't end with 0x01, then return an error
294      if(i == 0 || pm[-1] != 0x01)
295          ERROR_RETURN(TPM_RC_VALUE);
296      // pm should be pointing at the first part of the data
297      // and i is one greater than the number of bytes to move
298      i--;
299      if(i > dataOut->size)
300          // Special exit to preserve the size of the output buffer
301          return TPM_RC_VALUE;
302      memcpy(dataOut->buffer, pm, i);
```



```
303         dataOut->size = (UINT16)i;
304     Exit:
305         if(retVal != TPM_RC_SUCCESS)
306             dataOut->size = 0;
307         return retVal;
308     }
```

10.2.18.4.7 PKSC1v1_5Encode()

This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

Error Returns	Meaning
TPM_RC_VALUE	message size is too large

```
309     static TPM_RC
310     RSAES_PKSC1v1_5Encode(
311         TPM2B       *padded,        // OUT: the pad data
312         TPM2B       *message,       // IN: the message being padded
313         RAND_STATE  *rand
314         )
315     {
316         UINT32      ps = padded->size - message->size - 3;
317     //
318         if(message->size > padded->size - 11)
319             return TPM_RC_VALUE;
320         // move the message to the end of the buffer
321         memcpy(&padded->buffer[padded->size - message->size], message->buffer,
322                message->size);
323         // Set the first byte to 0x00 and the second to 0x02
324         padded->buffer[0] = 0;
325         padded->buffer[1] = 2;
326         // Fill with random bytes
327         DRBG_Generate(rand, &padded->buffer[2], (UINT16)ps);
328         // Set the delimiter for the random field to 0
329         padded->buffer[2 + ps] = 0;
330         // Now, the only messy part. Make sure that all the 'ps' bytes are non-zero
331         // In this implementation, use the value of the current index
332         for(ps++; ps > 1; ps--)
333         {
334             if(padded->buffer[ps] == 0)
335                 padded->buffer[ps] = 0x55;  // In the < 0.5% of the cases that the
336                                             // random value is 0, just pick a value to
337                                             // put into the spot.
338         }
339         return TPM_RC_SUCCESS;
340     }
```

10.2.18.4.8 RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

Error Returns	Meaning
TPM_RC_FAIL	decoding error or results would no fit into provided buffer

```
341     static TPM_RC
342     RSAES_Decode(
343         TPM2B       *message,       // OUT: the recovered message
344         TPM2B       *coded          // IN: the encoded message
345         )
346     {
```

```
347         BOOL        fail = FALSE;
348         UINT16      pSize;
349         fail = (coded->size < 11);
350         fail = (coded->buffer[0] != 0x00) | fail;
351         fail = (coded->buffer[1] != 0x02) | fail;
352         for(pSize = 2; pSize < coded->size; pSize++)
353         {
354             if(coded->buffer[pSize] == 0)
355                 break;
356         }
357         pSize++;
358         // Make sure that pSize has not gone over the end and that there are at least 8
359         // bytes of pad data.
360         fail = (pSize >= coded->size) | fail;
361         fail = ((pSize - 2) < 8) | fail;
362         if((message->size < (UINT16)(coded->size - pSize)) || fail)
363             return TPM_RC_VALUE;
364         message->size = coded->size - pSize;
365         memcpy(message->buffer, &coded->buffer[pSize], coded->size - pSize);
366         return TPM_RC_SUCCESS;
367     }
```

10.2.18.4.9 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.

Returns TPM_RC_SUCCESS or goes into failure mode.

```
368     static TPM_RC
369     PssEncode(
370         TPM2B           *out,       // OUT: the encoded buffer
371         TPM_ALG_ID       hashAlg,   // IN: hash algorithm for the encoding
372         TPM2B           *digest,    // IN: the digest
373         RAND_STATE      *rand       // IN: random number source
374         )
375     {
376         UINT32              hLen = CryptHashGetDigestSize(hashAlg);
377         BYTE                salt[MAX_RSA_KEY_BYTES - 1];
378         UINT16              saltSize;
379         BYTE               *ps = salt;
380         BYTE               *pOut;
381         UINT16              mLen;
382         HASH_STATE          hashState;
383         // These are fatal errors indicating bad TPM firmware
384         pAssert(out != NULL && hLen > 0 && digest != NULL);
385         // Get the size of the mask
386         mLen = (UINT16)(out->size - hLen - 1);
387         // Maximum possible salt size is mask length - 1
388         saltSize = mLen - 1;
389         // Use the maximum salt size allowed by FIPS 186-4
390         if(saltSize > hLen)
391             saltSize = (UINT16)hLen;
392     //using eOut for scratch space
393         // Set the first 8 bytes to zero
394         pOut = out->buffer;
395         memset(pOut, 0, 8);
396         // Get set the salt
397         DRBG_Generate(rand, salt, saltSize);
398         // Create the hash of the pad || input hash || salt
399         CryptHashStart(&hashState, hashAlg);
400         CryptDigestUpdate(&hashState, 8, pOut);
401         CryptDigestUpdate2B(&hashState, digest);
402         CryptDigestUpdate(&hashState, saltSize, salt);
```

```
403         CryptHashEnd(&hashState, hLen, &pOut[out->size - hLen - 1]);
404         // Create a mask
405         if(CryptMGF1(mLen, pOut, hashAlg, hLen, &pOut[mLen]) != mLen)
406             FAIL(FATAL_ERROR_INTERNAL);
407         // Since this implementation uses key sizes that are all even multiples of
408         // 8, just need to make sure that the most significant bit is CLEAR
409         *pOut &= 0x7f;
410         // Before we mess up the pOut value, set the last byte to 0xbc
411         pOut[out->size - 1] = 0xbc;
412         // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
413         pOut = &pOut[mLen - saltSize - 1];
414         *pOut++ ^= 0x01;
415         // XOR the salt data into the buffer
416         for(; saltSize > 0; saltSize--)
417             *pOut++ ^= *ps++;
418         // and we are done
419         return TPM_RC_SUCCESS;
420     }
```

10.2.18.4.10 PssDecode()

This function checks that the PSS encoded block was built from the provided digest. If the check is successful, TPM_RC_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.

Error Returns	Meaning
TPM_RC_SCHEME	*hashAlg* is not a supported hash algorithm
TPM_RC_VALUE	decode operation failed

```
421     static TPM_RC
422     PssDecode(
423         TPM_ALG_ID   hashAlg,        // IN: hash algorithm to use for the encoding
424         TPM2B       *dIn,            // In: the digest to compare
425         TPM2B       *eIn             // IN: the encoded data
426         )
427     {
428         UINT32           hLen = CryptHashGetDigestSize(hashAlg);
429         BYTE             mask[MAX_RSA_KEY_BYTES];
430         BYTE            *pm = mask;
431         BYTE            *pe;
432         BYTE             pad[8] = {0};
433         UINT32           i;
434         UINT32           mLen;
435         BYTE             fail;
436         TPM_RC           retVal = TPM_RC_SUCCESS;
437         HASH_STATE       hashState;
438         // These errors are indicative of failures due to programmer error
439         pAssert(dIn != NULL && eIn != NULL);
440         pe = eIn->buffer;
441         // check the hash scheme
442         if(hLen == 0)
443             ERROR_RETURN(TPM_RC_SCHEME);
444         // most significant bit must be zero
445         fail = pe[0] & 0x80;
446         // last byte must be 0xbc
447         fail |= pe[eIn->size - 1] ^ 0xbc;
448         // Use the hLen bytes at the end of the buffer to generate a mask
449         // Doesn't start at the end which is a flag byte
```

```
450         mLen = eIn->size - hLen - 1;
451         CryptMGF1(mLen, mask, hashAlg, hLen, &pe[mLen]);
452         // Clear the MSO of the mask to make it consistent with the encoding.
453         mask[0] &= 0x7F;
454         pAssert(mLen <= sizeof(mask));
455         // XOR the data into the mask to recover the salt. This sequence
456         // advances eIn so that it will end up pointing to the seed data
457         // which is the hash of the signature data
458         for(i = mLen; i > 0; i--)
459             *pm++ ^= *pe++;
460         // Find the first byte of 0x01 after a string of all 0x00
461         for(pm = mask, i = mLen; i > 0; i--)
462         {
463             if(*pm == 0x01)
464                 break;
465             else
466                 fail |= *pm++;
467         }
468         // i should not be zero
469         fail |= (i == 0);
470         // if we have failed, will continue using the entire mask as the salt value so
471         // that the timing attacks will not disclose anything (I don't think that this
472         // is a problem for TPM applications but, usually, we don't fail so this
473         // doesn't cost anything).
474         if(fail)
475         {
476             i = mLen;
477             pm = mask;
478         }
479         else
480         {
481             pm++;
482             i--;
483         }
484         // i contains the salt size and pm points to the salt. Going to use the input
485         // hash and the seed to recreate the hash in the lower portion of eIn.
486         CryptHashStart(&hashState, hashAlg);
487         // add the pad of 8 zeros
488         CryptDigestUpdate(&hashState, 8, pad);
489         // add the provided digest value
490         CryptDigestUpdate(&hashState, dIn->size, dIn->buffer);
491         // and the salt
492         CryptDigestUpdate(&hashState, i, pm);
493         // get the result
494         fail |= (CryptHashEnd(&hashState, hLen, mask) != hLen);
495         // Compare all bytes
496         for(pm = mask; hLen > 0; hLen--)
497             // don't use fail = because that could skip the increment and compare
498             // operations after the first failure and that gives away timing
499             // information.
500             fail |= *pm++ ^ *pe++;
501         retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
502     Exit:
503             return retVal;
504     }
```

10.2.18.4.11 RSASSA_Encode()

Encode a message using RSASSA method.

Error Returns	Meaning
TPM_RC_SCHEME	*hashAlg* is not a supported hash algorithm
TPM_RC_SIZE	*eOutSize* is not large enough
TPM_RC_VALUE	*hInSize* does not match the digest size of *hashAlg*

```
505   static TPM_RC
506   RSASSA_Encode(
507       TPM2B               *pOut,      // IN:OUT on in, the size of the public key
508                                       //        on out, the encoded area
509       TPM_ALG_ID           hashAlg,   // IN: hash algorithm for PKSC1v1_5
510       TPM2B               *hIn        // IN: digest value to encode
511       )
512   {
513       const BYTE      *der;
514       BYTE            *eOut;
515       INT32            derSize = CryptHashGetDer(hashAlg, &der);
516       INT32            fillSize;
517       TPM_RC           retVal = TPM_RC_SUCCESS;
518       // Can't use this scheme if the algorithm doesn't have a DER string defined.
519       if(derSize == 0)
520           ERROR_RETURN(TPM_RC_SCHEME);
521       // If the digest size of 'hashAl' doesn't match the input digest size, then
522       // the DER will misidentify the digest so return an error
523       if(CryptHashGetDigestSize(hashAlg) != hIn->size)
524           ERROR_RETURN(TPM_RC_VALUE);
525       fillSize = pOut->size - derSize - hIn->size - 3;
526       eOut = pOut->buffer;
527       // Make sure that this combination will fit in the provided space
528       if(fillSize < 8)
529           ERROR_RETURN(TPM_RC_SIZE);
530       // Start filling
531       *eOut++ = 0; // initial byte of zero
532       *eOut++ = 1; // byte of 0x01
533       for(; fillSize > 0; fillSize--)
534           *eOut++ = 0xff; // bunch of 0xff
535       *eOut++ = 0; // another 0
536       for(; derSize > 0; derSize--)
537           *eOut++ = *der++;   // copy the DER
538       der = hIn->buffer;
539       for(fillSize = hIn->size; fillSize > 0; fillSize--)
540           *eOut++ = *der++;   // copy the hash
541   Exit:
542       return retVal;
543   }
```

10.2.18.4.12 RSASSA_Decode()

This function performs the RSASSA decoding of a signature.

Error Returns	Meaning
TPM_RC_VALUE	decode unsuccessful
TPM_RC_SCHEME	*haslAlg* is not supported

```
544   static TPM_RC
545   RSASSA_Decode(
546       TPM_ALG_ID       hashAlg,        // IN: hash algorithm to use for the encoding
547       TPM2B           *hIn,            // In: the digest to compare
548       TPM2B           *eIn             // IN: the encoded data
549       )
```

```
{
    BYTE             fail;
    const BYTE      *der;
    BYTE            *pe;
    INT32            derSize = CryptHashGetDer(hashAlg, &der);
    INT32            hashSize = CryptHashGetDigestSize(hashAlg);
    INT32            fillSize;
    TPM_RC           retVal;
    BYTE            *digest;
    UINT16           digestSize;
    pAssert(hIn != NULL && eIn != NULL);
    pe = eIn->buffer;
    // Can't use this scheme if the algorithm doesn't have a DER string
    // defined or if the provided hash isn't the right size
    if(derSize == 0 || (unsigned)hashSize != hIn->size)
        ERROR_RETURN(TPM_RC_SCHEME);
    // Make sure that this combination will fit in the provided space
    // Since no data movement takes place, can just walk though this
    // and accept nearly random values. This can only be called from
    // CryptValidateSignature() so eInSize is known to be in range.
    fillSize = eIn->size - derSize - hashSize - 3;
    // Start checking (fail will become non-zero if any of the bytes do not have
    // the expected value.
    fail = *pe++;                   // initial byte of zero
    fail |= *pe++ ^ 1;              // byte of 0x01
    for(; fillSize > 0; fillSize--)
        fail |= *pe++ ^ 0xff;       // bunch of 0xff
    fail |= *pe++;                  // another 0
    for(; derSize > 0; derSize--)
        fail |= *pe++ ^ *der++;    // match the DER
    digestSize = hIn->size;
    digest = hIn->buffer;
    for(; digestSize > 0; digestSize--)
        fail |= *pe++ ^ *digest++; // match the hash
    retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
Exit:
    return retVal;
}
```

10.2.18.4.13 CryptRsaSelectScheme()

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.

```
TPMT_RSA_DECRYPT*
CryptRsaSelectScheme(
    TPMI_DH_OBJECT       rsaHandle,     // IN: handle of an RSA key
    TPMT_RSA_DECRYPT    *scheme         // IN: a sign or decrypt scheme
    )
{
    OBJECT              *rsaObject;
    TPMT_ASYM_SCHEME    *keyScheme;
    TPMT_RSA_DECRYPT    *retVal = NULL;
    // Get sign object pointer
    rsaObject = HandleToObject(rsaHandle);
    keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
    // if the default scheme of the object is TPM_ALG_NULL, then select the
    // input scheme
```

```
602         if(keyScheme->scheme == TPM_ALG_NULL)
603         {
604             retVal = scheme;
605         }
606         // if the object scheme is not TPM_ALG_NULL and the input scheme is
607         // TPM_ALG_NULL, then select the default scheme of the object.
608         else if(scheme->scheme == TPM_ALG_NULL)
609         {
610             // if input scheme is NULL
611             retVal = (TPMT_RSA_DECRYPT *)keyScheme;
612         }
613         // get here if both the object scheme and the input scheme are
614         // not TPM_ALG_NULL. Need to insure that they are the same.
615         // IMPLEMENTATION NOTE: This could cause problems if future versions have
616         // schemes that have more values than just a hash algorithm. A new function
617         // (IsSchemeSame()) might be needed then.
618         else if(keyScheme->scheme == scheme->scheme
619                 && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
620         {
621             retVal = scheme;
622         }
623         // two different, incompatible schemes specified will return NULL
624         return retVal;
625     }
```

10.2.18.4.14 CryptRsaLoadPrivateExponent()

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of *rsaKey* are not matched

```
626     TPM_RC
627     CryptRsaLoadPrivateExponent(
628         OBJECT          *rsaKey         // IN: the RSA key object
629         )
630     {
631         BN_RSA_INITIALIZED(bnN, &rsaKey->publicArea.unique.rsa);
632         BN_PRIME_INITIALIZED(bnP, &rsaKey->sensitive.sensitive.rsa);
633         BN_RSA(bnQ);
634         BN_PRIME(bnQr);
635         BN_WORD_INITIALIZED(bnE, (rsaKey->publicArea.parameters.rsaDetail.exponent == 0)
636                             ? RSA_DEFAULT_PUBLIC_EXPONENT
637                             : rsaKey->publicArea.parameters.rsaDetail.exponent);
638         TPM_RC          retVal = TPM_RC_SUCCESS;
639         if(!rsaKey->attributes.privateExp)
640         {
641             TEST(ALG_NULL_VALUE);
642             // Make sure that the bigNum used for the exponent is properly initialized
643             RsaInitializeExponent(&rsaKey->privateExponent);
644     //??         // Make sure that the prime is not plain wrong
645     //??         if(BnEqualZero(bnP))
646     //??             ERROR_RETURN(TPM_RC_BINDING);
647             // Find the second prime by division
648             BnDiv(bnQ, bnQr, bnN, bnP);
649             if(!BnEqualZero(bnQr))
650                 ERROR_RETURN(TPM_RC_BINDING);
651             // Compute the private exponent and return it if found
652             if(!ComputePrivateExponent(bnP, bnQ, bnE, bnN,
653                                        &rsaKey->privateExponent))
654                 ERROR_RETURN(TPM_RC_BINDING);
655         }
656     Exit:
657         rsaKey->attributes.privateExp = (retVal == TPM_RC_SUCCESS);
658         return retVal;
```

```
659     }
```

10.2.18.4.15 CryptRsaEncrypt()

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 treated 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.

Error Returns	Meaning
TPM_RC_VALUE	*cOutSize* is too small (must be the size of the modulus)
TPM_RC_SCHEME	*padType* is not a supported scheme

```
660     LIB_EXPORT TPM_RC
661     CryptRsaEncrypt(
662         TPM2B_PUBLIC_KEY_RSA        *cOut,          // OUT: the encrypted data
663         TPM2B                       *dIn,           // IN: the data to encrypt
664         OBJECT                      *key,           // IN: the key used for encryption
665         TPMT_RSA_DECRYPT            *scheme,        // IN: the type of padding and hash
666                                                     //     if needed
667         const TPM2B                 *label,         // IN: in case it is needed
668         RAND_STATE                  *rand           // IN: random number generator
669                                                     //     state (mostly for testing)
670         )
671     {
672         TPM_RC                       retVal = TPM_RC_SUCCESS;
673         TPM2B_PUBLIC_KEY_RSA         dataIn;
674     //
675         // if the input and output buffers are the same, copy the input to a scratch
676         // buffer so that things don't get messed up.
677         if(dIn == &cOut->b)
678         {
679             MemoryCopy2B(&dataIn.b, dIn, sizeof(dataIn.t.buffer));
680             dIn = &dataIn.b;
681         }
682         // All encryption schemes return the same size of data
683         cOut->t.size = key->publicArea.unique.rsa.t.size;
684         TEST(scheme->scheme);
685         switch(scheme->scheme)
686         {
687             case TPM_ALG_NULL:  // 'raw' encryption
688             {
689                 INT32           i;
690                 INT32           dSize = dIn->size;
691                 // dIn can have more bytes than cOut as long as the extra bytes
692                 // are zero. Note: the more significant bytes of a number in a byte
693                 // buffer are the bytes at the start of the array.
694                 for(i = 0; (i < dSize) && (dIn->buffer[i] == 0); i++);
695                 dSize -= i;
696                 if(dSize > cOut->t.size)
697                     ERROR_RETURN(TPM_RC_VALUE);
698                 // Pad cOut with zeros if dIn is smaller
699                 memset(cOut->t.buffer, 0, cOut->t.size - dSize);
700                 // And copy the rest of the value
```

```
701             memcpy(&cOut->t.buffer[cOut->t.size - dSize], &dIn->buffer[i], dSize);
702             // If the size of dIn is the same as cOut dIn could be larger than
703             // the modulus. If it is, then RSAEP() will catch it.
704         }
705         break;
706         case TPM_ALG_RSAES:
707             retVal = RSAES_PKSC1v1_5Encode(&cOut->b, dIn, rand);
708             break;
709         case TPM_ALG_OAEP:
710             retVal = OaepEncode(&cOut->b, scheme->details.oaep.hashAlg, label, dIn,
711                                 rand);
712             break;
713         default:
714             ERROR_RETURN(TPM_RC_SCHEME);
715             break;
716     }
717     // All the schemes that do padding will come here for the encryption step
718     // Check that the Encoding worked
719     if(retVal == TPM_RC_SUCCESS)
720         // Padding OK so do the encryption
721         retVal = RSAEP(&cOut->b, key);
722 Exit:
723     return retVal;
724 }
```

10.2.18.4.16 CryptRsaDecrypt()

This is the entry point for decryption using RSA. Decryption is use of the private exponent. The **padType** parameter determines what padding was used.

Error Returns	Meaning
TPM_RC_SIZE	*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
TPM_RC_VALUE	*dOutSize* is not large enough for the result
TPM_RC_SCHEME	*padType* is not supported

```
725 LIB_EXPORT TPM_RC
726 CryptRsaDecrypt(
727     TPM2B               *dOut,          // OUT: the decrypted data
728     TPM2B               *cIn,           // IN: the data to decrypt
729     OBJECT              *key,           // IN: the key to use for decryption
730     TPMT_RSA_DECRYPT    *scheme,        // IN: the padding scheme
731     const TPM2B         *label          // IN: in case it is needed for the scheme
732     )
733 {
734     TPM_RC                  retVal;
735     // Make sure that the necessary parameters are provided
736     pAssert(cIn != NULL && dOut != NULL && key != NULL);
737     // Size is checked to make sure that the encrypted value is the right size
738     if(cIn->size != key->publicArea.unique.rsa.t.size)
739         ERROR_RETURN(TPM_RC_SIZE);
740     TEST(scheme->scheme);
741     // For others that do padding, do the decryption in place and then
742     // go handle the decoding.
743     retVal = RSADP(cIn, key);
744     if(retVal == TPM_RC_SUCCESS)
745     {
746         // Remove padding
747         switch(scheme->scheme)
748         {
749             case TPM_ALG_NULL:
750                 if(dOut->size < cIn->size)
```

```
751                     return TPM_RC_VALUE;
752                 MemoryCopy2B(dOut, cIn, dOut->size);
753                 break;
754             case TPM_ALG_RSAES:
755                 retVal = RSAES_Decode(dOut, cIn);
756                 break;
757             case TPM_ALG_OAEP:
758                 retVal = OaepDecode(dOut, scheme->details.oaep.hashAlg, label, cIn);
759                 break;
760             default:
761                 retVal = TPM_RC_SCHEME;
762                 break;
763         }
764     }
765 Exit:
766     return retVal;
767 }
```

10.2.18.4.17 CryptRsaSign()

This function is used to generate an RSA signature of the type indicated in *scheme*.

Error Returns	Meaning
TPM_RC_SCHEME	*scheme* or *hashAlg* are not supported
TPM_RC_VALUE	*hInSize* does not match *hashAlg* (for RSASSA)

```
768 LIB_EXPORT TPM_RC
769 CryptRsaSign(
770     TPMT_SIGNATURE      *sigOut,
771     OBJECT              *key,           // IN: key to use
772     TPM2B_DIGEST        *hIn,           // IN: the digest to sign
773     RAND_STATE          *rand           // IN: the random number generator
774                                         //      to use (mostly for testing)
775     )
776 {
777     TPM_RC                retVal = TPM_RC_SUCCESS;
778     UINT16                modSize;
779     // parameter checks
780     pAssert(sigOut != NULL && key != NULL && hIn != NULL);
781     modSize = key->publicArea.unique.rsa.t.size;
782     // for all non-null signatures, the size is the size of the key modulus
783     sigOut->signature.rsapss.sig.t.size = modSize;
784     TEST(sigOut->sigAlg);
785     switch(sigOut->sigAlg)
786     {
787         case TPM_ALG_NULL:
788             sigOut->signature.rsapss.sig.t.size = 0;
789             return TPM_RC_SUCCESS;
790         case TPM_ALG_RSAPSS:
791             retVal = PssEncode(&sigOut->signature.rsapss.sig.b,
792                                sigOut->signature.rsapss.hash, &hIn->b, rand);
793             break;
794         case TPM_ALG_RSASSA:
795             retVal = RSASSA_Encode(&sigOut->signature.rsassa.sig.b,
796                                    sigOut->signature.rsassa.hash, &hIn->b);
797             break;
798         default:
799             retVal = TPM_RC_SCHEME;
800     }
801     if(retVal == TPM_RC_SUCCESS)
802     {
803         // Do the encryption using the private key
```



```
804             retVal = RSADP(&sigOut->signature.rsapss.sig.b, key);
805         }
806         return retVal;
807     }
```

10.2.18.4.18 CryptRsaValidateSignature()

This function is used to validate an RSA signature. If the signature is valid TPM_RC_SUCCESS is returned. If the signature is not valid, TPM_RC_SIGNATURE is returned. Other return codes indicate either parameter problems or fatal errors.

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature does not check
TPM_RC_SCHEME	unsupported scheme or hash algorithm

```
808 LIB_EXPORT TPM_RC
809 CryptRsaValidateSignature(
810     TPMT_SIGNATURE  *sig,           // IN: signature
811     OBJECT          *key,           // IN: public modulus
812     TPM2B_DIGEST    *digest         // IN: The digest being validated
813     )
814 {
815     TPM_RC          retVal;
816 //
817     // Fatal programming errors
818     pAssert(key != NULL && sig != NULL && digest != NULL);
819     switch(sig->sigAlg)
820     {
821         case TPM_ALG_RSAPSS:
822         case TPM_ALG_RSASSA:
823             break;
824         default:
825             return TPM_RC_SCHEME;
826     }
827     // Errors that might be caused by calling parameters
828     if(sig->signature.rsassa.sig.t.size != key->publicArea.unique.rsa.t.size)
829         ERROR_RETURN(TPM_RC_SIGNATURE);
830     TEST(sig->sigAlg);
831     // Decrypt the block
832     retVal = RSAEP(&sig->signature.rsassa.sig.b, key);
833     if(retVal == TPM_RC_SUCCESS)
834     {
835         switch(sig->sigAlg)
836         {
837             case TPM_ALG_RSAPSS:
838                 retVal = PssDecode(sig->signature.any.hashAlg, &digest->b,
839                                    &sig->signature.rsassa.sig.b);
840                 break;
841             case TPM_ALG_RSASSA:
842                 retVal = RSASSA_Decode(sig->signature.any.hashAlg, &digest->b,
843                                        &sig->signature.rsassa.sig.b);
844                 break;
845             default:
846                 return TPM_RC_SCHEME;
847         }
848     }
849 Exit:
850     return (retVal != TPM_RC_SUCCESS) ? TPM_RC_SIGNATURE : TPM_RC_SUCCESS;
851 }
852 #if defined SIMULATION && defined USE_RSA_KEY_CACHE
853 extern int s_rsaKeyCacheEnabled;
854 int GetCachedRsaKey(OBJECT *key, RAND_STATE *rand);
```



```
#define GET_CACHED_KEY(key, rand)                       \
            (s_rsaKeyCacheEnabled && GetCachedRsaKey(key, rand))
#else
#define GET_CACHED_KEY(key, rand)
#endif
```

10.2.18.4.19 CryptRsaGenerateKey()

Generate an RSA key from a provided seed

Error Returns	Meaning
TPM_RC_CANCELED	operation was canceled
TPM_RC_RANGE	public exponent is not supported
TPM_RC_VALUE	could not find a prime using the provided parameters

```
LIB_EXPORT TPM_RC
CryptRsaGenerateKey(
    OBJECT              *rsaKey,            // IN/OUT: The object structure in which
                                            //          the key is created.
    RAND_STATE          *rand               // IN: if not NULL, the deterministic
                                            //     RNG state
    )
{
    UINT32               i;
    BN_PRIME(bnP); // These four declarations initialize the number to 0
    BN_PRIME(bnQ);
    BN_RSA(bnD);
    BN_RSA(bnN);
    BN_WORD(bnE);
    UINT32               e;
    int                  keySizeInBits;
    TPMT_PUBLIC         *publicArea = &rsaKey->publicArea;
    TPMT_SENSITIVE      *sensitive = &rsaKey->sensitive;
    TPM_RC               retVal = TPM_RC_NO_RESULT;
//
// Need to make sure that the caller did not specify an exponent that is
// not supported
    e = publicArea->parameters.rsaDetail.exponent;
    if(e == 0)
        e = RSA_DEFAULT_PUBLIC_EXPONENT;
    if(e < 65537)
        ERROR_RETURN(TPM_RC_RANGE);
//    if(e == 197499) //???
//        e = e;      //???
    if(e != RSA_DEFAULT_PUBLIC_EXPONENT && !IsPrimeInt(e))
        ERROR_RETURN(TPM_RC_RANGE);
    BnSetWord(bnE, e);
    // Check that e is prime
    // check for supported key size.
    keySizeInBits = publicArea->parameters.rsaDetail.keyBits;
    if(((keySizeInBits % 1024) != 0)
       || (keySizeInBits > MAX_RSA_KEY_BITS)  // this might be redundant, but...
       || (keySizeInBits == 0))
        ERROR_RETURN(TPM_RC_VALUE);
    // Set the prime size for instrumentation purposes
    INSTRUMENT_SET(PrimeIndex, PRIME_INDEX(keySizeInBits / 2));
#if defined SIMULATION && defined USE_RSA_KEY_CACHE
    if(GET_CACHED_KEY(rsaKey, rand))
        return TPM_RC_SUCCESS;
#endif
    // Make sure that key generation has been tested
    TEST(ALG_NULL_VALUE);
```



```
907         // Need to initialize the privateExponent structure
908         RsaInitializeExponent(&rsaKey->privateExponent);
909         // The prime is computed in P. When a new prime is found, Q is checked to
910         // see if it is zero.  If so, P is copied to Q and a new P is found.
911         // When both P and Q are non-zero, the modulus and
912         // private exponent are computed and a trial encryption/decryption is
913         // performed.  If the encrypt/decrypt fails, assume that at least one of the
914         // primes is composite. Since we don't know which one, set Q to zero and start
915         // over and find a new pair of primes.
916         for(i = 1; (retVal != TPM_RC_SUCCESS) && (i != 100); i++)
917         {
918             if(_plat__IsCanceled())
919                 ERROR_RETURN(TPM_RC_CANCELED);
920             BnGeneratePrimeForRSA(bnP, keySizeInBits / 2, e, rand);
921             INSTRUMENT_INC(PrimeCounts[PrimeIndex]);
922             // If this is the second prime, make sure that it differs from the
923             // first prime by at least 2^100
924             if(BnEqualZero(bnQ))
925             {
926                 // copy p to q and compute another prime in p
927                 BnCopy(bnQ, bnP);
928                 continue;
929             }
930             // Make sure that the difference is at least 100 bits. Need to do it this
931             // way because the big numbers are only positive values
932             if(BnUnsignedCmp(bnP, bnQ) < 0)
933                 BnSub(bnD, bnQ, bnP);
934             else
935                 BnSub(bnD, bnP, bnQ);
936             if(BnMsb(bnD) < 100)
937                 continue;
938             //Form the public modulus and set the unique value
939             BnMult(bnN, bnP, bnQ);
940             BnTo2B(bnN, &publicArea->unique.rsa.b,
941                    (NUMBYTES)BITS_TO_BYTES(keySizeInBits));
942             // And the  prime to the sensitive area
943             BnTo2B(bnP, &sensitive->sensitive.rsa.b,
944                    (NUMBYTES)BITS_TO_BYTES(keySizeInBits) / 2);
945             // Make sure everything came out right. The MSb of the values must be
946             // one
947             if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
948                || ((sensitive->sensitive.rsa.t.buffer[0] & 0x80) == 0))
949                 FAIL(FATAL_ERROR_INTERNAL);
950             // Make sure that we can form the private exponent values
951             if(ComputePrivateExponent(bnP, bnQ, bnE, bnN, &rsaKey->privateExponent)
952                != TRUE)
953             {
954                 // If ComputePrivateExponent could not find an inverse for
955                 // Q, then copy P and recompute P. This might
956                 // cause both to be recomputed if P is also zero
957                 if(BnEqualZero(bnQ))
958                     BnCopy(bnQ, bnP);
959                 continue;
960             }
961             retVal = TPM_RC_SUCCESS;
962             // Do a trial encryption decryption if this is a signing key
963             if(publicArea->objectAttributes.sign)
964             {
965                 BN_RSA(temp1);
966                 BN_RSA(temp2);
967                 BnGenerateRandomInRange(temp1, bnN, rand);
968                 // Encrypt with public exponent...
969                 BnModExp(temp2, temp1, bnE, bnN);
970                 // ...  then decrypt with private exponent
971                 RsaPrivateKeyOp(temp2, bnN, bnP, &rsaKey->privateExponent);
972                 // If the starting and ending values are not the same,
```

```
973             // start over )-;
974             if(BnUnsignedCmp(temp2, temp1) != 0)
975             {
976                 BnSetWord(bnQ, 0);
977                 retVal = TPM_RC_NO_RESULT;
978             }
979         }
980     }
981 Exit:
982     if(retVal == TPM_RC_SUCCESS)
983         rsaKey->attributes.privateExp = SET;
984     return retVal;
985 }
986 #endif // TPM_ALG_RSA
```

10.2.19 CryptSym.c

10.2.19.1 Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These functions only use the single block encryption functions of the selected symmetric crypto library.

10.2.19.2 Includes, Defines, and Typedefs

```
1   #include "Tpm.h"
2   union tpmCryptKeySchedule_t {
3   #ifdef TPM_ALG_AES
4       tpmKeyScheduleAES           AES;
5   #endif
6   #ifdef TPM_ALG_SM4
7       tpmKeyScheduleSM4           SM4;
8   #endif
9   #ifdef TPM_ALG_CAMELLIA
10      tpmKeyScheduleCAMELLIA      CAMELLIA;
11  #endif
12  #ifdef TPM_ALG_TDES
13      tpmKeyScheduleTDES          TDES[3];
14  #endif
15  #if SYMMETRIC_ALIGNMENT == 8
16      uint64_t            alignment;
17  #else
18      uint32_t            alignment;
19  #endif
20  };
```

Each block cipher within a library is expected to conform to the same calling conventions with three parameters *(keySchedule*, *in*, and *out*) in the same order. That means that all algorithms would use the same order of the same parameters. The code is written assuming the *(keySchedule*, *in*, and *out*) order. However, if the library uses a different order, the order can be changed with a SWIZZLE macro that puts the parameters in the correct order. Note that all algorithms have to use the same order and number of parameters because the code to build the calling list is common for each call to encrypt or decrypt with the algorithm chosen by setting a function pointer to select the algorithm that is used.

```
21  #   define ENCRYPT(keySchedule, in, out)               \
22        encrypt(SWIZZLE(keySchedule, in, out))
23  #   define DECRYPT(keySchedule, in, out)               \
24        decrypt(SWIZZLE(keySchedule, in, out))
```

Note that the macros rely on *encrypt* as local values in the functions that use these macros. Those parameters are set by the macro that set the key schedule to be used for the call.

```
25  #define ENCRYPT_CASE(ALG)                                                   \
26      case TPM_ALG_##ALG:                                                     \
27          TpmCryptSetEncryptKey##ALG(key, keySizeInBits, &keySchedule.ALG);   \
28          encrypt = (TpmCryptSetSymKeyCall_t)TpmCryptEncrypt##ALG;            \
29          break;
30  #define DECRYPT_CASE(ALG)                                                   \
31      case TPM_ALG_##ALG:                                                     \
32          TpmCryptSetDecryptKey##ALG(key, keySizeInBits, &keySchedule.ALG);   \
33          decrypt = (TpmCryptSetSymKeyCall_t)TpmCryptDecrypt##ALG;            \
34          break;
35  #ifdef TPM_ALG_AES
36  #define ENCRYPT_CASE_AES    ENCRYPT_CASE(AES)
37  #define DECRYPT_CASE_AES    DECRYPT_CASE(AES)
38  #else
39  #define ENCRYPT_CASE_AES
40  #define DECRYPT_CASE_AES
41  #endif
42  #ifdef TPM_ALG_SM4
43  #define ENCRYPT_CASE_SM4    ENCRYPT_CASE(SM4)
44  #define DECRYPT_CASE_SM4    DECRYPT_CASE(SM4)
45  #else
46  #define ENCRYPT_CASE_SM4
47  #define DECRYPT_CASE_SM4
48  #endif
49  #ifdef TPM_ALG_CAMELLIA
50  #define ENCRYPT_CASE_CAMELLIA    ENCRYPT_CASE(CAMELLIA)
51  #define DECRYPT_CASE_CAMELLIA    DECRYPT_CASE(CAMELLIA)
52  #else
53  #define ENCRYPT_CASE_CAMELLIA
54  #define DECRYPT_CASE_CAMELLIA
55  #endif
56  #ifdef TPM_ALG_TDES
57  #define ENCRYPT_CASE_TDES    ENCRYPT_CASE(TDES)
58  #define DECRYPT_CASE_TDES    DECRYPT_CASE(TDES)
59  #else
60  #define ENCRYPT_CASE_TDES
61  #define DECRYPT_CASE_TDES
62  #endif
```

For each algorithm the case will either be defined or null.

```
63  #define     SELECT(direction)                           \
64      switch(algorithm)                                   \
65      {                                                   \
66          direction##_CASE_AES                            \
67          direction##_CASE_SM4                            \
68          direction##_CASE_CAMELLIA                       \
69          direction##_CASE_TDES                           \
70          default:                                        \
71              return TPM_RC_FAILURE;                      \
72      }
```

10.2.19.3 Obligatory Initialization Functions

10.2.19.3.1 CryptSymInit()

This function is called to do _TPM_Init() processing

```
73  BOOL
74  CryptSymInit(
75      void
76      )
77  {
```

```
78      return TRUE;
79  }
```

10.2.19.3.2 CryptSymStartup()

This function is called to do TPM2_Startup() processing

```
80  BOOL
81  CryptSymStartup(
82      void
83      )
84  {
85      return TRUE;
86  }
```

10.2.19.4 Data Access Functions

10.2.19.4.1 CryptGetSymmetricBlockSize()

This function returns the block size of the algorithm.

Return Value	Meaning
<= 0	cipher not supported
> 0	the cipher block size in bytes

```
87  LIB_EXPORT INT16
88  CryptGetSymmetricBlockSize(
89      TPM_ALG_ID      symmetricAlg,   // IN: the symmetric algorithm
90      UINT16          keySizeInBits   // IN: the key size
91      )
92  {
93      switch(symmetricAlg)
94      {
95  #ifdef TPM_ALG_AES
96          case TPM_ALG_AES:
97              switch(keySizeInBits)
98              {
99                  case 128:
100                     return AES_128_BLOCK_SIZE_BYTES;
101                 case 192:
102                     return AES_192_BLOCK_SIZE_BYTES;
103                 case 256:
104                     return AES_256_BLOCK_SIZE_BYTES;
105                 default:
106                     break;
107             }
108             break;
109 #endif
110 #ifdef TPM_ALG_SM4
111         case TPM_ALG_SM4:
112             switch(keySizeInBits)
113             {
114                 case 128:
115                     return SM4_128_BLOCK_SIZE_BYTES;
116                 default:
117                     break;
118             }
119 #endif
120 #ifdef TPM_ALG_CAMELLIA
121         case TPM_ALG_CAMELLIA:
```

```
122             switch(keySizeInBits)
123             {
124                 case 128:
125                     return CAMELLIA_128_BLOCK_SIZE_BYTES;
126                 case 192:
127                     return CAMELLIA_192_BLOCK_SIZE_BYTES;
128                 case 256:
129                     return CAMELLIA_256_BLOCK_SIZE_BYTES;
130                 default:
131                     break;
132             }
133 #endif
134 #ifdef TPM_ALG_TDES
135         case TPM_ALG_TDES:
136             switch(keySizeInBits)
137             {
138                 case 128:
139                     return TDES_128_BLOCK_SIZE_BYTES;
140                 case 192:
141                     return TDES_192_BLOCK_SIZE_BYTES;
142                 default:
143                     break;
144             }
145 #endif
146         default:
147             break;
148     }
149     return 0;
150 }
```

10.2.19.5 Symmetric Encryption

This function performs symmetric encryption based on the mode.

Error Returns	Meaning
TPM_RC_SUCCESS	if success
TPM_RC_SIZE	*dSize* is not a multiple of the block size for an algorithm that requires it
TPM_RC_FAILURE	Fatal error

```
151 LIB_EXPORT TPM_RC
152 CryptSymmetricEncrypt(
153     BYTE                *dOut,          // OUT:
154     TPM_ALG_ID           algorithm,     // IN: the symmetric algorithm
155     UINT16               keySizeInBits, // IN: key size in bits
156     const BYTE          *key,           // IN: key buffer. The size of this buffer
157                                         //     in bytes is (keySizeInBits + 7) / 8
158     TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption.
159     TPM_ALG_ID           mode,          // IN: Mode to use
160     INT32                dSize,         // IN: data size (may need to be a
161                                         //     multiple of the blockSize)
162     const BYTE          *dIn            // IN: data buffer
163     )
164 {
165     BYTE                *pIv;
166     int                  i;
167     BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
168     BYTE                *pT;
169     tpmCryptKeySchedule_t        keySchedule;
170     INT16                blockSize;
171     TpmCryptSetSymKeyCall_t      encrypt;
172     BYTE                *iv;
```

```
173         BYTE                defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
174     //
175         pAssert(dOut != NULL && key != NULL && dIn != NULL);
176         if(dSize == 0)
177             return TPM_RC_SUCCESS;
178         TEST(algorithm);
179         blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
180         if(blockSize == 0)
181             return TPM_RC_FAILURE;
182         // If the iv is provided, then it is expected to be block sized. In some cases,
183         // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
184         // with no knowledge of the actual block size. This function will set it.
185         if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
186         {
187             ivInOut->t.size = blockSize;
188             iv = ivInOut->t.buffer;
189         }
190         else
191             iv = defaultIv;
192         pIv = iv;
193         // Create encrypt key schedule and set the encryption function pointer.
194         SELECT(ENCRYPT);
195         switch(mode)
196         {
197     #ifdef TPM_ALG_CTR
198             case TPM_ALG_CTR:
199                 for(; dSize > 0; dSize -= blockSize)
200                 {
201                     // Encrypt the current value of the IV(counter)
202                     ENCRYPT(&keySchedule, iv, tmp);
203                     //increment the counter (counter is big-endian so start at end)
204                     for(i = blockSize - 1; i >= 0; i--)
205                         if((iv[i] += 1) != 0)
206                             break;
207                     // XOR the encrypted counter value with input and put into output
208                     pT = tmp;
209                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
210                         *dOut++ = *dIn++ ^ *pT++;
211                 }
212                 break;
213     #endif
214     #ifdef TPM_ALG_OFB
215             case TPM_ALG_OFB:
216                 // This is written so that dIn and dOut may be the same
217                 for(; dSize > 0; dSize -= blockSize)
218                 {
219                     // Encrypt the current value of the "IV"
220                     ENCRYPT(&keySchedule, iv, iv);
221                     // XOR the encrypted IV into dIn to create the cipher text (dOut)
222                     pIv = iv;
223                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
224                         *dOut++ = (*pIv++ ^ *dIn++);
225                 }
226                 break;
227     #endif
228     #ifdef TPM_ALG_CBC
229             case TPM_ALG_CBC:
230                 // For CBC the data size must be an even multiple of the
231                 // cipher block size
232                 if((dSize % blockSize) != 0)
233                     return TPM_RC_SIZE;
234                 // XOR the data block into the IV, encrypt the IV into the IV
235                 // and then copy the IV to the output
236                 for(; dSize > 0; dSize -= blockSize)
237                 {
238                     pIv = iv;
```

```
239                 for(i = blockSize; i > 0; i--)
240                     *pIv++ ^= *dIn++;
241                 ENCRYPT(&keySchedule, iv, iv);
242                 pIv = iv;
243                 for(i = blockSize; i > 0; i--)
244                     *dOut++ = *pIv++;
245             }
246             break;
247 #endif
248         // CFB is not optional
249         case TPM_ALG_CFB:
250             // Encrypt the IV into the IV, XOR in the data, and copy to output
251             for(; dSize > 0; dSize -= blockSize)
252             {
253                 // Encrypt the current value of the IV
254                 ENCRYPT(&keySchedule, iv, iv);
255                 pIv = iv;
256                 for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
257                     // XOR the data into the IV to create the cipher text
258                     // and put into the output
259                     *dOut++ = *pIv++ ^= *dIn++;
260             }
261             // If the inner loop (i loop) was smaller than blockSize, then dSize
262             // would have been smaller than blockSize and it is now negative. If
263             // it is negative, then it indicates how many bytes are needed to pad
264             // out the IV for the next round.
265             for(; dSize < 0; dSize++)
266                 *pIv++ = 0;
267             break;
268 #ifdef TPM_ALG_ECB
269         case TPM_ALG_ECB:
270             // For ECB the data size must be an even multiple of the
271             // cipher block size
272             if((dSize % blockSize) != 0)
273                 return TPM_RC_SIZE;
274             // Encrypt the input block to the output block
275             for(; dSize > 0; dSize -= blockSize)
276             {
277                 ENCRYPT(&keySchedule, dIn, dOut);
278                 dIn = &dIn[blockSize];
279                 dOut = &dOut[blockSize];
280             }
281             break;
282 #endif
283         default:
284             return TPM_RC_FAILURE;
285     }
286     return TPM_RC_SUCCESS;
287 }
```

10.2.19.5.1 CryptSymmetricDecrypt()

This function performs symmetric decryption based on the mode.

Error Returns	Meaning
TPM_RC_FAILURE	A fatal error
TPM_RC_SUCCESS	if success
TPM_RCS_SIZE	*dSize* is not a multiple of the block size for an algorithm that requires it

```
288 LIB_EXPORT TPM_RC
289 CryptSymmetricDecrypt(
```

```
290         BYTE                *dOut,          // OUT: decrypted data
291         TPM_ALG_ID           algorithm,     // IN: the symmetric algorithm
292         UINT16               keySizeInBits, // IN: key size in bits
293         const BYTE          *key,           // IN: key buffer. The size of this buffer
294                                             //     in bytes is (keySizeInBits + 7) / 8
295         TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption.
296         TPM_ALG_ID           mode,          // IN: Mode to use
297         INT32                dSize,         // IN: data size (may need to be a
298                                             //     multiple of the blockSize)
299         const BYTE          *dIn            // IN: data buffer
300         )
301     {
302         BYTE                *pIv;
303         int                  i;
304         BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
305         BYTE                *pT;
306         tpmCryptKeySchedule_t        keySchedule;
307         INT16                blockSize;
308         BYTE                *iv;
309         TpmCryptSetSymKeyCall_t        encrypt;
310         TpmCryptSetSymKeyCall_t        decrypt;
311         BYTE                 defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
312         // These are used but the compiler can't tell because they are initialized
313         // in case statements and it can't tell if they are always initialized
314         // when needed, so... Comment these out if the compiler can tell or doesn't
315         // care that these are initialized before use.
316         encrypt = NULL;
317         decrypt = NULL;
318         pAssert(dOut != NULL && key != NULL && dIn != NULL);
319         if(dSize == 0)
320             return TPM_RC_SUCCESS;
321         TEST(algorithm);
322         blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
323         // If the iv is provided, then it is expected to be block sized. In some cases,
324         // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
325         // with no knowledge of the actual block size. This function will set it.
326         if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
327         {
328             ivInOut->t.size = blockSize;
329             iv = ivInOut->t.buffer;
330         }
331         else
332             iv = defaultIv;
333         pIv = iv;
334         // Use the mode to select the key schedule to create. Encrypt always uses the
335         // encryption schedule. Depending on the mode, decryption might use either
336         // the decryption or encryption schedule.
337         switch(mode)
338         {
339     #if defined TPM_ALG_CBC || defined TPM_ALG_ECB
340             case ALG_CBC_VALUE: // decrypt = decrypt
341             case ALG_ECB_VALUE:
342                 // For ECB and CBC, the data size must be an even multiple of the
343                 // cipher block size
344                 if((dSize % blockSize) != 0)
345                     return TPM_RC_SIZE;
346                 SELECT(DECRYPT);
347                 break;
348     #endif
349             default:
350                 // For the remaining stream ciphers, use encryption to decrypt
351                 SELECT(ENCRYPT);
352                 break;
353         }
354         // Now do the mode-dependent decryption
355         switch(mode)
```

```
356         {
357     #ifdef TPM_ALG_CBC
358             case TPM_ALG_CBC:
359                 // Copy the input data to a temp buffer, decrypt the buffer into the
360                 // output, XOR in the IV, and copy the temp buffer to the IV and repeat.
361                 for(; dSize > 0; dSize -= blockSize)
362                 {
363                     pT = tmp;
364                     for(i = blockSize; i > 0; i--)
365                         *pT++ = *dIn++;
366                     DECRYPT(&keySchedule, tmp, dOut);
367                     pIv = iv;
368                     pT = tmp;
369                     for(i = blockSize; i > 0; i--)
370                     {
371                         *dOut++ ^= *pIv;
372                         *pIv++ = *pT++;
373                     }
374                 }
375                 break;
376     #endif
377             case TPM_ALG_CFB:
378                 for(; dSize > 0; dSize -= blockSize)
379                 {
380                     // Encrypt the IV into the temp buffer
381                     ENCRYPT(&keySchedule, iv, tmp);
382                     pT = tmp;
383                     pIv = iv;
384                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
385                         // Copy the current cipher text to IV, XOR
386                         // with the temp buffer and put into the output
387                         *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
388                 }
389                 // If the inner loop (i loop) was smaller than blockSize, then dSize
390                 // would have been smaller than blockSize and it is now negative
391                 // If it is negative, then it indicates how may fill bytes
392                 // are needed to pad out the IV for the next round.
393                 for(; dSize < 0; dSize++)
394                     *pIv++ = 0;
395                 break;
396     #ifdef TPM_ALG_CTR
397             case TPM_ALG_CTR:
398                 for(; dSize > 0; dSize -= blockSize)
399                 {
400                     // Encrypt the current value of the IV(counter)
401                     ENCRYPT(&keySchedule, iv, tmp);
402                     //increment the counter (counter is big-endian so start at end)
403                     for(i = blockSize - 1; i >= 0; i--)
404                         if((iv[i] += 1) != 0)
405                             break;
406                     // XOR the encrypted counter value with input and put into output
407                     pT = tmp;
408                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
409                         *dOut++ = *dIn++ ^ *pT++;
410                 }
411                 break;
412     #endif
413     #ifdef TPM_ALG_ECB
414             case TPM_ALG_ECB:
415                 for(; dSize > 0; dSize -= blockSize)
416                 {
417                     DECRYPT(&keySchedule, dIn, dOut);
418                     dIn = &dIn[blockSize];
419                     dOut = &dOut[blockSize];
420                 }
421                 break;
```

```
422     #endif
423     #ifdef TPM_ALG_OFB
424             case TPM_ALG_OFB:
425                 // This is written so that dIn and dOut may be the same
426                 for(; dSize > 0; dSize -= blockSize)
427                 {
428                     // Encrypt the current value of the "IV"
429                     ENCRYPT(&keySchedule, iv, iv);
430                     // XOR the encrypted IV into dIn to create the cipher text (dOut)
431                     pIv = iv;
432                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
433                         *dOut++ = (*pIv++ ^ *dIn++);
434                 }
435                 break;
436     #endif
437             default:
438                 return TPM_RC_FAILURE;
439         }
440         return TPM_RC_SUCCESS;
441     }
```

10.2.19.5.2 CryptSymKeyValidate()

Validate that a provided symmetric key meets the requirements of the TPM

Error Returns	Meaning
TPM_RC_KEY_SIZE	Key size specifiers do not match
TPM_RC_KEY	Key is not allowed

```
442     TPM_RC
443     CryptSymKeyValidate(
444         TPMT_SYM_DEF_OBJECT *symDef,
445         TPM2B_SYM_KEY       *key
446         )
447     {
448         if(key->t.size != BITS_TO_BYTES(symDef->keyBits.sym))
449             return TPM_RCS_KEY_SIZE;
450     #ifdef TPM_ALG_TDES
451         if(symDef->algorithm == TPM_ALG_TDES && !CryptDesValidateKey(key))
452             return TPM_RCS_KEY;
453     #endif // TPM_ALG_TDES
454         return TPM_RC_SUCCESS;
455     }
```

10.2.20 PrimeData.c

```
1     #include "Tpm.h"
```

This table is the product of all of the primes up to 1000. Checking to see if there is a GCD between the a prime candidate and this number will eliminate many prime candidates from consideration before running Miller-Rabin on the result.

```
2     const BN_STRUCT(43 * RADIX_BITS) s_CompositeOfSmallPrimes_ =
3     {44, 44,
4     { 0x2ED42696, 0x2BBFA177, 0x4820594F, 0xF73F4841,
5     0xBFAC313A, 0xCAC3EB81, 0xF6F26BF8, 0x7FAB5061,
6     0x59746FB7, 0xF71377F6, 0x3B19855B, 0xCBD03132,
7     0xBB92EF1B, 0x3AC3152C, 0xE87C8273, 0xC0AE0E69,
8     0x74A9E295, 0x448CCE86, 0x63CA1907, 0x8A0BF944,
9     0xF8CC3BE0, 0xC26F0AF5, 0xC501C02F, 0x6579441A,
```



```
10    0xD1099CDA, 0x6BC76A00, 0xC81A3228, 0xBFB1AB25,
11    0x70FA3841, 0x51B3D076, 0xCC2359ED, 0xD9EE0769,
12    0x75E47AF0, 0xD45FF31E, 0x52CCE4F6, 0x04DBC891,
13    0x96658ED2, 0x1753EFE5, 0x3AE4A5A6, 0x8FD4A97F,
14    0x8B15E7EB, 0x0243C3E1, 0xE0F0C31D, 0x0000000B }
15    };
16    bigConst        s_CompositeOfSmallPrimes = (const bigNum)&s_CompositeOfSmallPrimes_;
```

This table contains a bit for each of the odd values between 1 and 2^16 + 1. This table allows fast checking of the primes in that range. Don't change the size of this table unless you are prepared to do redo IsPrimeInt().

```
17    const uint32_t      s_LastPrimeInTable = 65537;
18    const uint32_t      s_PrimeTableSize = 4097;
19    const uint32_t      s_PrimesInTable = 6542;
20    const unsigned char s_PrimeTable[] = {
21        0x6e, 0xcb, 0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x81, 0x32, 0x4c, 0x4a, 0x86,
22        0x0d, 0x82, 0x96, 0x21, 0xc9, 0x34, 0x04, 0x5a, 0x20, 0x61, 0x89, 0xa4,
23        0x44, 0x11, 0x86, 0x29, 0xd1, 0x82, 0x28, 0x4a, 0x30, 0x40, 0x42, 0x32,
24        0x21, 0x99, 0x34, 0x08, 0x4b, 0x06, 0x25, 0x42, 0x84, 0x48, 0x8a, 0x14,
25        0x05, 0x42, 0x30, 0x6c, 0x08, 0xb4, 0x40, 0x0b, 0xa0, 0x08, 0x51, 0x12,
26        0x28, 0x89, 0x04, 0x65, 0x98, 0x30, 0x4c, 0x80, 0x96, 0x44, 0x12, 0x80,
27        0x21, 0x42, 0x12, 0x41, 0xc9, 0x04, 0x21, 0xc0, 0x32, 0x2d, 0x98, 0x00,
28        0x00, 0x49, 0x04, 0x08, 0x81, 0x96, 0x68, 0x82, 0xb0, 0x25, 0x08, 0x22,
29        0x48, 0x89, 0xa2, 0x40, 0x59, 0x26, 0x04, 0x90, 0x06, 0x40, 0x43, 0x30,
30        0x44, 0x92, 0x00, 0x69, 0x10, 0x82, 0x08, 0x08, 0xa4, 0x0d, 0x41, 0x12,
31        0x60, 0xc0, 0x00, 0x24, 0xd2, 0x22, 0x61, 0x08, 0x84, 0x04, 0x1b, 0x82,
32        0x01, 0xd3, 0x10, 0x01, 0x02, 0xa0, 0x44, 0xc0, 0x22, 0x60, 0x91, 0x14,
33        0x0c, 0x40, 0xa6, 0x04, 0xd2, 0x94, 0x20, 0x09, 0x94, 0x20, 0x52, 0x00,
34        0x08, 0x10, 0xa2, 0x4c, 0x00, 0x82, 0x01, 0x51, 0x10, 0x08, 0x8b, 0xa4,
35        0x25, 0x9a, 0x30, 0x44, 0x81, 0x10, 0x4c, 0x03, 0x02, 0x25, 0x52, 0x80,
36        0x08, 0x49, 0x84, 0x20, 0x50, 0x32, 0x00, 0x18, 0xa2, 0x40, 0x11, 0x24,
37        0x28, 0x01, 0x84, 0x01, 0x01, 0xa0, 0x41, 0x0a, 0x12, 0x45, 0x00, 0x36,
38        0x08, 0x00, 0x26, 0x29, 0x83, 0x82, 0x61, 0xc0, 0x80, 0x04, 0x10, 0x10,
39        0x6d, 0x00, 0x22, 0x48, 0x58, 0x26, 0x0c, 0xc2, 0x10, 0x48, 0x89, 0x24,
40        0x20, 0x58, 0x20, 0x45, 0x88, 0x24, 0x00, 0x19, 0x02, 0x25, 0xc0, 0x10,
41        0x68, 0x08, 0x14, 0x01, 0xca, 0x32, 0x28, 0x80, 0x00, 0x04, 0x4b, 0x26,
42        0x00, 0x13, 0x90, 0x60, 0x82, 0x80, 0x25, 0xd0, 0x00, 0x01, 0x10, 0x32,
43        0x0c, 0x43, 0x86, 0x21, 0x11, 0x00, 0x08, 0x43, 0x24, 0x04, 0x48, 0x10,
44        0x0c, 0x90, 0x92, 0x00, 0x43, 0x20, 0x2d, 0x00, 0x06, 0x09, 0x88, 0x24,
45        0x40, 0xc0, 0x32, 0x09, 0x09, 0x82, 0x00, 0x53, 0x80, 0x08, 0x80, 0x96,
46        0x41, 0x81, 0x00, 0x40, 0x48, 0x10, 0x48, 0x08, 0x96, 0x48, 0x58, 0x20,
47        0x29, 0xc3, 0x80, 0x20, 0x02, 0x94, 0x60, 0x92, 0x00, 0x20, 0x81, 0x22,
48        0x44, 0x10, 0xa0, 0x05, 0x40, 0x90, 0x01, 0x49, 0x20, 0x04, 0x0a, 0x00,
49        0x24, 0x89, 0x34, 0x48, 0x13, 0x80, 0x2c, 0xc0, 0x82, 0x29, 0x00, 0x24,
50        0x45, 0x08, 0x00, 0x08, 0x98, 0x36, 0x04, 0x52, 0x84, 0x04, 0xd0, 0x04,
51        0x00, 0x8a, 0x90, 0x44, 0x82, 0x32, 0x65, 0x18, 0x90, 0x00, 0x0a, 0x02,
52        0x01, 0x40, 0x02, 0x28, 0x40, 0xa4, 0x04, 0x92, 0x30, 0x04, 0x11, 0x86,
53        0x08, 0x42, 0x00, 0x2c, 0x52, 0x04, 0x08, 0xc9, 0x84, 0x60, 0x48, 0x12,
54        0x09, 0x99, 0x24, 0x44, 0x00, 0x24, 0x00, 0x03, 0x14, 0x21, 0x00, 0x10,
55        0x01, 0x1a, 0x32, 0x05, 0x88, 0x20, 0x40, 0x40, 0x06, 0x09, 0xc3, 0x84,
56        0x40, 0x01, 0x30, 0x60, 0x18, 0x02, 0x68, 0x11, 0x90, 0x0c, 0x02, 0xa2,
57        0x04, 0x00, 0x86, 0x29, 0x89, 0x14, 0x24, 0x82, 0x02, 0x41, 0x08, 0x80,
58        0x04, 0x19, 0x80, 0x08, 0x10, 0x12, 0x68, 0x42, 0xa4, 0x04, 0x00, 0x02,
59        0x61, 0x10, 0x06, 0x0c, 0x10, 0x00, 0x01, 0x12, 0x10, 0x20, 0x03, 0x94,
60        0x21, 0x42, 0x12, 0x65, 0x18, 0x94, 0x0c, 0x0a, 0x04, 0x28, 0x01, 0x14,
61        0x29, 0x0a, 0xa4, 0x40, 0xd0, 0x00, 0x40, 0x01, 0x90, 0x04, 0x41, 0x20,
62        0x2d, 0x40, 0x82, 0x48, 0xc1, 0x20, 0x00, 0x10, 0x30, 0x01, 0x08, 0x24,
63        0x04, 0x59, 0x84, 0x24, 0x00, 0x02, 0x29, 0x82, 0x00, 0x61, 0x58, 0x02,
64        0x48, 0x81, 0x16, 0x48, 0x10, 0x00, 0x21, 0x11, 0x06, 0x00, 0xca, 0xa0,
65        0x40, 0x02, 0x00, 0x04, 0x91, 0xb0, 0x00, 0x42, 0x04, 0x0c, 0x81, 0x06,
66        0x09, 0x48, 0x14, 0x25, 0x92, 0x20, 0x25, 0x11, 0xa0, 0x00, 0x0a, 0x86,
67        0x0c, 0xc1, 0x02, 0x48, 0x00, 0x20, 0x45, 0x08, 0x32, 0x00, 0x98, 0x06,
68        0x04, 0x13, 0x22, 0x00, 0x82, 0x04, 0x48, 0x81, 0x14, 0x44, 0x82, 0x12,
69        0x24, 0x18, 0x10, 0x40, 0x43, 0x80, 0x28, 0xd0, 0x04, 0x20, 0x81, 0x24,
```

```
 70      0x64, 0xd8, 0x00, 0x2c, 0x09, 0x12, 0x08, 0x41, 0xa2, 0x00, 0x00, 0x02,
 71      0x41, 0xca, 0x20, 0x41, 0xc0, 0x10, 0x01, 0x18, 0xa4, 0x04, 0x18, 0xa4,
 72      0x20, 0x12, 0x94, 0x20, 0x83, 0xa0, 0x40, 0x02, 0x32, 0x44, 0x80, 0x04,
 73      0x00, 0x18, 0x00, 0x0c, 0x40, 0x86, 0x60, 0x8a, 0x00, 0x64, 0x88, 0x12,
 74      0x05, 0x01, 0x82, 0x00, 0x4a, 0xa2, 0x01, 0xc1, 0x10, 0x61, 0x09, 0x04,
 75      0x01, 0x88, 0x00, 0x60, 0x01, 0xb4, 0x40, 0x08, 0x06, 0x01, 0x03, 0x80,
 76      0x08, 0x40, 0x94, 0x04, 0x8a, 0x20, 0x29, 0x80, 0x02, 0x0c, 0x52, 0x02,
 77      0x01, 0x42, 0x84, 0x00, 0x80, 0x84, 0x64, 0x02, 0x32, 0x48, 0x00, 0x30,
 78      0x44, 0x40, 0x22, 0x21, 0x00, 0x02, 0x08, 0xc3, 0xa0, 0x04, 0xd0, 0x20,
 79      0x40, 0x18, 0x16, 0x40, 0x40, 0x00, 0x28, 0x52, 0x90, 0x08, 0x82, 0x14,
 80      0x01, 0x18, 0x10, 0x08, 0x09, 0x82, 0x40, 0x0a, 0xa0, 0x20, 0x93, 0x80,
 81      0x08, 0xc0, 0x00, 0x20, 0x52, 0x00, 0x05, 0x01, 0x10, 0x40, 0x11, 0x06,
 82      0x0c, 0x82, 0x00, 0x00, 0x4b, 0x90, 0x44, 0x9a, 0x00, 0x28, 0x80, 0x90,
 83      0x04, 0x4a, 0x06, 0x09, 0x43, 0x02, 0x28, 0x00, 0x34, 0x01, 0x18, 0x00,
 84      0x65, 0x09, 0x80, 0x44, 0x03, 0x00, 0x24, 0x02, 0x82, 0x61, 0x48, 0x14,
 85      0x41, 0x00, 0x12, 0x28, 0x00, 0x34, 0x08, 0x51, 0x04, 0x05, 0x12, 0x90,
 86      0x28, 0x89, 0x84, 0x60, 0x12, 0x10, 0x49, 0x10, 0x26, 0x40, 0x49, 0x82,
 87      0x00, 0x91, 0x10, 0x01, 0x0a, 0x24, 0x40, 0x88, 0x10, 0x4c, 0x10, 0x04,
 88      0x00, 0x50, 0xa2, 0x2c, 0x40, 0x90, 0x48, 0x0a, 0xb0, 0x01, 0x50, 0x12,
 89      0x08, 0x00, 0xa4, 0x04, 0x09, 0xa0, 0x28, 0x92, 0x02, 0x00, 0x43, 0x10,
 90      0x21, 0x02, 0x20, 0x41, 0x81, 0x32, 0x00, 0x08, 0x04, 0x0c, 0x52, 0x00,
 91      0x21, 0x49, 0x84, 0x20, 0x10, 0x02, 0x01, 0x81, 0x10, 0x48, 0x40, 0x22,
 92      0x01, 0x01, 0x84, 0x69, 0xc1, 0x30, 0x01, 0xc8, 0x02, 0x44, 0x88, 0x00,
 93      0x0c, 0x01, 0x02, 0x2d, 0xc0, 0x12, 0x61, 0x00, 0xa0, 0x00, 0xc0, 0x30,
 94      0x40, 0x01, 0x12, 0x08, 0x0b, 0x20, 0x00, 0x80, 0x94, 0x40, 0x01, 0x84,
 95      0x40, 0x00, 0x32, 0x00, 0x10, 0x84, 0x00, 0x0b, 0x24, 0x00, 0x01, 0x06,
 96      0x29, 0x8a, 0x84, 0x41, 0x80, 0x10, 0x08, 0x08, 0x94, 0x4c, 0x03, 0x80,
 97      0x01, 0x40, 0x96, 0x40, 0x41, 0x20, 0x20, 0x50, 0x22, 0x25, 0x89, 0xa2,
 98      0x40, 0x40, 0xa4, 0x20, 0x02, 0x86, 0x28, 0x01, 0x20, 0x21, 0x4a, 0x10,
 99      0x08, 0x00, 0x14, 0x08, 0x40, 0x04, 0x25, 0x42, 0x02, 0x21, 0x43, 0x10,
100      0x04, 0x92, 0x00, 0x21, 0x11, 0xa0, 0x4c, 0x18, 0x22, 0x09, 0x03, 0x84,
101      0x41, 0x89, 0x10, 0x04, 0x82, 0x22, 0x24, 0x01, 0x14, 0x08, 0x08, 0x84,
102      0x08, 0xc1, 0x00, 0x09, 0x42, 0xb0, 0x41, 0x8a, 0x02, 0x00, 0x80, 0x36,
103      0x04, 0x49, 0xa0, 0x24, 0x91, 0x00, 0x00, 0x02, 0x94, 0x41, 0x92, 0x02,
104      0x01, 0x08, 0x06, 0x08, 0x09, 0x00, 0x01, 0xd0, 0x16, 0x28, 0x89, 0x80,
105      0x60, 0x00, 0x00, 0x68, 0x01, 0x90, 0x0c, 0x50, 0x20, 0x01, 0x40, 0x80,
106      0x40, 0x42, 0x30, 0x41, 0x00, 0x20, 0x25, 0x81, 0x06, 0x40, 0x49, 0x00,
107      0x08, 0x01, 0x12, 0x49, 0x00, 0xa0, 0x20, 0x18, 0x30, 0x05, 0x01, 0xa6,
108      0x00, 0x10, 0x24, 0x28, 0x00, 0x02, 0x20, 0xc8, 0x20, 0x00, 0x88, 0x12,
109      0x0c, 0x90, 0x92, 0x00, 0x02, 0x26, 0x01, 0x42, 0x16, 0x49, 0x00, 0x04,
110      0x24, 0x42, 0x02, 0x01, 0x88, 0x80, 0x0c, 0x1a, 0x80, 0x08, 0x10, 0x00,
111      0x60, 0x02, 0x94, 0x44, 0x88, 0x00, 0x69, 0x11, 0x30, 0x08, 0x12, 0xa0,
112      0x24, 0x13, 0x84, 0x00, 0x82, 0x00, 0x65, 0xc0, 0x10, 0x28, 0x00, 0x30,
113      0x04, 0x03, 0x20, 0x01, 0x11, 0x06, 0x01, 0xc8, 0x80, 0x00, 0xc2, 0x20,
114      0x08, 0x10, 0x82, 0x0c, 0x13, 0x02, 0x0c, 0x52, 0x06, 0x40, 0x00, 0xb0,
115      0x61, 0x40, 0x10, 0x01, 0x98, 0x86, 0x04, 0x10, 0x84, 0x08, 0x92, 0x14,
116      0x60, 0x41, 0x80, 0x41, 0x1a, 0x10, 0x04, 0x81, 0x22, 0x40, 0x41, 0x20,
117      0x29, 0x52, 0x00, 0x41, 0x08, 0x34, 0x60, 0x10, 0x00, 0x28, 0x01, 0x10,
118      0x40, 0x00, 0x84, 0x08, 0x42, 0x90, 0x20, 0x48, 0x04, 0x04, 0x52, 0x02,
119      0x00, 0x08, 0x20, 0x04, 0x00, 0x82, 0x0d, 0x00, 0x82, 0x40, 0x02, 0x10,
120      0x05, 0x48, 0x20, 0x40, 0x99, 0x00, 0x00, 0x01, 0x06, 0x24, 0xc0, 0x00,
121      0x68, 0x82, 0x04, 0x21, 0x12, 0x10, 0x44, 0x08, 0x04, 0x00, 0x40, 0xa6,
122      0x20, 0xd0, 0x16, 0x09, 0xc9, 0x24, 0x41, 0x02, 0x20, 0x0c, 0x09, 0x92,
123      0x40, 0x12, 0x00, 0x00, 0x40, 0x00, 0x09, 0x43, 0x84, 0x20, 0x98, 0x02,
124      0x01, 0x11, 0x24, 0x00, 0x43, 0x24, 0x00, 0x03, 0x90, 0x08, 0x41, 0x30,
125      0x24, 0x58, 0x20, 0x4c, 0x80, 0x82, 0x08, 0x10, 0x24, 0x25, 0x81, 0x06,
126      0x41, 0x09, 0x10, 0x20, 0x18, 0x10, 0x44, 0x80, 0x10, 0x00, 0x4a, 0x24,
127      0x0d, 0x01, 0x94, 0x28, 0x80, 0x30, 0x00, 0xc0, 0x02, 0x60, 0x10, 0x84,
128      0x0c, 0x02, 0x00, 0x09, 0x02, 0x82, 0x01, 0x08, 0x10, 0x04, 0xc2, 0x20,
129      0x68, 0x09, 0x06, 0x04, 0x18, 0x00, 0x00, 0x11, 0x90, 0x08, 0x0b, 0x10,
130      0x21, 0x82, 0x02, 0x0c, 0x10, 0xb6, 0x08, 0x00, 0x26, 0x00, 0x41, 0x02,
131      0x01, 0x4a, 0x24, 0x21, 0x1a, 0x20, 0x24, 0x80, 0x00, 0x44, 0x02, 0x00,
132      0x2d, 0x40, 0x02, 0x00, 0x8b, 0x94, 0x20, 0x10, 0x00, 0x20, 0x90, 0xa6,
133      0x40, 0x13, 0x00, 0x2c, 0x11, 0x86, 0x61, 0x01, 0x80, 0x41, 0x10, 0x02,
134      0x04, 0x81, 0x30, 0x48, 0x48, 0x20, 0x28, 0x50, 0x80, 0x21, 0x8a, 0x10,
135      0x04, 0x08, 0x10, 0x09, 0x10, 0x10, 0x48, 0x42, 0xa0, 0x0c, 0x82, 0x92,
```

```
136       0x60, 0xc0, 0x20, 0x05, 0xd2, 0x20, 0x40, 0x01, 0x00, 0x04, 0x08, 0x82,
137       0x2d, 0x82, 0x02, 0x00, 0x48, 0x80, 0x41, 0x48, 0x10, 0x00, 0x91, 0x04,
138       0x04, 0x03, 0x84, 0x00, 0xc2, 0x04, 0x68, 0x00, 0x00, 0x64, 0xc0, 0x22,
139       0x40, 0x08, 0x32, 0x44, 0x09, 0x86, 0x00, 0x91, 0x02, 0x28, 0x01, 0x00,
140       0x64, 0x48, 0x00, 0x24, 0x10, 0x90, 0x00, 0x43, 0x00, 0x21, 0x52, 0x86,
141       0x41, 0x8b, 0x90, 0x20, 0x40, 0x20, 0x08, 0x88, 0x04, 0x44, 0x13, 0x20,
142       0x00, 0x02, 0x84, 0x60, 0x81, 0x90, 0x24, 0x40, 0x30, 0x00, 0x08, 0x10,
143       0x08, 0x08, 0x02, 0x01, 0x10, 0x04, 0x20, 0x43, 0xb4, 0x40, 0x90, 0x12,
144       0x68, 0x01, 0x80, 0x4c, 0x18, 0x00, 0x08, 0xc0, 0x12, 0x49, 0x40, 0x10,
145       0x24, 0x1a, 0x00, 0x41, 0x89, 0x24, 0x4c, 0x10, 0x00, 0x04, 0x52, 0x10,
146       0x09, 0x4a, 0x20, 0x41, 0x48, 0x22, 0x69, 0x11, 0x14, 0x08, 0x10, 0x06,
147       0x24, 0x80, 0x84, 0x28, 0x00, 0x10, 0x00, 0x40, 0x10, 0x01, 0x08, 0x26,
148       0x08, 0x48, 0x06, 0x28, 0x00, 0x14, 0x01, 0x42, 0x84, 0x04, 0x0a, 0x20,
149       0x00, 0x01, 0x82, 0x08, 0x00, 0x82, 0x24, 0x12, 0x04, 0x40, 0x40, 0xa0,
150       0x40, 0x90, 0x10, 0x04, 0x90, 0x22, 0x40, 0x10, 0x20, 0x2c, 0x80, 0x10,
151       0x28, 0x43, 0x00, 0x04, 0x58, 0x00, 0x01, 0x81, 0x10, 0x48, 0x09, 0x20,
152       0x21, 0x83, 0x04, 0x00, 0x42, 0xa4, 0x44, 0x00, 0x00, 0x6c, 0x10, 0xa0,
153       0x44, 0x48, 0x80, 0x00, 0x83, 0x80, 0x48, 0xc9, 0x00, 0x00, 0x00, 0x02,
154       0x05, 0x10, 0xb0, 0x04, 0x13, 0x04, 0x29, 0x10, 0x92, 0x40, 0x08, 0x04,
155       0x44, 0x82, 0x22, 0x00, 0x19, 0x20, 0x00, 0x19, 0x20, 0x01, 0x81, 0x90,
156       0x60, 0x8a, 0x00, 0x41, 0xc0, 0x02, 0x45, 0x10, 0x04, 0x00, 0x02, 0xa2,
157       0x09, 0x40, 0x10, 0x21, 0x49, 0x20, 0x01, 0x42, 0x30, 0x2c, 0x00, 0x14,
158       0x44, 0x01, 0x22, 0x04, 0x02, 0x92, 0x08, 0x89, 0x04, 0x21, 0x80, 0x10,
159       0x05, 0x01, 0x20, 0x40, 0x41, 0x80, 0x04, 0x00, 0x12, 0x09, 0x40, 0xb0,
160       0x64, 0x58, 0x32, 0x01, 0x08, 0x90, 0x00, 0x41, 0x04, 0x09, 0xc1, 0x80,
161       0x61, 0x08, 0x90, 0x00, 0x9a, 0x00, 0x24, 0x01, 0x12, 0x08, 0x02, 0x26,
162       0x05, 0x82, 0x06, 0x08, 0x08, 0x00, 0x20, 0x48, 0x20, 0x00, 0x18, 0x24,
163       0x48, 0x03, 0x02, 0x00, 0x11, 0x00, 0x09, 0x00, 0x84, 0x01, 0x4a, 0x10,
164       0x01, 0x98, 0x00, 0x04, 0x18, 0x86, 0x00, 0xc0, 0x00, 0x20, 0x81, 0x80,
165       0x04, 0x10, 0x30, 0x05, 0x00, 0xb4, 0x0c, 0x4a, 0x82, 0x29, 0x91, 0x02,
166       0x28, 0x00, 0x20, 0x44, 0xc0, 0x00, 0x2c, 0x91, 0x80, 0x40, 0x01, 0xa2,
167       0x00, 0x12, 0x04, 0x09, 0xc3, 0x20, 0x00, 0x08, 0x02, 0x0c, 0x10, 0x22,
168       0x04, 0x00, 0x00, 0x2c, 0x11, 0x86, 0x00, 0xc0, 0x00, 0x00, 0x12, 0x32,
169       0x40, 0x89, 0x80, 0x40, 0x40, 0x02, 0x05, 0x50, 0x86, 0x60, 0x82, 0xa4,
170       0x60, 0x0a, 0x12, 0x4d, 0x80, 0x90, 0x08, 0x12, 0x80, 0x09, 0x02, 0x14,
171       0x48, 0x01, 0x24, 0x20, 0x8a, 0x00, 0x44, 0x90, 0x04, 0x04, 0x01, 0x02,
172       0x00, 0xd1, 0x12, 0x00, 0x0a, 0x04, 0x40, 0x00, 0x32, 0x21, 0x81, 0x24,
173       0x08, 0x19, 0x84, 0x20, 0x02, 0x04, 0x08, 0x89, 0x80, 0x24, 0x02, 0x02,
174       0x68, 0x18, 0x82, 0x44, 0x42, 0x00, 0x21, 0x40, 0x00, 0x28, 0x01, 0x80,
175       0x45, 0x82, 0x20, 0x40, 0x11, 0x80, 0x0c, 0x02, 0x00, 0x24, 0x40, 0x90,
176       0x01, 0x40, 0x20, 0x20, 0x50, 0x20, 0x28, 0x19, 0x00, 0x40, 0x09, 0x20,
177       0x08, 0x80, 0x04, 0x60, 0x40, 0x80, 0x20, 0x08, 0x30, 0x49, 0x09, 0x34,
178       0x00, 0x11, 0x24, 0x24, 0x82, 0x00, 0x41, 0xc2, 0x00, 0x04, 0x92, 0x02,
179       0x24, 0x80, 0x00, 0x0c, 0x02, 0xa0, 0x00, 0x01, 0x06, 0x60, 0x41, 0x04,
180       0x21, 0xd0, 0x00, 0x01, 0x01, 0x00, 0x48, 0x12, 0x84, 0x04, 0x91, 0x12,
181       0x08, 0x00, 0x24, 0x44, 0x00, 0x12, 0x41, 0x18, 0x26, 0x0c, 0x41, 0x80,
182       0x00, 0x52, 0x04, 0x20, 0x09, 0x00, 0x24, 0x90, 0x20, 0x48, 0x18, 0x02,
183       0x00, 0x03, 0xa2, 0x09, 0xd0, 0x14, 0x00, 0x8a, 0x84, 0x25, 0x4a, 0x00,
184       0x20, 0x98, 0x14, 0x40, 0x00, 0xa2, 0x05, 0x00, 0x00, 0x00, 0x40, 0x14,
185       0x01, 0x58, 0x20, 0x2c, 0x80, 0x84, 0x00, 0x09, 0x20, 0x20, 0x91, 0x02,
186       0x08, 0x02, 0xb0, 0x41, 0x08, 0x30, 0x00, 0x09, 0x10, 0x00, 0x18, 0x02,
187       0x21, 0x02, 0x02, 0x00, 0x00, 0x24, 0x44, 0x08, 0x12, 0x60, 0x00, 0xb2,
188       0x44, 0x12, 0x02, 0x0c, 0xc0, 0x80, 0x40, 0xc8, 0x20, 0x04, 0x50, 0x20,
189       0x05, 0x00, 0xb0, 0x04, 0x0b, 0x04, 0x29, 0x53, 0x00, 0x61, 0x48, 0x30,
190       0x00, 0x82, 0x20, 0x29, 0x00, 0x16, 0x00, 0x53, 0x22, 0x20, 0x43, 0x10,
191       0x48, 0x00, 0x80, 0x04, 0xd2, 0x00, 0x40, 0x00, 0xa2, 0x44, 0x03, 0x80,
192       0x29, 0x00, 0x04, 0x08, 0xc0, 0x04, 0x64, 0x40, 0x30, 0x28, 0x09, 0x84,
193       0x44, 0x50, 0x80, 0x21, 0x02, 0x92, 0x00, 0xc0, 0x10, 0x60, 0x88, 0x22,
194       0x08, 0x80, 0x00, 0x00, 0x18, 0x84, 0x04, 0x83, 0x96, 0x00, 0x81, 0x20,
195       0x05, 0x02, 0x00, 0x45, 0x88, 0x84, 0x00, 0x51, 0x20, 0x20, 0x51, 0x86,
196       0x41, 0x4b, 0x94, 0x00, 0x80, 0x00, 0x08, 0x11, 0x20, 0x4c, 0x58, 0x80,
197       0x04, 0x03, 0x06, 0x20, 0x89, 0x00, 0x05, 0x08, 0x22, 0x05, 0x90, 0x00,
198       0x40, 0x00, 0x82, 0x09, 0x50, 0x00, 0x00, 0x00, 0xa0, 0x41, 0xc2, 0x20,
199       0x08, 0x00, 0x16, 0x08, 0x40, 0x26, 0x21, 0xd0, 0x90, 0x08, 0x81, 0x90,
200       0x41, 0x00, 0x02, 0x44, 0x08, 0x10, 0x0c, 0x0a, 0x86, 0x09, 0x90, 0x04,
201       0x00, 0xc8, 0xa0, 0x04, 0x08, 0x30, 0x20, 0x89, 0x84, 0x00, 0x11, 0x22,
```

```
202       0x2c, 0x40, 0x00, 0x08, 0x02, 0xb0, 0x01, 0x48, 0x02, 0x01, 0x09, 0x20,
203       0x04, 0x03, 0x04, 0x00, 0x80, 0x02, 0x60, 0x42, 0x30, 0x21, 0x4a, 0x10,
204       0x44, 0x09, 0x02, 0x00, 0x01, 0x24, 0x00, 0x12, 0x82, 0x21, 0x80, 0xa4,
205       0x20, 0x10, 0x02, 0x04, 0x91, 0xa0, 0x40, 0x18, 0x04, 0x00, 0x02, 0x06,
206       0x69, 0x09, 0x00, 0x05, 0x58, 0x02, 0x01, 0x00, 0x00, 0x48, 0x00, 0x00,
207       0x00, 0x03, 0x92, 0x20, 0x00, 0x34, 0x01, 0xc8, 0x20, 0x48, 0x08, 0x30,
208       0x08, 0x42, 0x80, 0x20, 0x91, 0x90, 0x68, 0x01, 0x04, 0x40, 0x12, 0x02,
209       0x61, 0x00, 0x12, 0x08, 0x01, 0xa0, 0x00, 0x11, 0x04, 0x21, 0x48, 0x04,
210       0x24, 0x92, 0x00, 0x0c, 0x01, 0x84, 0x04, 0x00, 0x00, 0x01, 0x12, 0x96,
211       0x40, 0x01, 0xa0, 0x41, 0x88, 0x22, 0x28, 0x88, 0x00, 0x44, 0x42, 0x80,
212       0x24, 0x12, 0x14, 0x01, 0x42, 0x90, 0x60, 0x1a, 0x10, 0x04, 0x81, 0x10,
213       0x48, 0x08, 0x06, 0x29, 0x83, 0x02, 0x40, 0x02, 0x24, 0x64, 0x80, 0x10,
214       0x05, 0x80, 0x10, 0x40, 0x02, 0x02, 0x08, 0x42, 0x84, 0x01, 0x09, 0x20,
215       0x04, 0x50, 0x00, 0x60, 0x11, 0x30, 0x40, 0x13, 0x02, 0x04, 0x81, 0x00,
216       0x09, 0x08, 0x20, 0x45, 0x4a, 0x10, 0x61, 0x90, 0x26, 0x0c, 0x08, 0x02,
217       0x21, 0x91, 0x00, 0x60, 0x02, 0x04, 0x00, 0x02, 0x00, 0x0c, 0x08, 0x06,
218       0x08, 0x48, 0x84, 0x08, 0x11, 0x02, 0x00, 0x80, 0xa4, 0x00, 0x5a, 0x20,
219       0x00, 0x88, 0x04, 0x04, 0x02, 0x00, 0x09, 0x00, 0x14, 0x08, 0x49, 0x14,
220       0x20, 0xc8, 0x00, 0x04, 0x91, 0xa0, 0x40, 0x59, 0x80, 0x00, 0x12, 0x10,
221       0x00, 0x80, 0x80, 0x65, 0x00, 0x00, 0x04, 0x00, 0x80, 0x40, 0x19, 0x00,
222       0x21, 0x03, 0x84, 0x60, 0xc0, 0x04, 0x24, 0x1a, 0x12, 0x61, 0x80, 0x80,
223       0x08, 0x02, 0x04, 0x09, 0x42, 0x12, 0x20, 0x08, 0x34, 0x04, 0x90, 0x20,
224       0x01, 0x01, 0xa0, 0x00, 0x0b, 0x00, 0x08, 0x91, 0x92, 0x40, 0x02, 0x34,
225       0x40, 0x88, 0x10, 0x61, 0x19, 0x02, 0x00, 0x40, 0x04, 0x25, 0xc0, 0x80,
226       0x68, 0x08, 0x04, 0x21, 0x80, 0x22, 0x04, 0x00, 0xa0, 0x0c, 0x01, 0x84,
227       0x20, 0x41, 0x00, 0x08, 0x8a, 0x00, 0x20, 0x8a, 0x00, 0x48, 0x88, 0x04,
228       0x04, 0x11, 0x82, 0x08, 0x40, 0x86, 0x09, 0x49, 0xa4, 0x40, 0x00, 0x10,
229       0x01, 0x01, 0xa2, 0x04, 0x50, 0x80, 0x0c, 0x80, 0x00, 0x48, 0x82, 0xa0,
230       0x01, 0x18, 0x12, 0x41, 0x01, 0x04, 0x48, 0x41, 0x00, 0x24, 0x01, 0x00,
231       0x00, 0x88, 0x14, 0x00, 0x02, 0x00, 0x68, 0x01, 0x20, 0x08, 0x4a, 0x22,
232       0x08, 0x83, 0x80, 0x00, 0x89, 0x04, 0x01, 0xc2, 0x00, 0x00, 0x00, 0x34,
233       0x04, 0x00, 0x82, 0x28, 0x02, 0x02, 0x41, 0x4a, 0x90, 0x05, 0x82, 0x02,
234       0x09, 0x80, 0x24, 0x04, 0x41, 0x00, 0x01, 0x92, 0x80, 0x28, 0x01, 0x14,
235       0x00, 0x50, 0x20, 0x4c, 0x10, 0xb0, 0x04, 0x43, 0xa4, 0x21, 0x90, 0x04,
236       0x01, 0x02, 0x00, 0x44, 0x48, 0x00, 0x64, 0x08, 0x06, 0x00, 0x42, 0x20,
237       0x08, 0x02, 0x92, 0x01, 0x4a, 0x00, 0x20, 0x50, 0x32, 0x25, 0x90, 0x22,
238       0x04, 0x09, 0x00, 0x08, 0x11, 0x80, 0x21, 0x01, 0x10, 0x05, 0x00, 0x32,
239       0x08, 0x88, 0x94, 0x08, 0x08, 0x24, 0x0d, 0xc1, 0x80, 0x40, 0x0b, 0x20,
240       0x40, 0x18, 0x12, 0x04, 0x00, 0x22, 0x40, 0x10, 0x26, 0x05, 0xc1, 0x82,
241       0x00, 0x01, 0x30, 0x24, 0x02, 0x22, 0x41, 0x08, 0x24, 0x48, 0x1a, 0x00,
242       0x25, 0xd2, 0x12, 0x28, 0x42, 0x00, 0x04, 0x40, 0x30, 0x41, 0x00, 0x02,
243       0x00, 0x13, 0x20, 0x24, 0xd1, 0x84, 0x08, 0x89, 0x80, 0x04, 0x52, 0x00,
244       0x44, 0x18, 0xa4, 0x00, 0x00, 0x06, 0x20, 0x91, 0x10, 0x09, 0x42, 0x20,
245       0x24, 0x40, 0x30, 0x28, 0x00, 0x84, 0x40, 0x40, 0x80, 0x08, 0x10, 0x04,
246       0x09, 0x08, 0x04, 0x40, 0x08, 0x22, 0x00, 0x19, 0x02, 0x00, 0x00, 0x80,
247       0x2c, 0x02, 0x02, 0x21, 0x01, 0x90, 0x20, 0x40, 0x00, 0x0c, 0x00, 0x34,
248       0x48, 0x58, 0x20, 0x01, 0x43, 0x04, 0x20, 0x80, 0x14, 0x00, 0x90, 0x00,
249       0x6d, 0x11, 0x00, 0x00, 0x40, 0x20, 0x00, 0x03, 0x10, 0x40, 0x88, 0x30,
250       0x05, 0x4a, 0x00, 0x65, 0x10, 0x24, 0x08, 0x18, 0x84, 0x28, 0x03, 0x80,
251       0x20, 0x42, 0xb0, 0x40, 0x00, 0x10, 0x69, 0x19, 0x04, 0x00, 0x00, 0x80,
252       0x04, 0xc2, 0x04, 0x00, 0x01, 0x00, 0x05, 0x00, 0x22, 0x25, 0x08, 0x96,
253       0x04, 0x02, 0x22, 0x00, 0xd0, 0x10, 0x29, 0x01, 0xa0, 0x60, 0x08, 0x10,
254       0x04, 0x01, 0x16, 0x44, 0x10, 0x02, 0x28, 0x02, 0x82, 0x48, 0x40, 0x84,
255       0x20, 0x90, 0x22, 0x28, 0x80, 0x04, 0x00, 0x40, 0x04, 0x24, 0x00, 0x80,
256       0x29, 0x03, 0x10, 0x60, 0x48, 0x00, 0x00, 0x81, 0xa0, 0x00, 0x51, 0x20,
257       0x0c, 0xd1, 0x00, 0x01, 0x41, 0x20, 0x04, 0x92, 0x00, 0x00, 0x10, 0x92,
258       0x00, 0x42, 0x04, 0x05, 0x01, 0x86, 0x40, 0x80, 0x10, 0x20, 0x52, 0x20,
259       0x21, 0x00, 0x10, 0x48, 0x0a, 0x02, 0x00, 0xd0, 0x12, 0x41, 0x48, 0x80,
260       0x04, 0x00, 0x00, 0x48, 0x09, 0x22, 0x04, 0x00, 0x24, 0x00, 0x43, 0x10,
261       0x60, 0x0a, 0x00, 0x44, 0x12, 0x20, 0x2c, 0x08, 0x20, 0x44, 0x00, 0x84,
262       0x09, 0x40, 0x06, 0x08, 0xc1, 0x00, 0x40, 0x80, 0x20, 0x00, 0x98, 0x12,
263       0x48, 0x10, 0xa2, 0x20, 0x00, 0x84, 0x48, 0xc0, 0x10, 0x20, 0x90, 0x12,
264       0x08, 0x98, 0x82, 0x00, 0x0a, 0xa0, 0x04, 0x03, 0x00, 0x28, 0xc3, 0x00,
265       0x44, 0x42, 0x10, 0x04, 0x08, 0x04, 0x40, 0x00, 0x00, 0x05, 0x10, 0x00,
266       0x21, 0x03, 0x80, 0x04, 0x88, 0x12, 0x69, 0x10, 0x00, 0x04, 0x08, 0x04,
267       0x04, 0x02, 0x84, 0x48, 0x49, 0x04, 0x20, 0x18, 0x02, 0x64, 0x80, 0x30,
```

```
268       0x08, 0x01, 0x02, 0x00, 0x52, 0x12, 0x49, 0x08, 0x20, 0x41, 0x88, 0x10,
269       0x48, 0x08, 0x34, 0x00, 0x01, 0x86, 0x05, 0xd0, 0x00, 0x00, 0x83, 0x84,
270       0x21, 0x40, 0x02, 0x41, 0x10, 0x80, 0x48, 0x40, 0xa2, 0x20, 0x51, 0x00,
271       0x00, 0x49, 0x00, 0x01, 0x90, 0x20, 0x40, 0x18, 0x02, 0x40, 0x02, 0x22,
272       0x05, 0x40, 0x80, 0x08, 0x82, 0x10, 0x20, 0x18, 0x00, 0x05, 0x01, 0x82,
273       0x40, 0x58, 0x00, 0x04, 0x81, 0x90, 0x29, 0x01, 0xa0, 0x64, 0x00, 0x22,
274       0x40, 0x01, 0xa2, 0x00, 0x18, 0x04, 0x0d, 0x00, 0x00, 0x60, 0x80, 0x94,
275       0x60, 0x82, 0x10, 0x0d, 0x80, 0x30, 0x0c, 0x12, 0x20, 0x00, 0x00, 0x12,
276       0x40, 0xc0, 0x20, 0x21, 0x58, 0x02, 0x41, 0x10, 0x80, 0x44, 0x03, 0x02,
277       0x04, 0x13, 0x90, 0x29, 0x08, 0x00, 0x44, 0xc0, 0x00, 0x21, 0x00, 0x26,
278       0x00, 0x1a, 0x80, 0x01, 0x13, 0x14, 0x20, 0x0a, 0x14, 0x20, 0x00, 0x32,
279       0x61, 0x08, 0x00, 0x40, 0x42, 0x20, 0x09, 0x80, 0x06, 0x01, 0x81, 0x80,
280       0x60, 0x42, 0x00, 0x68, 0x90, 0x82, 0x08, 0x42, 0x80, 0x04, 0x02, 0x80,
281       0x09, 0x0b, 0x04, 0x00, 0x98, 0x00, 0x0c, 0x81, 0x06, 0x44, 0x48, 0x84,
282       0x28, 0x03, 0x92, 0x00, 0x01, 0x80, 0x40, 0x0a, 0x00, 0x0c, 0x81, 0x02,
283       0x08, 0x51, 0x04, 0x28, 0x90, 0x02, 0x20, 0x09, 0x10, 0x60, 0x00, 0x00,
284       0x09, 0x81, 0xa0, 0x0c, 0x00, 0xa4, 0x09, 0x00, 0x02, 0x28, 0x80, 0x20,
285       0x00, 0x02, 0x02, 0x04, 0x81, 0x14, 0x04, 0x00, 0x04, 0x09, 0x11, 0x12,
286       0x60, 0x40, 0x20, 0x01, 0x48, 0x30, 0x40, 0x11, 0x00, 0x08, 0x0a, 0x86,
287       0x00, 0x00, 0x04, 0x60, 0x81, 0x04, 0x01, 0xd0, 0x02, 0x41, 0x18, 0x90,
288       0x00, 0x0a, 0x20, 0x00, 0xc1, 0x06, 0x01, 0x08, 0x80, 0x64, 0xca, 0x10,
289       0x04, 0x99, 0x80, 0x48, 0x01, 0x82, 0x20, 0x50, 0x90, 0x48, 0x80, 0x84,
290       0x20, 0x90, 0x22, 0x00, 0x19, 0x00, 0x04, 0x18, 0x20, 0x24, 0x10, 0x86,
291       0x40, 0xc2, 0x00, 0x24, 0x12, 0x10, 0x44, 0x00, 0x16, 0x08, 0x10, 0x24,
292       0x00, 0x12, 0x06, 0x01, 0x08, 0x90, 0x00, 0x12, 0x02, 0x4d, 0x10, 0x80,
293       0x40, 0x50, 0x22, 0x00, 0x43, 0x10, 0x01, 0x00, 0x30, 0x21, 0x0a, 0x00,
294       0x00, 0x01, 0x14, 0x00, 0x10, 0x84, 0x04, 0xc1, 0x10, 0x29, 0x0a, 0x00,
295       0x01, 0x8a, 0x00, 0x20, 0x01, 0x12, 0x0c, 0x49, 0x20, 0x04, 0x81, 0x00,
296       0x48, 0x01, 0x04, 0x60, 0x80, 0x12, 0x0c, 0x08, 0x10, 0x48, 0x4a, 0x04,
297       0x28, 0x10, 0x00, 0x28, 0x40, 0x84, 0x45, 0x50, 0x10, 0x60, 0x10, 0x06,
298       0x44, 0x01, 0x80, 0x09, 0x00, 0x86, 0x01, 0x42, 0xa0, 0x00, 0x90, 0x00,
299       0x05, 0x90, 0x22, 0x40, 0x41, 0x00, 0x08, 0x80, 0x02, 0x08, 0xc0, 0x00,
300       0x01, 0x58, 0x30, 0x49, 0x09, 0x14, 0x00, 0x41, 0x02, 0x0c, 0x02, 0x80,
301       0x40, 0x89, 0x00, 0x24, 0x08, 0x10, 0x05, 0x90, 0x32, 0x40, 0x0a, 0x82,
302       0x08, 0x00, 0x12, 0x61, 0x00, 0x04, 0x21, 0x00, 0x22, 0x04, 0x10, 0x24,
303       0x08, 0x0a, 0x04, 0x01, 0x10, 0x00, 0x20, 0x40, 0x84, 0x04, 0x88, 0x22,
304       0x20, 0x90, 0x12, 0x00, 0x53, 0x06, 0x24, 0x01, 0x04, 0x40, 0x0b, 0x14,
305       0x60, 0x82, 0x02, 0x0d, 0x10, 0x90, 0x0c, 0x08, 0x20, 0x09, 0x00, 0x14,
306       0x09, 0x80, 0x80, 0x24, 0x82, 0x00, 0x40, 0x01, 0x02, 0x44, 0x01, 0x20,
307       0x0c, 0x40, 0x84, 0x40, 0x0a, 0x10, 0x41, 0x00, 0x30, 0x05, 0x09, 0x80,
308       0x44, 0x08, 0x20, 0x20, 0x02, 0x00, 0x49, 0x43, 0x20, 0x21, 0x00, 0x20,
309       0x00, 0x01, 0xb6, 0x08, 0x40, 0x04, 0x08, 0x02, 0x80, 0x01, 0x41, 0x80,
310       0x40, 0x08, 0x10, 0x24, 0x00, 0x20, 0x04, 0x12, 0x86, 0x09, 0xc0, 0x12,
311       0x21, 0x81, 0x14, 0x04, 0x00, 0x02, 0x20, 0x89, 0xb4, 0x44, 0x12, 0x80,
312       0x00, 0xd1, 0x00, 0x69, 0x40, 0x80, 0x00, 0x42, 0x12, 0x00, 0x18, 0x04,
313       0x00, 0x49, 0x06, 0x21, 0x02, 0x04, 0x28, 0x02, 0x84, 0x01, 0xc0, 0x10,
314       0x68, 0x00, 0x20, 0x08, 0x40, 0x00, 0x08, 0x91, 0x10, 0x01, 0x81, 0x24,
315       0x04, 0xd2, 0x10, 0x4c, 0x88, 0x86, 0x00, 0x10, 0x80, 0x0c, 0x02, 0x14,
316       0x00, 0x8a, 0x90, 0x40, 0x18, 0x20, 0x21, 0x80, 0xa4, 0x00, 0x58, 0x24,
317       0x20, 0x10, 0x10, 0x60, 0xc1, 0x30, 0x41, 0x48, 0x02, 0x48, 0x09, 0x00,
318       0x40, 0x09, 0x02, 0x05, 0x11, 0x82, 0x20, 0x4a, 0x20, 0x24, 0x18, 0x02,
319       0x0c, 0x10, 0x22, 0x0c, 0x0a, 0x04, 0x00, 0x03, 0x06, 0x48, 0x48, 0x04,
320       0x04, 0x02, 0x00, 0x21, 0x80, 0x84, 0x00, 0x18, 0x00, 0x0c, 0x02, 0x12,
321       0x01, 0x00, 0x14, 0x05, 0x82, 0x10, 0x41, 0x89, 0x12, 0x08, 0x40, 0xa4,
322       0x21, 0x01, 0x84, 0x48, 0x02, 0x10, 0x60, 0x40, 0x02, 0x28, 0x00, 0x14,
323       0x08, 0x40, 0xa0, 0x20, 0x51, 0x12, 0x00, 0xc2, 0x00, 0x01, 0x1a, 0x30,
324       0x40, 0x89, 0x12, 0x4c, 0x02, 0x80, 0x00, 0x00, 0x14, 0x01, 0x01, 0xa0,
325       0x21, 0x18, 0x22, 0x21, 0x18, 0x06, 0x40, 0x01, 0x80, 0x00, 0x90, 0x04,
326       0x48, 0x02, 0x30, 0x04, 0x08, 0x00, 0x05, 0x88, 0x24, 0x08, 0x48, 0x04,
327       0x24, 0x02, 0x06, 0x00, 0x80, 0x00, 0x00, 0x00, 0x10, 0x65, 0x11, 0x90,
328       0x00, 0x0a, 0x82, 0x04, 0xc3, 0x04, 0x60, 0x48, 0x24, 0x04, 0x92, 0x02,
329       0x44, 0x88, 0x80, 0x40, 0x18, 0x06, 0x29, 0x80, 0x10, 0x01, 0x00, 0x00,
330       0x44, 0xc8, 0x10, 0x21, 0x89, 0x30, 0x00, 0x4b, 0xa0, 0x01, 0x10, 0x14,
331       0x00, 0x02, 0x94, 0x40, 0x00, 0x20, 0x65, 0x00, 0xa2, 0x0c, 0x40, 0x22,
332       0x20, 0x81, 0x12, 0x20, 0x82, 0x04, 0x01, 0x10, 0x00, 0x08, 0x88, 0x00,
333       0x00, 0x11, 0x80, 0x04, 0x42, 0x80, 0x40, 0x41, 0x14, 0x00, 0x40, 0x32,
```

```
334         0x2c, 0x80, 0x24, 0x04, 0x19, 0x00, 0x00, 0x91, 0x00, 0x20, 0x83, 0x00,
335         0x05, 0x40, 0x20, 0x09, 0x01, 0x84, 0x40, 0x40, 0x20, 0x20, 0x11, 0x00,
336         0x40, 0x41, 0x90, 0x20, 0x00, 0x00, 0x40, 0x90, 0x92, 0x48, 0x18, 0x06,
337         0x08, 0x81, 0x80, 0x48, 0x01, 0x34, 0x24, 0x10, 0x20, 0x04, 0x00, 0x20,
338         0x04, 0x18, 0x06, 0x2d, 0x90, 0x10, 0x01, 0x00, 0x90, 0x00, 0x0a, 0x22,
339         0x01, 0x00, 0x22, 0x00, 0x11, 0x84, 0x01, 0x01, 0x00, 0x20, 0x88, 0x00,
340         0x44, 0x00, 0x22, 0x01, 0x00, 0xa6, 0x40, 0x02, 0x06, 0x20, 0x11, 0x00,
341         0x01, 0xc8, 0xa0, 0x04, 0x8a, 0x00, 0x28, 0x19, 0x80, 0x00, 0x52, 0xa0,
342         0x24, 0x12, 0x12, 0x09, 0x08, 0x24, 0x01, 0x48, 0x00, 0x04, 0x00, 0x24,
343         0x40, 0x02, 0x84, 0x08, 0x00, 0x04, 0x48, 0x40, 0x90, 0x60, 0x0a, 0x22,
344         0x01, 0x88, 0x14, 0x08, 0x01, 0x02, 0x08, 0xd3, 0x00, 0x20, 0xc0, 0x90,
345         0x24, 0x10, 0x00, 0x00, 0x01, 0xb0, 0x08, 0x0a, 0xa0, 0x00, 0x80, 0x00,
346         0x01, 0x09, 0x00, 0x20, 0x52, 0x02, 0x25, 0x00, 0x24, 0x04, 0x02, 0x84,
347         0x24, 0x10, 0x92, 0x40, 0x02, 0xa0, 0x40, 0x00, 0x22, 0x08, 0x11, 0x04,
348         0x08, 0x01, 0x22, 0x00, 0x42, 0x14, 0x00, 0x09, 0x90, 0x21, 0x00, 0x30,
349         0x6c, 0x00, 0x00, 0x0c, 0x00, 0x22, 0x09, 0x90, 0x10, 0x28, 0x40, 0x00,
350         0x20, 0xc0, 0x20, 0x00, 0x90, 0x00, 0x40, 0x01, 0x82, 0x05, 0x12, 0x12,
351         0x09, 0xc1, 0x04, 0x61, 0x80, 0x02, 0x28, 0x81, 0x24, 0x00, 0x49, 0x04,
352         0x08, 0x10, 0x86, 0x29, 0x41, 0x80, 0x21, 0x0a, 0x30, 0x49, 0x88, 0x90,
353         0x00, 0x41, 0x04, 0x29, 0x81, 0x80, 0x41, 0x09, 0x00, 0x40, 0x12, 0x10,
354         0x40, 0x00, 0x10, 0x40, 0x48, 0x02, 0x05, 0x80, 0x02, 0x21, 0x40, 0x20,
355         0x00, 0x58, 0x20, 0x60, 0x00, 0x90, 0x48, 0x00, 0x80, 0x28, 0xc0, 0x80,
356         0x48, 0x00, 0x00, 0x44, 0x80, 0x02, 0x00, 0x09, 0x06, 0x00, 0x12, 0x02,
357         0x01, 0x00, 0x10, 0x08, 0x83, 0x10, 0x45, 0x12, 0x00, 0x2c, 0x08, 0x04,
358         0x44, 0x00, 0x20, 0x20, 0xc0, 0x10, 0x20, 0x01, 0x00, 0x05, 0xc8, 0x20,
359         0x04, 0x98, 0x10, 0x08, 0x10, 0x00, 0x24, 0x02, 0x16, 0x40, 0x88, 0x00,
360         0x61, 0x88, 0x12, 0x24, 0x80, 0xa6, 0x00, 0x42, 0x00, 0x08, 0x10, 0x06,
361         0x48, 0x40, 0xa0, 0x00, 0x50, 0x20, 0x04, 0x81, 0xa4, 0x40, 0x18, 0x00,
362         0x08, 0x10, 0x80, 0x01, 0x01};
363     #if defined RSA_KEY_SIEVE && defined SIMULATION
364     UINT32  PrimeIndex = 0;
365     UINT32  failedAtIteration[10] = {0};
366     UINT32  PrimeCounts[3] = {0};
367     UINT32  MillerRabinTrials[3] = {0};
368     UINT32  totalFieldsSieved[3] = {0};
369     UINT32  bitsInFieldAfterSieve[3] = {0};
370     UINT32  emptyFieldsSieved[3] = {0};
371     UINT32  noPrimeFields[3] = {0};
372     UINT32  primesChecked[3] = {0};
373     UINT16  lastSievePrime = 0;
374     #endif
```

10.2.21 RsaKeyCache.c

10.2.21.1 Introduction

This file contains the functions to implement the RSA key cache that can be used to speed up simulation.

Only one key is created for each supported key size and it is returned whenever a key of that size is requested.

If desired, the key cache can be populated from a file. This allows multiple TPM to run with the same RSA keys. Also, when doing simulation, the DRBG will use preset sequences so it is not too hard to repeat sequences for debug or profile or stress.

When the key cache is enabled, a call to CryptRsaGenerateKey() will call the GetCachedRsaKey(). If the cache is enabled and populated, then the cached key of the requested size is returned. If a key of the requested size is not available, the no key is loaded and the requested key will need to be generated. If the cache is not populated, the TPM will open a file that has the appropriate name for the type of keys required (CRT or no-CRT). If the file is the right size, it is used. If the file doesn't exist or the file does not have the correct size, the TMP will populate the cache with new keys of the required size and write the cache data to the file so that they will be available the next time.

Currently, if two simulations are being run with TPM's that have different RSA key sizes (e.g,, one with 1024 and 2048 and another with 2048 and 3072, then the files will not match for the both of them and they will both try to overwrite the other's cache file. I may try to do something about this if necessary.

10.2.21.2 Includes, Types, Locals, and Defines

```
#include "Tpm.h"
#if defined SIMULATION && defined USE_RSA_KEY_CACHE
#include  <stdio.h>
#include "Platform_fp.h"
#include "RsaKeyCache_fp.h"
#if CRT_FORMAT_RSA == YES
#define CACHE_FILE_NAME "RsaKeyCacheCrt.data"
#else
#define CACHE_FILE_NAME "RsaKeyCacheNoCrt.data"
#endif
typedef struct _RSA_KEY_CACHE_
{
    TPM2B_PUBLIC_KEY_RSA        publicModulus;
    TPM2B_PRIVATE_KEY_RSA       privatePrime;
    privateExponent_t           privateExponent;
} RSA_KEY_CACHE;
```

Determine the number of RSA key sizes for the cache

```
#ifdef RSA_KEY_SIZE_BITS_1024
#define RSA_1024    YES
#else
#define RSA_1024    NO
#endif
#ifdef RSA_KEY_SIZE_BITS_2048
#define RSA_2048    YES
#else
#define RSA_2048    NO
#endif
#ifdef RSA_KEY_SIZE_BITS_3072
#define RSA_3072    YES
#else
#define RSA_3072    NO
#endif
#define comma
TPMI_RSA_KEY_BITS       SupportedRsaKeySizes[] = {
#if RSA_1024
    1024
#   undef comma
#   define comma ,
#endif
#if RSA_2048
    comma 2048
#   undef comma
#   define comma ,
#endif
#if RSA_3072
    comma 3072
#endif
};
#define RSA_KEY_CACHE_ENTRIES (RSA_1024 + RSA_2048 + RSA_3072)
```

The key cache holds one entry for each of the supported key sizes

```
RSA_KEY_CACHE        s_rsaKeyCache[RSA_KEY_CACHE_ENTRIES];
```

Indicates if the key cache is loaded. It can be loaded and enabled or disabled.

```
50  BOOL                s_keyCacheLoaded = 0;
```

Indicates if the key cache is enabled

```
51  int                 s_rsaKeyCacheEnabled = FALSE;
```

10.2.21.2.1 RsaKeyCacheControl()

Used to enable and disable the RSA key cache.

```
52  LIB_EXPORT void
53  RsaKeyCacheControl(
54      int             state
55      )
56  {
57      s_rsaKeyCacheEnabled = state;
58  }
```

10.2.21.2.2 InitializeKeyCache()

This will initialize the key cache and attempt to write it to a file for later use.

```
59  static BOOL
60  InitializeKeyCache(
61      OBJECT              *rsaKey,            // IN/OUT: The object structure in which
62                                              //          the key is created.
63      RAND_STATE          *rand               // IN: if not NULL, the deterministic
64                                              //     RNG state
65      )
66  {
67      int                  index;
68      TPM_KEY_BITS         keySave = rsaKey->publicArea.parameters.rsaDetail.keyBits;
69      const char          *fn = CACHE_FILE_NAME;
70      BOOL                 OK = TRUE;
71  //
72      s_rsaKeyCacheEnabled = FALSE;
73      for(index = 0; OK && index < RSA_KEY_CACHE_ENTRIES; index++)
74      {
75          rsaKey->publicArea.parameters.rsaDetail.keyBits
76              = SupportedRsaKeySizes[index];
77          OK = (CryptRsaGenerateKey(rsaKey, rand) == TPM_RC_SUCCESS);
78          if(OK)
79          {
80              s_rsaKeyCache[index].publicModulus = rsaKey->publicArea.unique.rsa;
81              s_rsaKeyCache[index].privatePrime = rsaKey->sensitive.sensitive.rsa;
82              s_rsaKeyCache[index].privateExponent = rsaKey->privateExponent;
83          }
84      }
85      rsaKey->publicArea.parameters.rsaDetail.keyBits = keySave;
86      s_keyCacheLoaded = OK;
87  #if defined SIMULATION && defined USE_RSA_KEY_CACHE && defined USE_KEY_CACHE_FILE
88      if(OK)
89      {
90          FILE                *cacheFile;
91  #if defined _MSC_VER && 1
92          if(fopen_s(&cacheFile, fn, "w+b") != 0)
93  #else
94          cacheFile = fopen(fn, "w+b");
95          if(NULL == cacheFile)
96  #endif
97          {
98              printf("Can't open %s for write.\n", fn);
```

```
 99          }
100          else
101          {
102              fseek(cacheFile, 0, SEEK_SET);
103              if(fwrite(s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
104                 != sizeof(s_rsaKeyCache))
105              {
106                  printf("Error writing cache to %s.", fn);
107              }
108          }
109          if(cacheFile)
110              fclose(cacheFile);
111      }
112  #endif
113      return s_keyCacheLoaded;
114  }
115  static BOOL
116  KeyCacheLoaded(
117      OBJECT              *rsaKey,            // IN/OUT: The object structure in which
118                                              //          the key is created.
119      RAND_STATE          *rand               // IN: if not NULL, the deterministic
120                                              //     RNG state
121      )
122  {
123  #if defined SIMULATION && defined USE_RSA_KEY_CACHE && defined USE_KEY_CACHE_FILE
124      if(!s_keyCacheLoaded)
125      {
126          FILE            *cacheFile;
127          const char *     fn = CACHE_FILE_NAME;
128  #if defined _MSC_VER && 1
129          if(fopen_s(&cacheFile, fn, "r+b") == 0)
130  #else
131          cacheFile = fopen(fn, "r+b");
132          if(NULL != cacheFile)
133  #endif
134          {
135              fseek(cacheFile, 0L, SEEK_END);
136              if(ftell(cacheFile) == sizeof(s_rsaKeyCache))
137              {
138                  fseek(cacheFile, 0L, SEEK_SET);
139                  s_keyCacheLoaded = (
140                      fread(&s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
141                      == sizeof(s_rsaKeyCache));
142              }
143              fclose(cacheFile);
144          }
145      }
146  #endif
147      if(!s_keyCacheLoaded)
148          s_rsaKeyCacheEnabled = InitializeKeyCache(rsaKey, rand);
149      return s_keyCacheLoaded;
150  }
151  BOOL
152  GetCachedRsaKey(
153      OBJECT              *key,
154      RAND_STATE          *rand               // IN: if not NULL, the deterministic
155                                              //     RNG state
156      )
157  {
158      int                     keyBits = key->publicArea.parameters.rsaDetail.keyBits;
159      int                     index;
160  //
161      if(KeyCacheLoaded(key, rand))
162      {
163          for(index = 0; index < RSA_KEY_CACHE_ENTRIES; index++)
164          {
```



```
165             if((s_rsaKeyCache[index].publicModulus.t.size * 8) == keyBits)
166             {
167                 key->publicArea.unique.rsa = s_rsaKeyCache[index].publicModulus;
168                 key->sensitive.sensitive.rsa = s_rsaKeyCache[index].privatePrime;
169                 key->privateExponent = s_rsaKeyCache[index].privateExponent;
170                 key->attributes.privateExp = SET;
171                 return TRUE;
172             }
173         }
174         return FALSE;
175     }
176     return s_keyCacheLoaded;
177 }
178 #endif  // defined SIMULATION && defined USE_RSA_KEY_CACHE
```

10.2.22 Ticket.c

10.2.22.1 Introduction

This clause contains the functions used for ticket computations.

10.2.22.2 Includes

```
1   #include "Tpm.h"
```

10.2.22.3 Functions

10.2.22.3.1 TicketIsSafe()

This function indicates if producing a ticket is safe. It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE or its substring of canonical form. If so, it is not safe to produce ticket for an input buffer claiming to be TPM generated buffer

Return Value	Meaning
TRUE	It is safe to produce ticket
FALSE	It is not safe to produce ticket

```
2   BOOL
3   TicketIsSafe(
4       TPM2B           *buffer
5       )
6   {
7       TPM_GENERATED   valueToCompare = TPM_GENERATED_VALUE;
8       BYTE            bufferToCompare[sizeof(valueToCompare)];
9       BYTE            *marshalBuffer;
10  //
11      // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
12      // it is not safe to generate a ticket
13      if(buffer->size < sizeof(valueToCompare))
14          return FALSE;
15      marshalBuffer = bufferToCompare;
16      TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);
17      if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
18          return FALSE;
19      else
20          return TRUE;
21  }
```

10.2.22.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```
22  void
23  TicketComputeVerified(
24      TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
25      TPM2B_DIGEST        *digest,         // IN: digest
26      TPM2B_NAME          *keyName,        // IN: name of key that signed the values
27      TPMT_TK_VERIFIED    *ticket          // OUT: verified ticket
28      )
29  {
30      TPM2B_AUTH          *proof;
31      HMAC_STATE           hmacState;
32  //
33      // Fill in ticket fields
34      ticket->tag = TPM_ST_VERIFIED;
35      ticket->hierarchy = hierarchy;
36      proof = HierarchyGetProof(hierarchy);
37      // Start HMAC using the proof value of the hierarchy as the HMAC key
38      ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
39                                               &proof->b);
40          //  TPM_ST_VERIFIED
41      CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
42          //  digest
43      CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
44          // key name
45      CryptDigestUpdate2B(&hmacState.hashState, &keyName->b);
46          // done
47      CryptHmacEnd2B(&hmacState, &ticket->digest.b);
48      return;
49  }
```

10.2.22.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```
50  void
51  TicketComputeAuth(
52      TPM_ST              type,           // IN: the type of ticket.
53      TPMI_RH_HIERARCHY   hierarchy,      // IN: hierarchy constant for ticket
54      UINT64              timeout,        // IN: timeout
55      BOOL                expiresOnReset,// IN: flag to indicate if ticket expires on
56                                          //      TPM Reset
57      TPM2B_DIGEST        *cpHashA,       // IN: input cpHashA
58      TPM2B_NONCE         *policyRef,     // IN: input policyRef
59      TPM2B_NAME          *entityName,    // IN: name of entity
60      TPMT_TK_AUTH        *ticket         // OUT: Created ticket
61      )
62  {
63      TPM2B_AUTH          *proof;
64      HMAC_STATE           hmacState;
65  //
66      // Get proper proof
67      proof = HierarchyGetProof(hierarchy);
68      // Fill in ticket fields
69      ticket->tag = type;
70      ticket->hierarchy = hierarchy;
71      // Start HMAC with hierarchy proof as the HMAC key
72      ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
73                                               &proof->b);
74          //  TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED,
75      CryptDigestUpdateInt(&hmacState, sizeof(UINT16), ticket->tag);
76      // cpHash
```

```
77      CryptDigestUpdate2B(&hmacState.hashState, &cpHashA->b);
78          //  policyRef
79      CryptDigestUpdate2B(&hmacState.hashState, &policyRef->b);
80          //  keyName
81      CryptDigestUpdate2B(&hmacState.hashState, &entityName->b);
82          //  timeout
83      CryptDigestUpdateInt(&hmacState, sizeof(timeout), timeout);
84      if(timeout != 0)
85      {
86              //  epoch
87          CryptDigestUpdateInt(&hmacState.hashState, sizeof(CLOCK_NONCE),
88                               g_timeEpoch);
89              // reset count
90          if(expiresOnReset)
91              CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
92                                   gp.totalResetCount);
93      }
94          // done
95      CryptHmacEnd2B(&hmacState, &ticket->digest.b);
96      return;
97  }
```

10.2.22.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```
98  void
99  TicketComputeHashCheck(
100     TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
101     TPM_ALG_ID           hashAlg,        // IN: the hash algorithm for 'digest'
102     TPM2B_DIGEST        *digest,         // IN: input digest
103     TPMT_TK_HASHCHECK   *ticket          // OUT: Created ticket
104     )
105 {
106     TPM2B_AUTH          *proof;
107     HMAC_STATE           hmacState;
108 //
109     // Get proper proof
110     proof = HierarchyGetProof(hierarchy);
111     // Fill in ticket fields
112     ticket->tag = TPM_ST_HASHCHECK;
113     ticket->hierarchy = hierarchy;
114     // Start HMAC using hierarchy proof as HMAC key
115     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
116                                              &proof->b);
117         //  TPM_ST_HASHCHECK
118     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
119         //  hash algorithm
120     CryptDigestUpdateInt(&hmacState, sizeof(hashAlg), hashAlg);
121         //  digest
122     CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
123         // done
124     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
125     return;
126 }
```

10.2.22.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```
127 void
128 TicketComputeCreation(
129     TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy for ticket
```

```
130         TPM2B_NAME          *name,          // IN: object name
131         TPM2B_DIGEST        *creation,      // IN: creation hash
132         TPMT_TK_CREATION    *ticket         // OUT: created ticket
133         )
134     {
135         TPM2B_AUTH          *proof;
136         HMAC_STATE           hmacState;
137         // Get proper proof
138         proof = HierarchyGetProof(hierarchy);
139         // Fill in ticket fields
140         ticket->tag = TPM_ST_CREATION;
141         ticket->hierarchy = hierarchy;
142         // Start HMAC using hierarchy proof as HMAC key
143         ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
144                                                  &proof->b);
145             //  TPM_ST_CREATION
146         CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
147             //  name if provided
148         if(name != NULL)
149             CryptDigestUpdate2B(&hmacState.hashState, &name->b);
150             //  creation hash
151         CryptDigestUpdate2B(&hmacState.hashState, &creation->b);
152             // Done
153         CryptHmacEnd2B(&hmacState, &ticket->digest.b);
154         return;
155     }
```


Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are used to define a TPM profile. Some of the values are derived from the values in the *TCG Algorithm Registry* [19] and others are simply chosen by the manufacturer. The values here are chosen to represent a full featured TPM so that all of the TPM's capabilities can be simulated and tested. This file would change based on the implementation.

NOTE The file listed below was generated by an automated tool using three documents as inputs. They are:

1) The TCG_Algorithm Registery,
2) Part 2 of this specification, and
3) A purpose-built document that contains vendor-specific information in tables.

A.2 Implementation.h

```
#ifndef _IMPLEMENTATION_H_
#define _IMPLEMENTATION_H_
#include    "TpmBuildSwitches.h"
#include    "BaseTypes.h"
#include    "TPMB.h"
#undef TRUE
#undef FALSE
```

This table is built in to TpmStructures() Change these definitions to turn all algorithms or commands on or off

```
#define      ALG_YES      YES
#define      ALG_NO       NO
#define      CC_YES       YES
#define      CC_NO        NO
```

From TPM 2.0 Part 2: Table 4 - Defines for Logic Values

```
#define  TRUE     1
#define  FALSE    0
#define  YES      1
#define  NO       0
#define  SET      1
#define  CLEAR    0
```

From Vendor-Specific: Table 1 - Defines for Processor Values

```
#define  BIG_ENDIAN_TPM       NO
#define  LITTLE_ENDIAN_TPM    YES
#define  AUTO_ALIGN           NO
```

From Vendor-Specific: Table 2 - Defines for Implemented Algorithms

```
#define  ALG_RSA              ALG_YES
#define  ALG_SHA1             ALG_YES
#define  ALG_HMAC             ALG_YES
#define  ALG_TDES             ALG_YES
#define  ALG_AES              ALG_YES
```

```
26  #define  ALG_MGF1              ALG_YES
27  #define  ALG_XOR               ALG_YES
28  #define  ALG_KEYEDHASH         ALG_YES
29  #define  ALG_SHA256            ALG_YES
30  #define  ALG_SHA384            ALG_YES
31  #define  ALG_SHA512            ALG_NO
32  #define  ALG_SM3_256           ALG_NO
33  #define  ALG_SM4               ALG_NO
34  #define  ALG_RSASSA            (ALG_YES*ALG_RSA)
35  #define  ALG_RSAES             (ALG_YES*ALG_RSA)
36  #define  ALG_RSAPSS            (ALG_YES*ALG_RSA)
37  #define  ALG_OAEP              (ALG_YES*ALG_RSA)
38  #define  ALG_ECC               ALG_YES
39  #define  ALG_ECDH              (ALG_YES*ALG_ECC)
40  #define  ALG_ECDSA             (ALG_YES*ALG_ECC)
41  #define  ALG_ECDAA             (ALG_YES*ALG_ECC)
42  #define  ALG_SM2               (ALG_NO*ALG_ECC)
43  #define  ALG_ECSCHNORR         (ALG_YES*ALG_ECC)
44  #define  ALG_ECMQV             (ALG_NO*ALG_ECC)
45  #define  ALG_SYMCIPHER         ALG_YES
46  #define  ALG_KDF1_SP800_56A    (ALG_YES*ALG_ECC)
47  #define  ALG_KDF2              ALG_NO
48  #define  ALG_KDF1_SP800_108    ALG_YES
49  #define  ALG_CTR               ALG_YES
50  #define  ALG_OFB               ALG_YES
51  #define  ALG_CBC               ALG_YES
52  #define  ALG_CFB               ALG_YES
53  #define  ALG_ECB               ALG_YES
```

From Vendor-Specific: Table 3 - Defines for Key Size Constants

```
54  #define  RSA_KEY_SIZES_BITS         {1024,2048}
55  #define  RSA_KEY_SIZE_BITS_1024     RSA_ALLOWED_KEY_SIZE_1024
56  #define  RSA_KEY_SIZE_BITS_2048     RSA_ALLOWED_KEY_SIZE_2048
57  #define  MAX_RSA_KEY_BITS           2048
58  #define  MAX_RSA_KEY_BYTES          256
59  #define  TDES_KEY_SIZES_BITS        {128,192}
60  #define  TDES_KEY_SIZE_BITS_128     TDES_ALLOWED_KEY_SIZE_128
61  #define  TDES_KEY_SIZE_BITS_192     TDES_ALLOWED_KEY_SIZE_192
62  #define  MAX_TDES_KEY_BITS          192
63  #define  MAX_TDES_KEY_BYTES         24
64  #define MAX_TDES_BLOCK_SIZE_BYTES                           \
65              MAX(TDES_128_BLOCK_SIZE_BYTES,                  \
66              MAX(TDES_192_BLOCK_SIZE_BYTES, 0))
67  #define  AES_KEY_SIZES_BITS         {128,256}
68  #define  AES_KEY_SIZE_BITS_128      AES_ALLOWED_KEY_SIZE_128
69  #define  AES_KEY_SIZE_BITS_256      AES_ALLOWED_KEY_SIZE_256
70  #define  MAX_AES_KEY_BITS           256
71  #define  MAX_AES_KEY_BYTES          32
72  #define MAX_AES_BLOCK_SIZE_BYTES                            \
73              MAX(AES_128_BLOCK_SIZE_BYTES,                   \
74              MAX(AES_256_BLOCK_SIZE_BYTES, 0))
75  #define  SM4_KEY_SIZES_BITS         {128}
76  #define  SM4_KEY_SIZE_BITS_128      SM4_ALLOWED_KEY_SIZE_128
77  #define  MAX_SM4_KEY_BITS           128
78  #define  MAX_SM4_KEY_BYTES          16
79  #define MAX_SM4_BLOCK_SIZE_BYTES                            \
80              MAX(SM4_128_BLOCK_SIZE_BYTES, 0)
81  #define  CAMELLIA_KEY_SIZES_BITS    {128}
82  #define  CAMELLIA_KEY_SIZE_BITS_128    CAMELLIA_ALLOWED_KEY_SIZE_128
83  #define  MAX_CAMELLIA_KEY_BITS      128
84  #define  MAX_CAMELLIA_KEY_BYTES     16
85  #define MAX_CAMELLIA_BLOCK_SIZE_BYTES                       \
86              MAX(CAMELLIA_128_BLOCK_SIZE_BYTES, 0)
```

From Vendor-Specific: Table 4 - Defines for Implemented Curves

```
 87  #define  ECC_NIST_P192          NO
 88  #define  ECC_NIST_P224          NO
 89  #define  ECC_NIST_P256          YES
 90  #define  ECC_NIST_P384          YES
 91  #define  ECC_NIST_P521          NO
 92  #define  ECC_BN_P256            YES
 93  #define  ECC_BN_P638            NO
 94  #define  ECC_SM2_P256           NO
 95  #define  ECC_CURVES             \
 96      {TPM_ECC_BN_P256, TPM_ECC_BN_P638, TPM_ECC_NIST_P192, TPM_ECC_NIST_P224, \
 97      TPM_ECC_NIST_P256, TPM_ECC_NIST_P384, TPM_ECC_NIST_P521, TPM_ECC_SM2_P256}
 98  #define  ECC_CURVE_COUNT        \
 99      (ECC_BN_P256 + ECC_BN_P638 + ECC_NIST_P192 + ECC_NIST_P224 + \
100      ECC_NIST_P256 + ECC_NIST_P384 + ECC_NIST_P521 + ECC_SM2_P256)
101  #define  MAX_ECC_KEY_BITS       \
102      MAX(ECC_BN_P256*256, MAX(ECC_BN_P638*638, \
103      MAX(ECC_NIST_P192*192, MAX(ECC_NIST_P224*224, \
104      MAX(ECC_NIST_P256*256, MAX(ECC_NIST_P384*384, \
105      MAX(ECC_NIST_P521*521, MAX(ECC_SM2_P256*256, \
106      0)))))))
107  #define  MAX_ECC_KEY_BYTES      BITS_TO_BYTES(MAX_ECC_KEY_BITS)
```

From Vendor-Specific: Table 5 - Defines for Implemented Commands

```
108  #define  CC_ActivateCredential          CC_YES
109  #define  CC_Certify                     CC_YES
110  #define  CC_CertifyCreation             CC_YES
111  #define  CC_ChangeEPS                   CC_YES
112  #define  CC_ChangePPS                   CC_YES
113  #define  CC_Clear                       CC_YES
114  #define  CC_ClearControl                CC_YES
115  #define  CC_ClockRateAdjust             CC_YES
116  #define  CC_ClockSet                    CC_YES
117  #define  CC_Commit                      (CC_YES*ALG_ECC)
118  #define  CC_ContextLoad                 CC_YES
119  #define  CC_ContextSave                 CC_YES
120  #define  CC_Create                      CC_YES
121  #define  CC_CreatePrimary               CC_YES
122  #define  CC_DictionaryAttackLockReset   CC_YES
123  #define  CC_DictionaryAttackParameters  CC_YES
124  #define  CC_Duplicate                   CC_YES
125  #define  CC_ECC_Parameters              (CC_YES*ALG_ECC)
126  #define  CC_ECDH_KeyGen                 (CC_YES*ALG_ECC)
127  #define  CC_ECDH_ZGen                   (CC_YES*ALG_ECC)
128  #define  CC_EncryptDecrypt              CC_YES
129  #define  CC_EventSequenceComplete       CC_YES
130  #define  CC_EvictControl                CC_YES
131  #define  CC_FieldUpgradeData            CC_NO
132  #define  CC_FieldUpgradeStart           CC_NO
133  #define  CC_FirmwareRead                CC_NO
134  #define  CC_FlushContext                CC_YES
135  #define  CC_GetCapability               CC_YES
136  #define  CC_GetCommandAuditDigest       CC_YES
137  #define  CC_GetRandom                   CC_YES
138  #define  CC_GetSessionAuditDigest       CC_YES
139  #define  CC_GetTestResult               CC_YES
140  #define  CC_GetTime                     CC_YES
141  #define  CC_Hash                        CC_YES
142  #define  CC_HashSequenceStart           CC_YES
143  #define  CC_HierarchyChangeAuth         CC_YES
144  #define  CC_HierarchyControl            CC_YES
145  #define  CC_HMAC                        CC_YES
146  #define  CC_HMAC_Start                  CC_YES
```

```
147   #define  CC_Import                      CC_YES
148   #define  CC_IncrementalSelfTest         CC_YES
149   #define  CC_Load                        CC_YES
150   #define  CC_LoadExternal                CC_YES
151   #define  CC_MakeCredential              CC_YES
152   #define  CC_NV_Certify                  CC_YES
153   #define  CC_NV_ChangeAuth               CC_YES
154   #define  CC_NV_DefineSpace              CC_YES
155   #define  CC_NV_Extend                   CC_YES
156   #define  CC_NV_GlobalWriteLock          CC_YES
157   #define  CC_NV_Increment                CC_YES
158   #define  CC_NV_Read                     CC_YES
159   #define  CC_NV_ReadLock                 CC_YES
160   #define  CC_NV_ReadPublic               CC_YES
161   #define  CC_NV_SetBits                  CC_YES
162   #define  CC_NV_UndefineSpace            CC_YES
163   #define  CC_NV_UndefineSpaceSpecial     CC_YES
164   #define  CC_NV_Write                    CC_YES
165   #define  CC_NV_WriteLock                CC_YES
166   #define  CC_ObjectChangeAuth            CC_YES
167   #define  CC_PCR_Allocate                CC_YES
168   #define  CC_PCR_Event                   CC_YES
169   #define  CC_PCR_Extend                  CC_YES
170   #define  CC_PCR_Read                    CC_YES
171   #define  CC_PCR_Reset                   CC_YES
172   #define  CC_PCR_SetAuthPolicy           CC_YES
173   #define  CC_PCR_SetAuthValue            CC_YES
174   #define  CC_PolicyAuthorize             CC_YES
175   #define  CC_PolicyAuthValue             CC_YES
176   #define  CC_PolicyCommandCode           CC_YES
177   #define  CC_PolicyCounterTimer          CC_YES
178   #define  CC_PolicyCpHash                CC_YES
179   #define  CC_PolicyDuplicationSelect     CC_YES
180   #define  CC_PolicyGetDigest             CC_YES
181   #define  CC_PolicyLocality              CC_YES
182   #define  CC_PolicyNameHash              CC_YES
183   #define  CC_PolicyNV                    CC_YES
184   #define  CC_PolicyOR                    CC_YES
185   #define  CC_PolicyPassword              CC_YES
186   #define  CC_PolicyPCR                   CC_YES
187   #define  CC_PolicyPhysicalPresence      CC_YES
188   #define  CC_PolicyRestart               CC_YES
189   #define  CC_PolicySecret                CC_YES
190   #define  CC_PolicySigned                CC_YES
191   #define  CC_PolicyTicket                CC_YES
192   #define  CC_PP_Commands                 CC_YES
193   #define  CC_Quote                       CC_YES
194   #define  CC_ReadClock                   CC_YES
195   #define  CC_ReadPublic                  CC_YES
196   #define  CC_Rewrap                      CC_YES
197   #define  CC_RSA_Decrypt                 (CC_YES*ALG_RSA)
198   #define  CC_RSA_Encrypt                 (CC_YES*ALG_RSA)
199   #define  CC_SelfTest                    CC_YES
200   #define  CC_SequenceComplete            CC_YES
201   #define  CC_SequenceUpdate              CC_YES
202   #define  CC_SetAlgorithmSet             CC_YES
203   #define  CC_SetCommandCodeAuditStatus   CC_YES
204   #define  CC_SetPrimaryPolicy            CC_YES
205   #define  CC_Shutdown                    CC_YES
206   #define  CC_Sign                        CC_YES
207   #define  CC_StartAuthSession            CC_YES
208   #define  CC_Startup                     CC_YES
209   #define  CC_StirRandom                  CC_YES
210   #define  CC_TestParms                   CC_YES
211   #define  CC_Unseal                      CC_YES
212   #define  CC_VerifySignature             CC_YES
```



```
213   #define  CC_ZGen_2Phase                (CC_YES*ALG_ECC)
214   #define  CC_EC_Ephemeral               (CC_YES*ALG_ECC)
215   #define  CC_PolicyNvWritten            CC_YES
216   #define  CC_PolicyTemplate             CC_YES
217   #define  CC_CreateLoaded               CC_YES
218   #define  CC_PolicyAuthorizeNV          CC_YES
219   #define  CC_EncryptDecrypt2            CC_YES
220   #define  CC_Vendor_TCG_Test            CC_YES
```

From Vendor-Specific: Table 6 - Defines for PLATFORM Values

```
221   #define  PLATFORM_FAMILY         TPM_SPEC_FAMILY
222   #define  PLATFORM_LEVEL          TPM_SPEC_LEVEL
223   #define  PLATFORM_VERSION        TPM_SPEC_VERSION
224   #define  PLATFORM_YEAR           TPM_SPEC_YEAR
225   #define  PLATFORM_DAY_OF_YEAR    TPM_SPEC_DAY_OF_YEAR
```

From Vendor-Specific: Table 7 - Defines for Implementation Values

```
226   #define  FIELD_UPGRADE_IMPLEMENTED      NO
227   #define  RADIX_BITS                     32
228   #define  HASH_ALIGNMENT                 4
229   #define  SYMMETRIC_ALIGNMENT            4
230   #define  HASH_LIB                       OSSL
231   #define  SYM_LIB                        OSSL
232   #define  MATH_LIB                       OSSL
233   #define  BSIZE                          UINT16
234   #define  IMPLEMENTATION_PCR             24
235   #define  PLATFORM_PCR                   24
236   #define  DRTM_PCR                       17
237   #define  HCRTM_PCR                      0
238   #define  NUM_LOCALITIES                 5
239   #define  MAX_HANDLE_NUM                 3
240   #define  MAX_ACTIVE_SESSIONS            64
241   #define  CONTEXT_SLOT                   UINT16
242   #define  CONTEXT_COUNTER                UINT64
243   #define  MAX_LOADED_SESSIONS            3
244   #define  MAX_SESSION_NUM                3
245   #define  MAX_LOADED_OBJECTS             3
246   #define  MIN_EVICT_OBJECTS              2
247   #define  NUM_POLICY_PCR_GROUP           1
248   #define  NUM_AUTHVALUE_PCR_GROUP        1
249   #define  MAX_CONTEXT_SIZE               2474
250   #define  MAX_DIGEST_BUFFER              1024
251   #define  MAX_NV_INDEX_SIZE              2048
252   #define  MAX_NV_BUFFER_SIZE             1024
253   #define  MAX_CAP_BUFFER                 1024
254   #define  NV_MEMORY_SIZE                 16384
255   #define  MIN_COUNTER_INDICES            8
256   #define  NUM_STATIC_PCR                 16
257   #define  MAX_ALG_LIST_SIZE              64
258   #define  PRIMARY_SEED_SIZE              32
259   #define  CONTEXT_ENCRYPT_ALGORITHM      AES
260   #define  NV_CLOCK_UPDATE_INTERVAL       12
261   #define  NUM_POLICY_PCR                 1
262   #define  MAX_COMMAND_SIZE               4096
263   #define  MAX_RESPONSE_SIZE              4096
264   #define  ORDERLY_BITS                   8
265   #define  MAX_SYM_DATA                   128
266   #define  MAX_RNG_ENTROPY_SIZE           64
267   #define  RAM_INDEX_SPACE                512
268   #define  RSA_DEFAULT_PUBLIC_EXPONENT    0x00010001
269   #define  ENABLE_PCR_NO_INCREMENT        YES
270   #define  CRT_FORMAT_RSA                 YES
271   #define  VENDOR_COMMAND_COUNT           0
```



```
#define  MAX_VENDOR_BUFFER_SIZE         1024
```

From TCG Algorithm Registry: Table 2 - Definition of TPM_ALG_ID Constants

```
typedef  UINT16             TPM_ALG_ID;
#define  ALG_ERROR_VALUE             0x0000
#define  TPM_ALG_ERROR               (TPM_ALG_ID)(ALG_ERROR_VALUE)
#define  ALG_RSA_VALUE               0x0001
#if defined ALG_RSA && ALG_RSA == YES
#define  TPM_ALG_RSA                 (TPM_ALG_ID)(ALG_RSA_VALUE)
#endif
#define  ALG_TDES_VALUE              0x0003
#if defined ALG_TDES && ALG_TDES == YES
#define  TPM_ALG_TDES                (TPM_ALG_ID)(ALG_TDES_VALUE)
#endif
#define  ALG_SHA_VALUE               0x0004
#if defined ALG_SHA && ALG_SHA == YES
#define  TPM_ALG_SHA                 (TPM_ALG_ID)(ALG_SHA_VALUE)
#endif
#define  ALG_SHA1_VALUE              0x0004
#if defined ALG_SHA1 && ALG_SHA1 == YES
#define  TPM_ALG_SHA1                (TPM_ALG_ID)(ALG_SHA1_VALUE)
#endif
#define  ALG_HMAC_VALUE              0x0005
#if defined ALG_HMAC && ALG_HMAC == YES
#define  TPM_ALG_HMAC                (TPM_ALG_ID)(ALG_HMAC_VALUE)
#endif
#define  ALG_AES_VALUE               0x0006
#if defined ALG_AES && ALG_AES == YES
#define  TPM_ALG_AES                 (TPM_ALG_ID)(ALG_AES_VALUE)
#endif
#define  ALG_MGF1_VALUE              0x0007
#if defined ALG_MGF1 && ALG_MGF1 == YES
#define  TPM_ALG_MGF1                (TPM_ALG_ID)(ALG_MGF1_VALUE)
#endif
#define  ALG_KEYEDHASH_VALUE         0x0008
#if defined ALG_KEYEDHASH && ALG_KEYEDHASH == YES
#define  TPM_ALG_KEYEDHASH           (TPM_ALG_ID)(ALG_KEYEDHASH_VALUE)
#endif
#define  ALG_XOR_VALUE               0x000A
#if defined ALG_XOR && ALG_XOR == YES
#define  TPM_ALG_XOR                 (TPM_ALG_ID)(ALG_XOR_VALUE)
#endif
#define  ALG_SHA256_VALUE            0x000B
#if defined ALG_SHA256 && ALG_SHA256 == YES
#define  TPM_ALG_SHA256              (TPM_ALG_ID)(ALG_SHA256_VALUE)
#endif
#define  ALG_SHA384_VALUE            0x000C
#if defined ALG_SHA384 && ALG_SHA384 == YES
#define  TPM_ALG_SHA384              (TPM_ALG_ID)(ALG_SHA384_VALUE)
#endif
#define  ALG_SHA512_VALUE            0x000D
#if defined ALG_SHA512 && ALG_SHA512 == YES
#define  TPM_ALG_SHA512              (TPM_ALG_ID)(ALG_SHA512_VALUE)
#endif
#define  ALG_NULL_VALUE              0x0010
#define  TPM_ALG_NULL                (TPM_ALG_ID)(ALG_NULL_VALUE)
#define  ALG_SM3_256_VALUE           0x0012
#if defined ALG_SM3_256 && ALG_SM3_256 == YES
#define  TPM_ALG_SM3_256             (TPM_ALG_ID)(ALG_SM3_256_VALUE)
#endif
#define  ALG_SM4_VALUE               0x0013
#if defined ALG_SM4 && ALG_SM4 == YES
#define  TPM_ALG_SM4                 (TPM_ALG_ID)(ALG_SM4_VALUE)
#endif
```

```
334   #define  ALG_RSASSA_VALUE            0x0014
335   #if defined ALG_RSASSA && ALG_RSASSA == YES
336   #define  TPM_ALG_RSASSA              (TPM_ALG_ID)(ALG_RSASSA_VALUE)
337   #endif
338   #define  ALG_RSAES_VALUE             0x0015
339   #if defined ALG_RSAES && ALG_RSAES == YES
340   #define  TPM_ALG_RSAES               (TPM_ALG_ID)(ALG_RSAES_VALUE)
341   #endif
342   #define  ALG_RSAPSS_VALUE            0x0016
343   #if defined ALG_RSAPSS && ALG_RSAPSS == YES
344   #define  TPM_ALG_RSAPSS              (TPM_ALG_ID)(ALG_RSAPSS_VALUE)
345   #endif
346   #define  ALG_OAEP_VALUE              0x0017
347   #if defined ALG_OAEP && ALG_OAEP == YES
348   #define  TPM_ALG_OAEP                (TPM_ALG_ID)(ALG_OAEP_VALUE)
349   #endif
350   #define  ALG_ECDSA_VALUE             0x0018
351   #if defined ALG_ECDSA && ALG_ECDSA == YES
352   #define  TPM_ALG_ECDSA               (TPM_ALG_ID)(ALG_ECDSA_VALUE)
353   #endif
354   #define  ALG_ECDH_VALUE              0x0019
355   #if defined ALG_ECDH && ALG_ECDH == YES
356   #define  TPM_ALG_ECDH                (TPM_ALG_ID)(ALG_ECDH_VALUE)
357   #endif
358   #define  ALG_ECDAA_VALUE             0x001A
359   #if defined ALG_ECDAA && ALG_ECDAA == YES
360   #define  TPM_ALG_ECDAA               (TPM_ALG_ID)(ALG_ECDAA_VALUE)
361   #endif
362   #define  ALG_SM2_VALUE               0x001B
363   #if defined ALG_SM2 && ALG_SM2 == YES
364   #define  TPM_ALG_SM2                 (TPM_ALG_ID)(ALG_SM2_VALUE)
365   #endif
366   #define  ALG_ECSCHNORR_VALUE         0x001C
367   #if defined ALG_ECSCHNORR && ALG_ECSCHNORR == YES
368   #define  TPM_ALG_ECSCHNORR           (TPM_ALG_ID)(ALG_ECSCHNORR_VALUE)
369   #endif
370   #define  ALG_ECMQV_VALUE             0x001D
371   #if defined ALG_ECMQV && ALG_ECMQV == YES
372   #define  TPM_ALG_ECMQV               (TPM_ALG_ID)(ALG_ECMQV_VALUE)
373   #endif
374   #define  ALG_KDF1_SP800_56A_VALUE    0x0020
375   #if defined ALG_KDF1_SP800_56A && ALG_KDF1_SP800_56A == YES
376   #define  TPM_ALG_KDF1_SP800_56A      (TPM_ALG_ID)(ALG_KDF1_SP800_56A_VALUE)
377   #endif
378   #define  ALG_KDF2_VALUE              0x0021
379   #if defined ALG_KDF2 && ALG_KDF2 == YES
380   #define  TPM_ALG_KDF2                (TPM_ALG_ID)(ALG_KDF2_VALUE)
381   #endif
382   #define  ALG_KDF1_SP800_108_VALUE    0x0022
383   #if defined ALG_KDF1_SP800_108 && ALG_KDF1_SP800_108 == YES
384   #define  TPM_ALG_KDF1_SP800_108      (TPM_ALG_ID)(ALG_KDF1_SP800_108_VALUE)
385   #endif
386   #define  ALG_ECC_VALUE               0x0023
387   #if defined ALG_ECC && ALG_ECC == YES
388   #define  TPM_ALG_ECC                 (TPM_ALG_ID)(ALG_ECC_VALUE)
389   #endif
390   #define  ALG_SYMCIPHER_VALUE         0x0025
391   #if defined ALG_SYMCIPHER && ALG_SYMCIPHER == YES
392   #define  TPM_ALG_SYMCIPHER           (TPM_ALG_ID)(ALG_SYMCIPHER_VALUE)
393   #endif
394   #define  ALG_CAMELLIA_VALUE          0x0026
395   #if defined ALG_CAMELLIA && ALG_CAMELLIA == YES
396   #define  TPM_ALG_CAMELLIA            (TPM_ALG_ID)(ALG_CAMELLIA_VALUE)
397   #endif
398   #define  ALG_CTR_VALUE               0x0040
399   #if defined ALG_CTR && ALG_CTR == YES
```



```
400   #define  TPM_ALG_CTR                (TPM_ALG_ID)(ALG_CTR_VALUE)
401   #endif
402   #define  ALG_OFB_VALUE              0x0041
403   #if defined ALG_OFB && ALG_OFB == YES
404   #define  TPM_ALG_OFB                (TPM_ALG_ID)(ALG_OFB_VALUE)
405   #endif
406   #define  ALG_CBC_VALUE              0x0042
407   #if defined ALG_CBC && ALG_CBC == YES
408   #define  TPM_ALG_CBC                (TPM_ALG_ID)(ALG_CBC_VALUE)
409   #endif
410   #define  ALG_CFB_VALUE              0x0043
411   #if defined ALG_CFB && ALG_CFB == YES
412   #define  TPM_ALG_CFB                (TPM_ALG_ID)(ALG_CFB_VALUE)
413   #endif
414   #define  ALG_ECB_VALUE              0x0044
415   #if defined ALG_ECB && ALG_ECB == YES
416   #define  TPM_ALG_ECB                (TPM_ALG_ID)(ALG_ECB_VALUE)
417   #endif
418   #define  TPM_ALG_FIRST              (TPM_ALG_ID)(0x0001)
419   #define  ALG_FIRST_VALUE            0x0001
420   #define  TPM_ALG_LAST               (TPM_ALG_ID)(0x0044)
421   #define  ALG_LAST_VALUE             0x0044
```

From TCG Algorithm Registry: Table 3 - Definition of TPM_ECC_CURVE Constants

```
422   typedef  UINT16             TPM_ECC_CURVE;
423   #define  TPM_ECC_NONE          (TPM_ECC_CURVE)(0x0000)
424   #define  TPM_ECC_NIST_P192     (TPM_ECC_CURVE)(0x0001)
425   #define  TPM_ECC_NIST_P224     (TPM_ECC_CURVE)(0x0002)
426   #define  TPM_ECC_NIST_P256     (TPM_ECC_CURVE)(0x0003)
427   #define  TPM_ECC_NIST_P384     (TPM_ECC_CURVE)(0x0004)
428   #define  TPM_ECC_NIST_P521     (TPM_ECC_CURVE)(0x0005)
429   #define  TPM_ECC_BN_P256       (TPM_ECC_CURVE)(0x0010)
430   #define  TPM_ECC_BN_P638       (TPM_ECC_CURVE)(0x0011)
431   #define  TPM_ECC_SM2_P256      (TPM_ECC_CURVE)(0x0020)
```

From TCG Algorithm Registry: Table 4 - Defines for NIST_P192 ECC Values Data is in CryptEccData.c
From TCG Algorithm Registry: Table 5 - Defines for NIST_P224 ECC Values Data is in CryptEccData.c
From TCG Algorithm Registry: Table 6 - Defines for NIST_P256 ECC Values Data is in CryptEccData.c
From TCG Algorithm Registry: Table 7 - Defines for NIST_P384 ECC Values Data is in CryptEccData.c
From TCG Algorithm Registry: Table 8 - Defines for NIST_P521 ECC Values Data is in CryptEccData.c
From TCG Algorithm Registry: Table 9 - Defines for BN_P256 ECC Values Data is in CryptEccData.c
From TCG Algorithm Registry: Table 10 - Defines for BN_P638 ECC Values Data is in CryptEccData.c
From TCG Algorithm Registry: Table 11 - Defines for SM2_P256 ECC Values Data is in CryptEccData.c
From TCG Algorithm Registry: Table 12 - Defines for SHA1 Hash Values

```
432   #define  SHA1_DIGEST_SIZE    20
433   #define  SHA1_BLOCK_SIZE     64
434   #define  SHA1_DER_SIZE       15
435   #define  SHA1_DER            \
436       0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x0E, \
437       0x03, 0x02, 0x1A, 0x05, 0x00, 0x04, 0x14
```

From TCG Algorithm Registry: Table 13 - Defines for SHA256 Hash Values

```
438   #define  SHA256_DIGEST_SIZE    32
439   #define  SHA256_BLOCK_SIZE     64
440   #define  SHA256_DER_SIZE       19
441   #define  SHA256_DER            \
442       0x30, 0x31, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, \
443       0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01, 0x05, \
444       0x00, 0x04, 0x20
```


From TCG Algorithm Registry: Table 14 - Defines for SHA384 Hash Values

```
445   #define  SHA384_DIGEST_SIZE    48
446   #define  SHA384_BLOCK_SIZE     128
447   #define  SHA384_DER_SIZE       19
448   #define  SHA384_DER            \
449       0x30, 0x41, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, \
450       0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02, 0x05, \
451       0x00, 0x04, 0x30
```

From TCG Algorithm Registry: Table 15 - Defines for SHA512 Hash Values

```
452   #define  SHA512_DIGEST_SIZE    64
453   #define  SHA512_BLOCK_SIZE     128
454   #define  SHA512_DER_SIZE       19
455   #define  SHA512_DER            \
456       0x30, 0x51, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, \
457       0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03, 0x05, \
458       0x00, 0x04, 0x40
```

From TCG Algorithm Registry: Table 16 - Defines for SM3_256 Hash Values

```
459   #define  SM3_256_DIGEST_SIZE    32
460   #define  SM3_256_BLOCK_SIZE     64
461   #define  SM3_256_DER_SIZE       18
462   #define  SM3_256_DER            \
463       0x30, 0x30, 0x30, 0x0C, 0x06, 0x08, 0x2A, 0x81, \
464       0x1C, 0x81, 0x45, 0x01, 0x83, 0x11, 0x05, 0x00, \
465       0x04, 0x20
```

From TCG Algorithm Registry: Table 17 - Defines for AES Symmetric Cipher Algorithm Constants

```
466   #define  AES_ALLOWED_KEY_SIZE_128    YES
467   #define  AES_ALLOWED_KEY_SIZE_192    YES
468   #define  AES_ALLOWED_KEY_SIZE_256    YES
469   #define  AES_128_BLOCK_SIZE_BYTES    16
470   #define  AES_192_BLOCK_SIZE_BYTES    16
471   #define  AES_256_BLOCK_SIZE_BYTES    16
```

From TCG Algorithm Registry: Table 18 - Defines for SM4 Symmetric Cipher Algorithm Constants

```
472   #define  SM4_ALLOWED_KEY_SIZE_128    YES
473   #define  SM4_128_BLOCK_SIZE_BYTES    16
```

From TCG Algorithm Registry: Table 19 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```
474   #define  CAMELLIA_ALLOWED_KEY_SIZE_128    YES
475   #define  CAMELLIA_ALLOWED_KEY_SIZE_192    YES
476   #define  CAMELLIA_ALLOWED_KEY_SIZE_256    YES
477   #define  CAMELLIA_128_BLOCK_SIZE_BYTES    16
478   #define  CAMELLIA_192_BLOCK_SIZE_BYTES    16
479   #define  CAMELLIA_256_BLOCK_SIZE_BYTES    16
```

From TCG Algorithm Registry: Table 17 - Defines for TDES Symmetric Cipher Algorithm Constants

```
480   #define  TDES_ALLOWED_KEY_SIZE_128    YES
481   #define  TDES_ALLOWED_KEY_SIZE_192    YES
482   #define  TDES_128_BLOCK_SIZE_BYTES    8
483   #define  TDES_192_BLOCK_SIZE_BYTES    8
```

From TPM 2.0 Part 2: Table 12 - Definition of TPM_CC Constants

```
484   typedef  UINT32            TPM_CC;
485   #ifndef CC_NV_UndefineSpaceSpecial
486   #   define CC_NV_UndefineSpaceSpecial NO
487   #endif
488   #if CC_NV_UndefineSpaceSpecial == YES
489   #define  TPM_CC_NV_UndefineSpaceSpecial       (TPM_CC)(0x0000011f)
490   #endif
491   #ifndef CC_EvictControl
492   #   define CC_EvictControl NO
493   #endif
494   #if CC_EvictControl == YES
495   #define  TPM_CC_EvictControl                  (TPM_CC)(0x00000120)
496   #endif
497   #ifndef CC_HierarchyControl
498   #   define CC_HierarchyControl NO
499   #endif
500   #if CC_HierarchyControl == YES
501   #define  TPM_CC_HierarchyControl              (TPM_CC)(0x00000121)
502   #endif
503   #ifndef CC_NV_UndefineSpace
504   #   define CC_NV_UndefineSpace NO
505   #endif
506   #if CC_NV_UndefineSpace == YES
507   #define  TPM_CC_NV_UndefineSpace              (TPM_CC)(0x00000122)
508   #endif
509   #ifndef CC_ChangeEPS
510   #   define CC_ChangeEPS NO
511   #endif
512   #if CC_ChangeEPS == YES
513   #define  TPM_CC_ChangeEPS                     (TPM_CC)(0x00000124)
514   #endif
515   #ifndef CC_ChangePPS
516   #   define CC_ChangePPS NO
517   #endif
518   #if CC_ChangePPS == YES
519   #define  TPM_CC_ChangePPS                     (TPM_CC)(0x00000125)
520   #endif
521   #ifndef CC_Clear
522   #   define CC_Clear NO
523   #endif
524   #if CC_Clear == YES
525   #define  TPM_CC_Clear                         (TPM_CC)(0x00000126)
526   #endif
527   #ifndef CC_ClearControl
528   #   define CC_ClearControl NO
529   #endif
530   #if CC_ClearControl == YES
531   #define  TPM_CC_ClearControl                  (TPM_CC)(0x00000127)
532   #endif
533   #ifndef CC_ClockSet
534   #   define CC_ClockSet NO
535   #endif
536   #if CC_ClockSet == YES
537   #define  TPM_CC_ClockSet                      (TPM_CC)(0x00000128)
538   #endif
539   #ifndef CC_HierarchyChangeAuth
540   #   define CC_HierarchyChangeAuth NO
541   #endif
542   #if CC_HierarchyChangeAuth == YES
543   #define  TPM_CC_HierarchyChangeAuth           (TPM_CC)(0x00000129)
544   #endif
545   #ifndef CC_NV_DefineSpace
546   #   define CC_NV_DefineSpace NO
547   #endif
548   #if CC_NV_DefineSpace == YES
549   #define  TPM_CC_NV_DefineSpace                (TPM_CC)(0x0000012a)
```

```
550   #endif
551   #ifndef CC_PCR_Allocate
552   #   define CC_PCR_Allocate NO
553   #endif
554   #if CC_PCR_Allocate == YES
555   #define  TPM_CC_PCR_Allocate                  (TPM_CC)(0x0000012b)
556   #endif
557   #ifndef CC_PCR_SetAuthPolicy
558   #   define CC_PCR_SetAuthPolicy NO
559   #endif
560   #if CC_PCR_SetAuthPolicy == YES
561   #define  TPM_CC_PCR_SetAuthPolicy             (TPM_CC)(0x0000012c)
562   #endif
563   #ifndef CC_PP_Commands
564   #   define CC_PP_Commands NO
565   #endif
566   #if CC_PP_Commands == YES
567   #define  TPM_CC_PP_Commands                   (TPM_CC)(0x0000012d)
568   #endif
569   #ifndef CC_SetPrimaryPolicy
570   #   define CC_SetPrimaryPolicy NO
571   #endif
572   #if CC_SetPrimaryPolicy == YES
573   #define  TPM_CC_SetPrimaryPolicy              (TPM_CC)(0x0000012e)
574   #endif
575   #ifndef CC_FieldUpgradeStart
576   #   define CC_FieldUpgradeStart NO
577   #endif
578   #if CC_FieldUpgradeStart == YES
579   #define  TPM_CC_FieldUpgradeStart             (TPM_CC)(0x0000012f)
580   #endif
581   #ifndef CC_ClockRateAdjust
582   #   define CC_ClockRateAdjust NO
583   #endif
584   #if CC_ClockRateAdjust == YES
585   #define  TPM_CC_ClockRateAdjust               (TPM_CC)(0x00000130)
586   #endif
587   #ifndef CC_CreatePrimary
588   #   define CC_CreatePrimary NO
589   #endif
590   #if CC_CreatePrimary == YES
591   #define  TPM_CC_CreatePrimary                 (TPM_CC)(0x00000131)
592   #endif
593   #ifndef CC_NV_GlobalWriteLock
594   #   define CC_NV_GlobalWriteLock NO
595   #endif
596   #if CC_NV_GlobalWriteLock == YES
597   #define  TPM_CC_NV_GlobalWriteLock            (TPM_CC)(0x00000132)
598   #endif
599   #ifndef CC_GetCommandAuditDigest
600   #   define CC_GetCommandAuditDigest NO
601   #endif
602   #if CC_GetCommandAuditDigest == YES
603   #define  TPM_CC_GetCommandAuditDigest         (TPM_CC)(0x00000133)
604   #endif
605   #ifndef CC_NV_Increment
606   #   define CC_NV_Increment NO
607   #endif
608   #if CC_NV_Increment == YES
609   #define  TPM_CC_NV_Increment                  (TPM_CC)(0x00000134)
610   #endif
611   #ifndef CC_NV_SetBits
612   #   define CC_NV_SetBits NO
613   #endif
614   #if CC_NV_SetBits == YES
615   #define  TPM_CC_NV_SetBits                    (TPM_CC)(0x00000135)
```



```
616   #endif
617   #ifndef CC_NV_Extend
618   #   define CC_NV_Extend NO
619   #endif
620   #if CC_NV_Extend == YES
621   #define  TPM_CC_NV_Extend                    (TPM_CC)(0x00000136)
622   #endif
623   #ifndef CC_NV_Write
624   #   define CC_NV_Write NO
625   #endif
626   #if CC_NV_Write == YES
627   #define  TPM_CC_NV_Write                     (TPM_CC)(0x00000137)
628   #endif
629   #ifndef CC_NV_WriteLock
630   #   define CC_NV_WriteLock NO
631   #endif
632   #if CC_NV_WriteLock == YES
633   #define  TPM_CC_NV_WriteLock                 (TPM_CC)(0x00000138)
634   #endif
635   #ifndef CC_DictionaryAttackLockReset
636   #   define CC_DictionaryAttackLockReset NO
637   #endif
638   #if CC_DictionaryAttackLockReset == YES
639   #define  TPM_CC_DictionaryAttackLockReset    (TPM_CC)(0x00000139)
640   #endif
641   #ifndef CC_DictionaryAttackParameters
642   #   define CC_DictionaryAttackParameters NO
643   #endif
644   #if CC_DictionaryAttackParameters == YES
645   #define  TPM_CC_DictionaryAttackParameters   (TPM_CC)(0x0000013a)
646   #endif
647   #ifndef CC_NV_ChangeAuth
648   #   define CC_NV_ChangeAuth NO
649   #endif
650   #if CC_NV_ChangeAuth == YES
651   #define  TPM_CC_NV_ChangeAuth                (TPM_CC)(0x0000013b)
652   #endif
653   #ifndef CC_PCR_Event
654   #   define CC_PCR_Event NO
655   #endif
656   #if CC_PCR_Event == YES
657   #define  TPM_CC_PCR_Event                    (TPM_CC)(0x0000013c)
658   #endif
659   #ifndef CC_PCR_Reset
660   #   define CC_PCR_Reset NO
661   #endif
662   #if CC_PCR_Reset == YES
663   #define  TPM_CC_PCR_Reset                    (TPM_CC)(0x0000013d)
664   #endif
665   #ifndef CC_SequenceComplete
666   #   define CC_SequenceComplete NO
667   #endif
668   #if CC_SequenceComplete == YES
669   #define  TPM_CC_SequenceComplete             (TPM_CC)(0x0000013e)
670   #endif
671   #ifndef CC_SetAlgorithmSet
672   #   define CC_SetAlgorithmSet NO
673   #endif
674   #if CC_SetAlgorithmSet == YES
675   #define  TPM_CC_SetAlgorithmSet              (TPM_CC)(0x0000013f)
676   #endif
677   #ifndef CC_SetCommandCodeAuditStatus
678   #   define CC_SetCommandCodeAuditStatus NO
679   #endif
680   #if CC_SetCommandCodeAuditStatus == YES
681   #define  TPM_CC_SetCommandCodeAuditStatus    (TPM_CC)(0x00000140)
```



```
682   #endif
683   #ifndef CC_FieldUpgradeData
684   #   define CC_FieldUpgradeData NO
685   #endif
686   #if CC_FieldUpgradeData == YES
687   #define  TPM_CC_FieldUpgradeData              (TPM_CC)(0x00000141)
688   #endif
689   #ifndef CC_IncrementalSelfTest
690   #   define CC_IncrementalSelfTest NO
691   #endif
692   #if CC_IncrementalSelfTest == YES
693   #define  TPM_CC_IncrementalSelfTest           (TPM_CC)(0x00000142)
694   #endif
695   #ifndef CC_SelfTest
696   #   define CC_SelfTest NO
697   #endif
698   #if CC_SelfTest == YES
699   #define  TPM_CC_SelfTest                      (TPM_CC)(0x00000143)
700   #endif
701   #ifndef CC_Startup
702   #   define CC_Startup NO
703   #endif
704   #if CC_Startup == YES
705   #define  TPM_CC_Startup                       (TPM_CC)(0x00000144)
706   #endif
707   #ifndef CC_Shutdown
708   #   define CC_Shutdown NO
709   #endif
710   #if CC_Shutdown == YES
711   #define  TPM_CC_Shutdown                      (TPM_CC)(0x00000145)
712   #endif
713   #ifndef CC_StirRandom
714   #   define CC_StirRandom NO
715   #endif
716   #if CC_StirRandom == YES
717   #define  TPM_CC_StirRandom                    (TPM_CC)(0x00000146)
718   #endif
719   #ifndef CC_ActivateCredential
720   #   define CC_ActivateCredential NO
721   #endif
722   #if CC_ActivateCredential == YES
723   #define  TPM_CC_ActivateCredential            (TPM_CC)(0x00000147)
724   #endif
725   #ifndef CC_Certify
726   #   define CC_Certify NO
727   #endif
728   #if CC_Certify == YES
729   #define  TPM_CC_Certify                       (TPM_CC)(0x00000148)
730   #endif
731   #ifndef CC_PolicyNV
732   #   define CC_PolicyNV NO
733   #endif
734   #if CC_PolicyNV == YES
735   #define  TPM_CC_PolicyNV                      (TPM_CC)(0x00000149)
736   #endif
737   #ifndef CC_CertifyCreation
738   #   define CC_CertifyCreation NO
739   #endif
740   #if CC_CertifyCreation == YES
741   #define  TPM_CC_CertifyCreation               (TPM_CC)(0x0000014a)
742   #endif
743   #ifndef CC_Duplicate
744   #   define CC_Duplicate NO
745   #endif
746   #if CC_Duplicate == YES
747   #define  TPM_CC_Duplicate                     (TPM_CC)(0x0000014b)
```

```
748   #endif
749   #ifndef CC_GetTime
750   #   define CC_GetTime NO
751   #endif
752   #if CC_GetTime == YES
753   #define  TPM_CC_GetTime                       (TPM_CC)(0x0000014c)
754   #endif
755   #ifndef CC_GetSessionAuditDigest
756   #   define CC_GetSessionAuditDigest NO
757   #endif
758   #if CC_GetSessionAuditDigest == YES
759   #define  TPM_CC_GetSessionAuditDigest         (TPM_CC)(0x0000014d)
760   #endif
761   #ifndef CC_NV_Read
762   #   define CC_NV_Read NO
763   #endif
764   #if CC_NV_Read == YES
765   #define  TPM_CC_NV_Read                       (TPM_CC)(0x0000014e)
766   #endif
767   #ifndef CC_NV_ReadLock
768   #   define CC_NV_ReadLock NO
769   #endif
770   #if CC_NV_ReadLock == YES
771   #define  TPM_CC_NV_ReadLock                   (TPM_CC)(0x0000014f)
772   #endif
773   #ifndef CC_ObjectChangeAuth
774   #   define CC_ObjectChangeAuth NO
775   #endif
776   #if CC_ObjectChangeAuth == YES
777   #define  TPM_CC_ObjectChangeAuth              (TPM_CC)(0x00000150)
778   #endif
779   #ifndef CC_PolicySecret
780   #   define CC_PolicySecret NO
781   #endif
782   #if CC_PolicySecret == YES
783   #define  TPM_CC_PolicySecret                  (TPM_CC)(0x00000151)
784   #endif
785   #ifndef CC_Rewrap
786   #   define CC_Rewrap NO
787   #endif
788   #if CC_Rewrap == YES
789   #define  TPM_CC_Rewrap                        (TPM_CC)(0x00000152)
790   #endif
791   #ifndef CC_Create
792   #   define CC_Create NO
793   #endif
794   #if CC_Create == YES
795   #define  TPM_CC_Create                        (TPM_CC)(0x00000153)
796   #endif
797   #ifndef CC_ECDH_ZGen
798   #   define CC_ECDH_ZGen NO
799   #endif
800   #if CC_ECDH_ZGen == YES
801   #define  TPM_CC_ECDH_ZGen                     (TPM_CC)(0x00000154)
802   #endif
803   #ifndef CC_HMAC
804   #   define CC_HMAC NO
805   #endif
806   #if CC_HMAC == YES
807   #define  TPM_CC_HMAC                          (TPM_CC)(0x00000155)
808   #endif
809   #ifndef CC_Import
810   #   define CC_Import NO
811   #endif
812   #if CC_Import == YES
813   #define  TPM_CC_Import                        (TPM_CC)(0x00000156)
```

```
814   #endif
815   #ifndef CC_Load
816   #   define CC_Load NO
817   #endif
818   #if CC_Load == YES
819   #define  TPM_CC_Load                          (TPM_CC)(0x00000157)
820   #endif
821   #ifndef CC_Quote
822   #   define CC_Quote NO
823   #endif
824   #if CC_Quote == YES
825   #define  TPM_CC_Quote                         (TPM_CC)(0x00000158)
826   #endif
827   #ifndef CC_RSA_Decrypt
828   #   define CC_RSA_Decrypt NO
829   #endif
830   #if CC_RSA_Decrypt == YES
831   #define  TPM_CC_RSA_Decrypt                   (TPM_CC)(0x00000159)
832   #endif
833   #ifndef CC_HMAC_Start
834   #   define CC_HMAC_Start NO
835   #endif
836   #if CC_HMAC_Start == YES
837   #define  TPM_CC_HMAC_Start                    (TPM_CC)(0x0000015b)
838   #endif
839   #ifndef CC_SequenceUpdate
840   #   define CC_SequenceUpdate NO
841   #endif
842   #if CC_SequenceUpdate == YES
843   #define  TPM_CC_SequenceUpdate                (TPM_CC)(0x0000015c)
844   #endif
845   #ifndef CC_Sign
846   #   define CC_Sign NO
847   #endif
848   #if CC_Sign == YES
849   #define  TPM_CC_Sign                          (TPM_CC)(0x0000015d)
850   #endif
851   #ifndef CC_Unseal
852   #   define CC_Unseal NO
853   #endif
854   #if CC_Unseal == YES
855   #define  TPM_CC_Unseal                        (TPM_CC)(0x0000015e)
856   #endif
857   #ifndef CC_PolicySigned
858   #   define CC_PolicySigned NO
859   #endif
860   #if CC_PolicySigned == YES
861   #define  TPM_CC_PolicySigned                  (TPM_CC)(0x00000160)
862   #endif
863   #ifndef CC_ContextLoad
864   #   define CC_ContextLoad NO
865   #endif
866   #if CC_ContextLoad == YES
867   #define  TPM_CC_ContextLoad                   (TPM_CC)(0x00000161)
868   #endif
869   #ifndef CC_ContextSave
870   #   define CC_ContextSave NO
871   #endif
872   #if CC_ContextSave == YES
873   #define  TPM_CC_ContextSave                   (TPM_CC)(0x00000162)
874   #endif
875   #ifndef CC_ECDH_KeyGen
876   #   define CC_ECDH_KeyGen NO
877   #endif
878   #if CC_ECDH_KeyGen == YES
879   #define  TPM_CC_ECDH_KeyGen                   (TPM_CC)(0x00000163)
```



```
880   #endif
881   #ifndef CC_EncryptDecrypt
882   #   define CC_EncryptDecrypt NO
883   #endif
884   #if CC_EncryptDecrypt == YES
885   #define  TPM_CC_EncryptDecrypt                (TPM_CC)(0x00000164)
886   #endif
887   #ifndef CC_FlushContext
888   #   define CC_FlushContext NO
889   #endif
890   #if CC_FlushContext == YES
891   #define  TPM_CC_FlushContext                  (TPM_CC)(0x00000165)
892   #endif
893   #ifndef CC_LoadExternal
894   #   define CC_LoadExternal NO
895   #endif
896   #if CC_LoadExternal == YES
897   #define  TPM_CC_LoadExternal                  (TPM_CC)(0x00000167)
898   #endif
899   #ifndef CC_MakeCredential
900   #   define CC_MakeCredential NO
901   #endif
902   #if CC_MakeCredential == YES
903   #define  TPM_CC_MakeCredential                (TPM_CC)(0x00000168)
904   #endif
905   #ifndef CC_NV_ReadPublic
906   #   define CC_NV_ReadPublic NO
907   #endif
908   #if CC_NV_ReadPublic == YES
909   #define  TPM_CC_NV_ReadPublic                 (TPM_CC)(0x00000169)
910   #endif
911   #ifndef CC_PolicyAuthorize
912   #   define CC_PolicyAuthorize NO
913   #endif
914   #if CC_PolicyAuthorize == YES
915   #define  TPM_CC_PolicyAuthorize               (TPM_CC)(0x0000016a)
916   #endif
917   #ifndef CC_PolicyAuthValue
918   #   define CC_PolicyAuthValue NO
919   #endif
920   #if CC_PolicyAuthValue == YES
921   #define  TPM_CC_PolicyAuthValue               (TPM_CC)(0x0000016b)
922   #endif
923   #ifndef CC_PolicyCommandCode
924   #   define CC_PolicyCommandCode NO
925   #endif
926   #if CC_PolicyCommandCode == YES
927   #define  TPM_CC_PolicyCommandCode             (TPM_CC)(0x0000016c)
928   #endif
929   #ifndef CC_PolicyCounterTimer
930   #   define CC_PolicyCounterTimer NO
931   #endif
932   #if CC_PolicyCounterTimer == YES
933   #define  TPM_CC_PolicyCounterTimer            (TPM_CC)(0x0000016d)
934   #endif
935   #ifndef CC_PolicyCpHash
936   #   define CC_PolicyCpHash NO
937   #endif
938   #if CC_PolicyCpHash == YES
939   #define  TPM_CC_PolicyCpHash                  (TPM_CC)(0x0000016e)
940   #endif
941   #ifndef CC_PolicyLocality
942   #   define CC_PolicyLocality NO
943   #endif
944   #if CC_PolicyLocality == YES
945   #define  TPM_CC_PolicyLocality                (TPM_CC)(0x0000016f)
```



```
946   #endif
947   #ifndef CC_PolicyNameHash
948   #   define CC_PolicyNameHash NO
949   #endif
950   #if CC_PolicyNameHash == YES
951   #define  TPM_CC_PolicyNameHash                (TPM_CC)(0x00000170)
952   #endif
953   #ifndef CC_PolicyOR
954   #   define CC_PolicyOR NO
955   #endif
956   #if CC_PolicyOR == YES
957   #define  TPM_CC_PolicyOR                      (TPM_CC)(0x00000171)
958   #endif
959   #ifndef CC_PolicyTicket
960   #   define CC_PolicyTicket NO
961   #endif
962   #if CC_PolicyTicket == YES
963   #define  TPM_CC_PolicyTicket                  (TPM_CC)(0x00000172)
964   #endif
965   #ifndef CC_ReadPublic
966   #   define CC_ReadPublic NO
967   #endif
968   #if CC_ReadPublic == YES
969   #define  TPM_CC_ReadPublic                    (TPM_CC)(0x00000173)
970   #endif
971   #ifndef CC_RSA_Encrypt
972   #   define CC_RSA_Encrypt NO
973   #endif
974   #if CC_RSA_Encrypt == YES
975   #define  TPM_CC_RSA_Encrypt                   (TPM_CC)(0x00000174)
976   #endif
977   #ifndef CC_StartAuthSession
978   #   define CC_StartAuthSession NO
979   #endif
980   #if CC_StartAuthSession == YES
981   #define  TPM_CC_StartAuthSession              (TPM_CC)(0x00000176)
982   #endif
983   #ifndef CC_VerifySignature
984   #   define CC_VerifySignature NO
985   #endif
986   #if CC_VerifySignature == YES
987   #define  TPM_CC_VerifySignature               (TPM_CC)(0x00000177)
988   #endif
989   #ifndef CC_ECC_Parameters
990   #   define CC_ECC_Parameters NO
991   #endif
992   #if CC_ECC_Parameters == YES
993   #define  TPM_CC_ECC_Parameters                (TPM_CC)(0x00000178)
994   #endif
995   #ifndef CC_FirmwareRead
996   #   define CC_FirmwareRead NO
997   #endif
998   #if CC_FirmwareRead == YES
999   #define  TPM_CC_FirmwareRead                  (TPM_CC)(0x00000179)
1000  #endif
1001  #ifndef CC_GetCapability
1002  #   define CC_GetCapability NO
1003  #endif
1004  #if CC_GetCapability == YES
1005  #define  TPM_CC_GetCapability                 (TPM_CC)(0x0000017a)
1006  #endif
1007  #ifndef CC_GetRandom
1008  #   define CC_GetRandom NO
1009  #endif
1010  #if CC_GetRandom == YES
1011  #define  TPM_CC_GetRandom                     (TPM_CC)(0x0000017b)
```



```
1012    #endif
1013    #ifndef CC_GetTestResult
1014    #   define CC_GetTestResult NO
1015    #endif
1016    #if CC_GetTestResult == YES
1017    #define  TPM_CC_GetTestResult                (TPM_CC)(0x0000017c)
1018    #endif
1019    #ifndef CC_Hash
1020    #   define CC_Hash NO
1021    #endif
1022    #if CC_Hash == YES
1023    #define  TPM_CC_Hash                         (TPM_CC)(0x0000017d)
1024    #endif
1025    #ifndef CC_PCR_Read
1026    #   define CC_PCR_Read NO
1027    #endif
1028    #if CC_PCR_Read == YES
1029    #define  TPM_CC_PCR_Read                     (TPM_CC)(0x0000017e)
1030    #endif
1031    #ifndef CC_PolicyPCR
1032    #   define CC_PolicyPCR NO
1033    #endif
1034    #if CC_PolicyPCR == YES
1035    #define  TPM_CC_PolicyPCR                    (TPM_CC)(0x0000017f)
1036    #endif
1037    #ifndef CC_PolicyRestart
1038    #   define CC_PolicyRestart NO
1039    #endif
1040    #if CC_PolicyRestart == YES
1041    #define  TPM_CC_PolicyRestart                (TPM_CC)(0x00000180)
1042    #endif
1043    #ifndef CC_ReadClock
1044    #   define CC_ReadClock NO
1045    #endif
1046    #if CC_ReadClock == YES
1047    #define  TPM_CC_ReadClock                    (TPM_CC)(0x00000181)
1048    #endif
1049    #ifndef CC_PCR_Extend
1050    #   define CC_PCR_Extend NO
1051    #endif
1052    #if CC_PCR_Extend == YES
1053    #define  TPM_CC_PCR_Extend                   (TPM_CC)(0x00000182)
1054    #endif
1055    #ifndef CC_PCR_SetAuthValue
1056    #   define CC_PCR_SetAuthValue NO
1057    #endif
1058    #if CC_PCR_SetAuthValue == YES
1059    #define  TPM_CC_PCR_SetAuthValue             (TPM_CC)(0x00000183)
1060    #endif
1061    #ifndef CC_NV_Certify
1062    #   define CC_NV_Certify NO
1063    #endif
1064    #if CC_NV_Certify == YES
1065    #define  TPM_CC_NV_Certify                   (TPM_CC)(0x00000184)
1066    #endif
1067    #ifndef CC_EventSequenceComplete
1068    #   define CC_EventSequenceComplete NO
1069    #endif
1070    #if CC_EventSequenceComplete == YES
1071    #define  TPM_CC_EventSequenceComplete        (TPM_CC)(0x00000185)
1072    #endif
1073    #ifndef CC_HashSequenceStart
1074    #   define CC_HashSequenceStart NO
1075    #endif
1076    #if CC_HashSequenceStart == YES
1077    #define  TPM_CC_HashSequenceStart            (TPM_CC)(0x00000186)
```

```
1078  #endif
1079  #ifndef CC_PolicyPhysicalPresence
1080  #   define CC_PolicyPhysicalPresence NO
1081  #endif
1082  #if CC_PolicyPhysicalPresence == YES
1083  #define  TPM_CC_PolicyPhysicalPresence        (TPM_CC)(0x00000187)
1084  #endif
1085  #ifndef CC_PolicyDuplicationSelect
1086  #   define CC_PolicyDuplicationSelect NO
1087  #endif
1088  #if CC_PolicyDuplicationSelect == YES
1089  #define  TPM_CC_PolicyDuplicationSelect       (TPM_CC)(0x00000188)
1090  #endif
1091  #ifndef CC_PolicyGetDigest
1092  #   define CC_PolicyGetDigest NO
1093  #endif
1094  #if CC_PolicyGetDigest == YES
1095  #define  TPM_CC_PolicyGetDigest               (TPM_CC)(0x00000189)
1096  #endif
1097  #ifndef CC_TestParms
1098  #   define CC_TestParms NO
1099  #endif
1100  #if CC_TestParms == YES
1101  #define  TPM_CC_TestParms                     (TPM_CC)(0x0000018a)
1102  #endif
1103  #ifndef CC_Commit
1104  #   define CC_Commit NO
1105  #endif
1106  #if CC_Commit == YES
1107  #define  TPM_CC_Commit                        (TPM_CC)(0x0000018b)
1108  #endif
1109  #ifndef CC_PolicyPassword
1110  #   define CC_PolicyPassword NO
1111  #endif
1112  #if CC_PolicyPassword == YES
1113  #define  TPM_CC_PolicyPassword                (TPM_CC)(0x0000018c)
1114  #endif
1115  #ifndef CC_ZGen_2Phase
1116  #   define CC_ZGen_2Phase NO
1117  #endif
1118  #if CC_ZGen_2Phase == YES
1119  #define  TPM_CC_ZGen_2Phase                   (TPM_CC)(0x0000018d)
1120  #endif
1121  #ifndef CC_EC_Ephemeral
1122  #   define CC_EC_Ephemeral NO
1123  #endif
1124  #if CC_EC_Ephemeral == YES
1125  #define  TPM_CC_EC_Ephemeral                  (TPM_CC)(0x0000018e)
1126  #endif
1127  #ifndef CC_PolicyNvWritten
1128  #   define CC_PolicyNvWritten NO
1129  #endif
1130  #if CC_PolicyNvWritten == YES
1131  #define  TPM_CC_PolicyNvWritten               (TPM_CC)(0x0000018f)
1132  #endif
1133  #ifndef CC_PolicyTemplate
1134  #   define CC_PolicyTemplate NO
1135  #endif
1136  #if CC_PolicyTemplate == YES
1137  #define  TPM_CC_PolicyTemplate                (TPM_CC)(0x00000190)
1138  #endif
1139  #ifndef CC_CreateLoaded
1140  #   define CC_CreateLoaded NO
1141  #endif
1142  #if CC_CreateLoaded == YES
1143  #define  TPM_CC_CreateLoaded                  (TPM_CC)(0x00000191)
```

```
1144    #endif
1145    #ifndef CC_PolicyAuthorizeNV
1146    #   define CC_PolicyAuthorizeNV NO
1147    #endif
1148    #if CC_PolicyAuthorizeNV == YES
1149    #define  TPM_CC_PolicyAuthorizeNV             (TPM_CC)(0x00000192)
1150    #endif
1151    #ifndef CC_EncryptDecrypt2
1152    #   define CC_EncryptDecrypt2 NO
1153    #endif
1154    #if CC_EncryptDecrypt2 == YES
1155    #define  TPM_CC_EncryptDecrypt2               (TPM_CC)(0x00000193)
1156    #endif
1157    #define  CC_VEND                              (TPM_CC)(0x20000000)
1158    #ifndef CC_Vendor_TCG_Test
1159    #   define CC_Vendor_TCG_Test NO
1160    #endif
1161    #if CC_Vendor_TCG_Test == YES
1162    #define  TPM_CC_Vendor_TCG_Test               (TPM_CC)(0x20000000)
1163    #endif
1164    #ifndef  COMPRESSED_LISTS
1165    #define ADD_FILL    1
1166    #else
1167    #define ADD_FILL   0
1168    #endif
```

Size the array of library commands based on whether or not the array is packed (only defined commands) or dense (having entries for unimplemented commands)

```
1169    #define LIBRARY_COMMAND_ARRAY_SIZE       (0      \
1170        + (ADD_FILL || CC_NV_UndefineSpaceSpecial)    /* 0x0000011f */ \
1171        + (ADD_FILL || CC_EvictControl)               /* 0x00000120 */ \
1172        + (ADD_FILL || CC_HierarchyControl)           /* 0x00000121 */ \
1173        + (ADD_FILL || CC_NV_UndefineSpace)           /* 0x00000122 */ \
1174        + ADD_FILL                                    /* 0x00000123 */ \
1175        + (ADD_FILL || CC_ChangeEPS)                  /* 0x00000124 */ \
1176        + (ADD_FILL || CC_ChangePPS)                  /* 0x00000125 */ \
1177        + (ADD_FILL || CC_Clear)                      /* 0x00000126 */ \
1178        + (ADD_FILL || CC_ClearControl)               /* 0x00000127 */ \
1179        + (ADD_FILL || CC_ClockSet)                   /* 0x00000128 */ \
1180        + (ADD_FILL || CC_HierarchyChangeAuth)        /* 0x00000129 */ \
1181        + (ADD_FILL || CC_NV_DefineSpace)             /* 0x0000012a */ \
1182        + (ADD_FILL || CC_PCR_Allocate)               /* 0x0000012b */ \
1183        + (ADD_FILL || CC_PCR_SetAuthPolicy)          /* 0x0000012c */ \
1184        + (ADD_FILL || CC_PP_Commands)                /* 0x0000012d */ \
1185        + (ADD_FILL || CC_SetPrimaryPolicy)           /* 0x0000012e */ \
1186        + (ADD_FILL || CC_FieldUpgradeStart)          /* 0x0000012f */ \
1187        + (ADD_FILL || CC_ClockRateAdjust)            /* 0x00000130 */ \
1188        + (ADD_FILL || CC_CreatePrimary)              /* 0x00000131 */ \
1189        + (ADD_FILL || CC_NV_GlobalWriteLock)         /* 0x00000132 */ \
1190        + (ADD_FILL || CC_GetCommandAuditDigest)      /* 0x00000133 */ \
1191        + (ADD_FILL || CC_NV_Increment)               /* 0x00000134 */ \
1192        + (ADD_FILL || CC_NV_SetBits)                 /* 0x00000135 */ \
1193        + (ADD_FILL || CC_NV_Extend)                  /* 0x00000136 */ \
1194        + (ADD_FILL || CC_NV_Write)                   /* 0x00000137 */ \
1195        + (ADD_FILL || CC_NV_WriteLock)               /* 0x00000138 */ \
1196        + (ADD_FILL || CC_DictionaryAttackLockReset)  /* 0x00000139 */ \
1197        + (ADD_FILL || CC_DictionaryAttackParameters) /* 0x0000013a */ \
1198        + (ADD_FILL || CC_NV_ChangeAuth)              /* 0x0000013b */ \
1199        + (ADD_FILL || CC_PCR_Event)                  /* 0x0000013c */ \
1200        + (ADD_FILL || CC_PCR_Reset)                  /* 0x0000013d */ \
1201        + (ADD_FILL || CC_SequenceComplete)           /* 0x0000013e */ \
1202        + (ADD_FILL || CC_SetAlgorithmSet)            /* 0x0000013f */ \
1203        + (ADD_FILL || CC_SetCommandCodeAuditStatus)  /* 0x00000140 */ \
1204        + (ADD_FILL || CC_FieldUpgradeData)           /* 0x00000141 */ \
```

```
1205        + (ADD_FILL || CC_IncrementalSelfTest)        /* 0x00000142 */ \
1206        + (ADD_FILL || CC_SelfTest)                   /* 0x00000143 */ \
1207        + (ADD_FILL || CC_Startup)                    /* 0x00000144 */ \
1208        + (ADD_FILL || CC_Shutdown)                   /* 0x00000145 */ \
1209        + (ADD_FILL || CC_StirRandom)                 /* 0x00000146 */ \
1210        + (ADD_FILL || CC_ActivateCredential)         /* 0x00000147 */ \
1211        + (ADD_FILL || CC_Certify)                    /* 0x00000148 */ \
1212        + (ADD_FILL || CC_PolicyNV)                   /* 0x00000149 */ \
1213        + (ADD_FILL || CC_CertifyCreation)            /* 0x0000014a */ \
1214        + (ADD_FILL || CC_Duplicate)                  /* 0x0000014b */ \
1215        + (ADD_FILL || CC_GetTime)                    /* 0x0000014c */ \
1216        + (ADD_FILL || CC_GetSessionAuditDigest)      /* 0x0000014d */ \
1217        + (ADD_FILL || CC_NV_Read)                    /* 0x0000014e */ \
1218        + (ADD_FILL || CC_NV_ReadLock)                /* 0x0000014f */ \
1219        + (ADD_FILL || CC_ObjectChangeAuth)           /* 0x00000150 */ \
1220        + (ADD_FILL || CC_PolicySecret)               /* 0x00000151 */ \
1221        + (ADD_FILL || CC_Rewrap)                     /* 0x00000152 */ \
1222        + (ADD_FILL || CC_Create)                     /* 0x00000153 */ \
1223        + (ADD_FILL || CC_ECDH_ZGen)                  /* 0x00000154 */ \
1224        + (ADD_FILL || CC_HMAC)                       /* 0x00000155 */ \
1225        + (ADD_FILL || CC_Import)                     /* 0x00000156 */ \
1226        + (ADD_FILL || CC_Load)                       /* 0x00000157 */ \
1227        + (ADD_FILL || CC_Quote)                      /* 0x00000158 */ \
1228        + (ADD_FILL || CC_RSA_Decrypt)                /* 0x00000159 */ \
1229        + ADD_FILL                                          /* 0x0000015a */ \
1230        + (ADD_FILL || CC_HMAC_Start)                 /* 0x0000015b */ \
1231        + (ADD_FILL || CC_SequenceUpdate)             /* 0x0000015c */ \
1232        + (ADD_FILL || CC_Sign)                       /* 0x0000015d */ \
1233        + (ADD_FILL || CC_Unseal)                     /* 0x0000015e */ \
1234        + ADD_FILL                                          /* 0x0000015f */ \
1235        + (ADD_FILL || CC_PolicySigned)               /* 0x00000160 */ \
1236        + (ADD_FILL || CC_ContextLoad)                /* 0x00000161 */ \
1237        + (ADD_FILL || CC_ContextSave)                /* 0x00000162 */ \
1238        + (ADD_FILL || CC_ECDH_KeyGen)                /* 0x00000163 */ \
1239        + (ADD_FILL || CC_EncryptDecrypt)             /* 0x00000164 */ \
1240        + (ADD_FILL || CC_FlushContext)               /* 0x00000165 */ \
1241        + ADD_FILL                                          /* 0x00000166 */ \
1242        + (ADD_FILL || CC_LoadExternal)               /* 0x00000167 */ \
1243        + (ADD_FILL || CC_MakeCredential)             /* 0x00000168 */ \
1244        + (ADD_FILL || CC_NV_ReadPublic)              /* 0x00000169 */ \
1245        + (ADD_FILL || CC_PolicyAuthorize)            /* 0x0000016a */ \
1246        + (ADD_FILL || CC_PolicyAuthValue)            /* 0x0000016b */ \
1247        + (ADD_FILL || CC_PolicyCommandCode)          /* 0x0000016c */ \
1248        + (ADD_FILL || CC_PolicyCounterTimer)         /* 0x0000016d */ \
1249        + (ADD_FILL || CC_PolicyCpHash)               /* 0x0000016e */ \
1250        + (ADD_FILL || CC_PolicyLocality)             /* 0x0000016f */ \
1251        + (ADD_FILL || CC_PolicyNameHash)             /* 0x00000170 */ \
1252        + (ADD_FILL || CC_PolicyOR)                   /* 0x00000171 */ \
1253        + (ADD_FILL || CC_PolicyTicket)               /* 0x00000172 */ \
1254        + (ADD_FILL || CC_ReadPublic)                 /* 0x00000173 */ \
1255        + (ADD_FILL || CC_RSA_Encrypt)                /* 0x00000174 */ \
1256        + ADD_FILL                                          /* 0x00000175 */ \
1257        + (ADD_FILL || CC_StartAuthSession)           /* 0x00000176 */ \
1258        + (ADD_FILL || CC_VerifySignature)            /* 0x00000177 */ \
1259        + (ADD_FILL || CC_ECC_Parameters)             /* 0x00000178 */ \
1260        + (ADD_FILL || CC_FirmwareRead)               /* 0x00000179 */ \
1261        + (ADD_FILL || CC_GetCapability)              /* 0x0000017a */ \
1262        + (ADD_FILL || CC_GetRandom)                  /* 0x0000017b */ \
1263        + (ADD_FILL || CC_GetTestResult)              /* 0x0000017c */ \
1264        + (ADD_FILL || CC_Hash)                       /* 0x0000017d */ \
1265        + (ADD_FILL || CC_PCR_Read)                   /* 0x0000017e */ \
1266        + (ADD_FILL || CC_PolicyPCR)                  /* 0x0000017f */ \
1267        + (ADD_FILL || CC_PolicyRestart)              /* 0x00000180 */ \
1268        + (ADD_FILL || CC_ReadClock)                  /* 0x00000181 */ \
1269        + (ADD_FILL || CC_PCR_Extend)                 /* 0x00000182 */ \
1270        + (ADD_FILL || CC_PCR_SetAuthValue)           /* 0x00000183 */ \
```

```
1271        + (ADD_FILL || CC_NV_Certify)                /* 0x00000184 */ \
1272        + (ADD_FILL || CC_EventSequenceComplete)     /* 0x00000185 */ \
1273        + (ADD_FILL || CC_HashSequenceStart)         /* 0x00000186 */ \
1274        + (ADD_FILL || CC_PolicyPhysicalPresence)    /* 0x00000187 */ \
1275        + (ADD_FILL || CC_PolicyDuplicationSelect)   /* 0x00000188 */ \
1276        + (ADD_FILL || CC_PolicyGetDigest)           /* 0x00000189 */ \
1277        + (ADD_FILL || CC_TestParms)                 /* 0x0000018a */ \
1278        + (ADD_FILL || CC_Commit)                    /* 0x0000018b */ \
1279        + (ADD_FILL || CC_PolicyPassword)            /* 0x0000018c */ \
1280        + (ADD_FILL || CC_ZGen_2Phase)               /* 0x0000018d */ \
1281        + (ADD_FILL || CC_EC_Ephemeral)              /* 0x0000018e */ \
1282        + (ADD_FILL || CC_PolicyNvWritten)           /* 0x0000018f */ \
1283        + (ADD_FILL || CC_PolicyTemplate)            /* 0x00000190 */ \
1284        + (ADD_FILL || CC_CreateLoaded)              /* 0x00000191 */ \
1285        + (ADD_FILL || CC_PolicyAuthorizeNV)         /* 0x00000192 */ \
1286        + (ADD_FILL || CC_EncryptDecrypt2)           /* 0x00000193 */ \
1287        )
1288    #define VENDOR_COMMAND_ARRAY_SIZE   ( 0   \
1289        + CC_Vendor_TCG_Test            \
1290        )
1291    #define COMMAND_COUNT           \
1292                (LIBRARY_COMMAND_ARRAY_SIZE + VENDOR_COMMAND_ARRAY_SIZE)
1293    #ifndef MAX
1294    #define MAX(a, b) ((a) > (b) ? (a) : (b))
1295    #endif
1296    #define MAX_HASH_BLOCK_SIZE  (                  \
1297        MAX(ALG_SHA1 * SHA1_BLOCK_SIZE,             \
1298        MAX(ALG_SHA256 * SHA256_BLOCK_SIZE,         \
1299        MAX(ALG_SHA384 * SHA384_BLOCK_SIZE,         \
1300        MAX(ALG_SHA512 * SHA512_BLOCK_SIZE,         \
1301        MAX(ALG_SM3_256 * SM3_256_BLOCK_SIZE,       \
1302        0 ))))))
1303    #define MAX_DIGEST_SIZE      (                  \
1304        MAX(ALG_SHA1 * SHA1_DIGEST_SIZE,            \
1305        MAX(ALG_SHA256 * SHA256_DIGEST_SIZE,        \
1306        MAX(ALG_SHA384 * SHA384_DIGEST_SIZE,        \
1307        MAX(ALG_SHA512 * SHA512_DIGEST_SIZE,        \
1308        MAX(ALG_SM3_256 * SM3_256_DIGEST_SIZE,      \
1309        0 ))))))
1310    #if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
1311    #error "Hash data not valid"
1312    #endif
1313    #define HASH_COUNT (ALG_SHA1+ALG_SHA256+ALG_SHA384+ALG_SHA512+ALG_SM3_256)
```

Define the 2B structure that would hold any hash block

```
1314    TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
```

Following typedef is for some old code

```
1315    typedef TPM2B_MAX_HASH_BLOCK    TPM2B_HASH_BLOCK;
1316    #ifndef MAX
1317    #define MAX(a, b) ((a) > (b) ? (a) : (b))
1318    #endif
1319    #ifndef ALG_AES
1320    #   define ALG_AES         NO
1321    #endif
1322    #ifndef MAX_AES_KEY_BITS
1323    #   define      MAX_AES_KEY_BITS  0
1324    #   define      MAX_AES_BLOCK_SIZE_BYTES 0
1325    #endif
1326    #ifndef ALG_CAMELLIA
1327    #   define ALG_CAMELLIA         NO
1328    #endif
1329    #ifndef MAX_CAMELLIA_KEY_BITS
```

```
#   define      MAX_CAMELLIA_KEY_BITS  0
#   define      MAX_CAMELLIA_BLOCK_SIZE_BYTES 0
#endif
#ifndef ALG_SM4
#   define ALG_SM4         NO
#endif
#ifndef MAX_SM4_KEY_BITS
#   define      MAX_SM4_KEY_BITS  0
#   define      MAX_SM4_BLOCK_SIZE_BYTES 0
#endif
#ifndef ALG_TDES
#   define ALG_TDES         NO
#endif
#ifndef MAX_TDES_KEY_BITS
#   define      MAX_TDES_KEY_BITS  0
#   define      MAX_TDES_BLOCK_SIZE_BYTES 0
#endif
#define MAX_SYM_KEY_BITS (                                  \
            MAX(MAX_AES_KEY_BITS * ALG_AES,           \
            MAX(MAX_CAMELLIA_KEY_BITS * ALG_CAMELLIA,           \
            MAX(MAX_SM4_KEY_BITS * ALG_SM4,           \
            MAX(MAX_TDES_KEY_BITS * ALG_TDES,           \
            0)))))
#define  MAX_SYM_KEY_BYTES ((MAX_SYM_KEY_BITS + 7) / 8)
#define MAX_SYM_BLOCK_SIZE  (                               \
            MAX(MAX_AES_BLOCK_SIZE_BYTES * ALG_AES,   \
            MAX(MAX_CAMELLIA_BLOCK_SIZE_BYTES * ALG_CAMELLIA,   \
            MAX(MAX_SM4_BLOCK_SIZE_BYTES * ALG_SM4,   \
            MAX(MAX_TDES_BLOCK_SIZE_BYTES * ALG_TDES,   \
            0)))))
#if MAX_SYM_KEY_BITS == 0 || MAX_SYM_BLOCK_SIZE == 0
#   error Bad size for MAX_SYM_KEY_BITS or MAX_SYM_BLOCK_SIZE
#endif
```

Define the 2B structure for a seed

```
TPM2B_TYPE(SEED, PRIMARY_SEED_SIZE);
#endif  // _IMPLEMENTATION_H_
```

Annex B
(informative)
Library-Specific

B.1 Introduction

This clause contains the files that are specific to a cryptographic library used by the TPM code.

Three categories are defined for cryptographic functions:

1) big number math (asymmetric cryptography),
2) symmetric ciphers, and
3) hash functions.

The code is structured to make it possible to use different libraries for different categories. For example, one might choose to use OpenSSL for its math library, but use a different library for hashing and symmetric cryptography. Since OpenSSL supports all three categories, it might be more typical to combine libraries of specific functions; that is, one library might only contain block ciphers while another supports big number math.

B.2 OpenSSL-Specific Files

B.2.1. Introduction

The following files are specific to a port that uses the OpenSSL library for cryptographic functions.

B.2.2. Header Files

B.2.2.1. TpmToOsslHash.h

B.2.2.1.1. Introduction

This header file is used to *splice* the OpenSSL() hash code into the TPM code.

```
#ifndef _TPM_TO_OSSL_HASH_H_
#define _TPM_TO_OSSL_HASH_H_
#if HASH_LIB == OSSL
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <openssl/ossl_typ.h>
```

Redefine the internal name used for each of the hash state structures to the name used by the library. These defines need to be known in all parts of the TPM so that the structure sizes can be properly computed when needed.

```
#define tpmHashStateSHA1_t        SHA_CTX
#define tpmHashStateSHA256_t      SHA256_CTX
#define tpmHashStateSHA384_t      SHA512_CTX
#define tpmHashStateSHA512_t      SHA512_CTX
#ifdef TPM_ALG_SM3
#   error "Symcrypt does not support SM3"
#endif
```

The defines below are only needed when compiling CryptHash.c. This isolation is primarily to avoid name space collision. However, if there is a real collision, it will likely show up when the linker tries to put things together.

```
#ifdef _CRYPT_HASH_C_
typedef BYTE          *PBYTE;
typedef const BYTE    *PCBYTE;
```

Define the interface between CryptHash.c to the functions provided by the library. For each method, define the calling parameters of the method and then define how the method is invoked in CryptHash.c.

All hashes are required to have the same calling sequence. If they don't, create a simple adaptation function that converts from the **standard** form of the call to the form used by the specific hash (and then send a nasty letter to the person who wrote the hash function for the library).

The macro that calls the method also defines how the parameters get swizzled between the default form (in CryptHash.c)and the library form.

Initialize the hash context

```
#define HASH_START_METHOD_DEF   void (HASH_START_METHOD)(PANY_HASH_STATE state)
#define HASH_START(hashState)                                                  \
                ((hashState)->def->method.start)(&(hashState)->state);
```


Add data to the hash

```
#define HASH_DATA_METHOD_DEF                                                \
                void (HASH_DATA_METHOD)(PANY_HASH_STATE state,              \
                                        PCBYTE buffer,                      \
                                        size_t size)
#define HASH_DATA(hashState, dInSize, dIn)                                  \
                ((hashState)->def->method.data)(&(hashState)->state, dIn, dInSize)
```

Finalize the hash and get the digest

```
#define HASH_END_METHOD_DEF                                                 \
                void (HASH_END_METHOD)(BYTE *buffer, PANY_HASH_STATE state)
#define HASH_END(hashState, buffer)                                         \
                ((hashState)->def->method.end)(buffer, &(hashState)->state)
```

Copy the hash context

NOTE: For import, export, and copy, memcpy() is used since there is no reformatting necessary between the internal and external forms.

```
#define HASH_STATE_COPY_METHOD_DEF                                          \
                void (HASH_STATE_COPY_METHOD)(PANY_HASH_STATE to,           \
                                              PCANY_HASH_STATE from,        \
                                              size_t size)
#define HASH_STATE_COPY(hashStateOut, hashStateIn)                          \
                ((hashStateIn)->def->method.copy)(&(hashStateOut)->state,   \
                                              &(hashStateIn)->state,        \
                                              (hashStateIn)->def->contextSize)
```

Copy (with reformatting when necessary) an internal hash structure to an external blob

```
#define  HASH_STATE_EXPORT_METHOD_DEF                                       \
                void (HASH_STATE_EXPORT_METHOD)(BYTE *to,                   \
                                          PCANY_HASH_STATE from,            \
                                          size_t size)
#define  HASH_STATE_EXPORT(to, hashStateFrom)                               \
                ((hashStateFrom)->def->method.copyOut)                      \
                        (&(((BYTE *)(to))[offsetof(HASH_STATE, state)]),    \
                         &(hashStateFrom)->state,                           \
                         (hashStateFrom)->def->contextSize)
```

Copy from an external blob to an internal formate (with reformatting when necessary

```
#define  HASH_STATE_IMPORT_METHOD_DEF                                       \
                void (HASH_STATE_IMPORT_METHOD)(PANY_HASH_STATE to,         \
                                            const BYTE *from,               \
                                             size_t size)
#define  HASH_STATE_IMPORT(hashStateTo, from)                               \
                ((hashStateTo)->def->method.copyIn)                         \
                        (&(hashStateTo)->state,                             \
                         &(((const BYTE *)(from))[offsetof(HASH_STATE, state)]),\
                         (hashStateTo)->def->contextSize)
```

Function aliases. The code in CryptHash.c uses the internal designation for the functions. These need to be translated to the function names of the library.

Internal	External
Designation	Designation

```
#define tpmHashStart_SHA1           SHA1_Init   // external name of the
```



```
57                                                  // initialization method
58    #define tpmHashData_SHA1            SHA1_Update
59    #define tpmHashEnd_SHA1             SHA1_Final
60    #define tpmHashStateCopy_SHA1       memcpy
61    #define tpmHashStateExport_SHA1     memcpy
62    #define tpmHashStateImport_SHA1     memcpy
63    #define tpmHashStart_SHA256         SHA256_Init
64    #define tpmHashData_SHA256          SHA256_Update
65    #define tpmHashEnd_SHA256           SHA256_Final
66    #define tpmHashStateCopy_SHA256     memcpy
67    #define tpmHashStateExport_SHA256   memcpy
68    #define tpmHashStateImport_SHA256   memcpy
69    #define tpmHashStart_SHA384         SHA384_Init
70    #define tpmHashData_SHA384          SHA384_Update
71    #define tpmHashEnd_SHA384           SHA384_Final
72    #define tpmHashStateCopy_SHA384     memcpy
73    #define tpmHashStateExport_SHA384   memcpy
74    #define tpmHashStateImport_SHA384   memcpy
75    #define tpmHashStart_SHA512         SHA512_Init
76    #define tpmHashData_SHA512          SHA512_Update
77    #define tpmHashEnd_SHA512           SHA512_Final
78    #define tpmHashStateCopy_SHA512     memcpy
79    #define tpmHashStateExport_SHA512   memcpy
80    #define tpmHashStateImport_SHA512   memcpy
81    #endif // _CRYPT_HASH_C_
82    #define LibHashInit()
```

This definition would change if there were something to report

```
83    #define HashLibSimulationEnd()
84    #endif // HASH_LIB == OSSL
85    #endif //
```


B.2.2.2. TpmToOsslMath.h

B.2.2.2.1. Introduction

This file contains the structure definitions used for ECC in the LibTopCrypt() version of the code. These definitions would change, based on the library. The ECC-related structures that cross the TPM interface are defined in TpmTypes.h

```
#ifndef _TPM_TO_OSSL_MATH_H
#define _TPM_TO_OSSL_MATH_H
#if MATH_LIB == OSSL
#include <openssl/evp.h>
#include <openssl/ec.h>
#include <openssl/bn.h>
```

Make sure that the library is using the correct size for a crypt word

```
#if    defined THIRTY_TWO_BIT && (RADIX_BITS != 32)  \
    || defined SIXTY_FOUR_BIT && (RADIX_BITS != 64)
#  error "Ossl library is using different radix"
#endif
```

Allocate a local BIGNUM value. For the allocation, a *bigNum* structure is created as is a local BIGNUM. The *bigNum* is initialized and then the BIGNUM is set to reference the local value.

```
#define BIG_VAR(name, bits)                                          \
    BN_VAR(name##Bn, (bits));                                        \
    BIGNUM          _##name;                                         \
    BIGNUM          *name = BigInitialized(&_##name,                 \
                                BnInit(name##Bn,                     \
                                BYTES_TO_CRYPT_WORDS(sizeof(_##name##Bn.d))))
```

Allocate a BIGNUM and initialize with the values in a *bigNum* initializer

```
#define BIG_INITIALIZED(name, initializer)                    \
    BIGNUM           _##name;                                 \
    BIGNUM          *name = BigInitialized(&_##name, initializer)
typedef struct
{
    const ECC_CURVE_DATA    *C;     // the TPM curve values
    EC_GROUP                *G;     // group parameters
    BN_CTX                  *CTX;   // the context for the math (this might not be
                                    // the context in which the curve was created>;
} OSSL_CURVE_DATA;
typedef OSSL_CURVE_DATA      *bigCurve;
#define AccessCurveData(E)  ((E)->C)
#define CURVE_INITIALIZED(name, initializer)                     \
    OSSL_CURVE_DATA     _##name;                                \
    bigCurve            name =  BnCurveInitialize(&_##name, initializer)
#include "TpmToOsslSupport_fp.h"
#define CURVE_FREE(E)                   \
    if(E != NULL)                       \
    {                                   \
        if(E->G != NULL)                \
            EC_GROUP_free(E->G);        \
        OsslContextLeave(E->CTX);       \
    }
#define OSSL_ENTER()     BN_CTX      *CTX = OsslContextEnter()
#define OSSL_LEAVE()     OsslContextLeave(CTX)
```

This definition would change if there were something to report

```
42  #define MathLibSimulationEnd()
43  #endif // MATH_LIB == OSSL
44  #endif
```

B.2.2.3. TpmToOsslSym.h

B.2.2.3.1. Introduction

This header file is used to *splice* the OpenSSL() library into the TPM code.

The support required of a library are a hash module, a block cipher module and portions of a big number library.

```
#ifndef _TPM_TO_OSSL_SYM_H_
#define _TPM_TO_OSSL_SYM_H_
#if SYM_LIB == OSSL
#include <openssl/aes.h>
#include <openssl/des.h>
#include <openssl/bn.h>
#include <openssl/ossl_typ.h>
#ifdef TPM_ALG_SM4
#error "SM4 is not available"
#endif
#ifdef  TPM_ALG_CAMELLIA
#error "Camellia is not available"
#endif
```

Define the order of parameters to the library functions that do block encryption and decryption.

```
typedef void(*TpmCryptSetSymKeyCall_t)(
    const BYTE  *in,
    BYTE        *out,
    void *keySchedule
    );
```

The Crypt functions that call the block encryption function use the parameters in the order:

a) *keySchedule*

b) in buffer

c) out buffer Since open SSL uses the order in *encryptoCall_t* above, need to swizzle the values to the order required by the library.

```
#define SWIZZLE(keySchedule, in, out)                                   \
    (const BYTE *)(in), (BYTE *)(out), (void *)(keySchedule)
```

Macros to set up the encryption/decryption key schedules

AES:

```
#define TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)          \
    AES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
#define TpmCryptSetDecryptKeyAES(key, keySizeInBits, schedule)          \
    AES_set_decrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
```

TDES:

```
#define TpmCryptSetEncryptKeyTDES(key, keySizeInBits, schedule)         \
    TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *)(schedule))
#define TpmCryptSetDecryptKeyTDES(key, keySizeInBits, schedule)         \
    TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *)(schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly. Currently, only used by CryptRand.c

When using these calls, to call the AES block encryption code, the caller should use: *TpmCryptEncryptAES*(SWIZZLE*(keySchedule*, in, out));

```
29  #define TpmCryptEncryptAES          AES_encrypt
30  #define TpmCryptDecryptAES          AES_decrypt
31  #define tpmKeyScheduleAES           AES_KEY
32  #define TpmCryptEncryptTDES         TDES_encrypt
33  #define TpmCryptDecryptTDES         TDES_decrypt
34  #define tpmKeyScheduleTDES          DES_key_schedule
35  typedef union tpmCryptKeySchedule_t tpmCryptKeySchedule_t;
36  #ifdef TPM_ALG_TDES
37  #include "TpmToOsslDesSupport_fp.h"
38  #endif
```

This definition would change if there were something to report

```
39  #define SymLibSimulationEnd()
40  #endif // SYM_LIB == OSSL
41  #endif // _TPM_TO_OSSL_SYM_H_
```


B.2.3. Source Files

B.2.3.1. TpmToOsslDesSupport.c

B.2.3.1.1. Introduction

The functions in this file are used for initialization of the interface to the OpenSSL() library.

B.2.3.1.2. Defines and Includes

```
#include "Tpm.h"
#if SYM_LIB == OSSL && defined TPM_ALG_TDES
```

B.2.3.1.3. Functions

B.2.3.1.3.1. TDES_set_encyrpt_key()

This function makes creation of a TDES key look like the creation of a key for any of the other OpenSSL() block ciphers. It will create three key schedules, one for each of the DES keys. If there are only two keys, then the third schedule is a copy of the first.

```
void
TDES_set_encrypt_key(
    const BYTE                  *key,
    UINT16                       keySizeInBits,
    tpmKeyScheduleTDES          *keySchedule
    )
{
    DES_set_key_unchecked((const_DES_cblock *)key, &keySchedule[0]);
    DES_set_key_unchecked((const_DES_cblock *)&key[8], &keySchedule[1]);
    // If is two-key, copy the schedule for K1 into K3, otherwise, compute the
    // the schedule for K3
    if(keySizeInBits == 128)
        keySchedule[2] = keySchedule[0];
    else
        DES_set_key_unchecked((const_DES_cblock *)&key[16],
                              &keySchedule[2]);
}
```

B.2.3.1.3.2. TDES_encyrpt()

The TPM code uses one key schedule. For TDES, the schedule contains three schedules. OpenSSL() wants the schedules referenced separately. This function does that.

```
void TDES_encrypt(
    const BYTE              *in,
    BYTE                    *out,
    tpmKeyScheduleTDES      *ks
    )
{
    DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
                     &ks[0], &ks[1], &ks[2],
                     DES_ENCRYPT);
}
```


B.2.3.1.3.3. TDES_decrypt()

As with TDES_encypt() this function bridges between the TPM single schedule model and the OpenSSL() three schedule model.

```
30  void TDES_decrypt(
31      const BYTE          *in,
32      BYTE                *out,
33      tpmKeyScheduleTDES   *ks
34      )
35  {
36      DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
37                       &ks[0], &ks[1], &ks[2],
38                       DES_DECRYPT);
39  }
40  #endif // SYM_LIB == OSSL
```


B.2.3.2. TpmToOsslMath.c

B.2.3.2.1. Introduction

This file contains the math functions that are not implemented in the BnMath() library (yet). These math functions will call the OpenSSL() library to execute the operations. There is a difference between the internal format and the OpenSSL() format. To call the OpenSSL() function, a BIGNUM structure is created for each passed variable. The sizes in the bignum_t are copied and the *d* pointer in the BIGNUM is set to point to the *d* parameter of the bignum_t. On return, SetSizeOsslToTpm() is used for each returned variable to make sure that the pointers are not changed. The size of the returned BIGGNUM is copied to bignum_t.

B.2.3.2.2. Includes and Defines

```
#include "Tpm.h"
#if MATH_LIB == OSSL
#include "TpmToOsslMath_fp.h"
```

B.2.3.2.3. Functions

```
#if 0
INLINE  void
SetSizeOsslToTpm(
    bigNum        a,
    BIGNUM       *b
    )
{
    if(a != NULL)
    {
        pAssert(((crypt_uword_t *)b->d == &a->d[0])
                && ((unsigned)b->top <= a->allocated));
        a->size = b->top;
    }
}
#endif
```

B.2.3.2.3.1. OsslToTpmBn()

This function converts an OpenSSL() BIGNUM to a TPM bignum. In this implementation it is assumed that OpenSSL() used the same format for a big number as does the TPM -- an array of native-endian words in little-endian order.

If the array allocated for the OpenSSL() BIGNUM is not the space within the TPM bignum, then the data is copied. Otherwise, just the size field of the BIGNUM is copied.

```
void
OsslToTpmBn(
    bigNum          bn,
    BIGNUM          *osslBn
    )
{
    if(bn != NULL)
    {
        if(osslBn->d != bn->d)
        {
            int         i;
            pAssert((unsigned)osslBn->top <= BnGetAllocated(bn));
            for(i = 0; i < osslBn->top; i++)
                bn->d[i] = osslBn->d[i];
```

```
        }
        BnSetTop(bn, osslBn->top);
    }
}
BIGNUM *
BigInitialized(
    BIGNUM             *toInit,
    bigConst            initializer
    )
{
    if(toInit == NULL || initializer == NULL)
        return NULL;
    toInit->d = (crypt_uword_t *)&initializer->d[0];
    toInit->dmax = initializer->allocated;
    toInit->top = initializer->size;
    toInit->neg = 0;
    toInit->flags = 0;
    return toInit;
}
#ifndef OSSL_DEBUG
#   define BIGNUM_PRINT(label, bn, eol)
#   define DEBUG_PRINT(x)
#else
#   define DEBUG_PRINT(x)   printf("%s", x)
#   define BIGNUM_PRINT(label, bn, eol) BIGNUM_print((label), (bn), (eol))
static
void BIGNUM_print(
    const char      *label,
    const BIGNUM    *a,
    BOOL             eol
    )
{
    BN_ULONG        *d;
    int              i;
    int              notZero = FALSE;
    if(label != NULL)
        printf("%s", label);
    if(a == NULL)
    {
        printf("NULL");
        goto done;
    }
    if (a->neg)
        printf("-");
    for(i = a->top, d = &a->d[i - 1]; i > 0; i--)
    {
        int         j;
        BN_ULONG    l = *d--;
        for(j = BN_BITS2 - 8; j >= 0; j -= 8)
        {
            BYTE    b = (BYTE)((l >> j) & 0xFF);
            notZero = notZero || (b != 0);
            if(notZero)
                printf("%02x", b);
        }
        if(!notZero)
            printf("0");
    }
done:
    if(eol)
        printf("\n");
    return;
}
#endif
#ifdef LIBRARY_COMPATIBILITY_CHECK
void
```

```
 99  MathLibraryCompatibilityCheck(
100      void
101      )
102  {
103      OSSL_ENTER();
104      BIGNUM          *osslTemp = BN_CTX_get(CTX);
105      BN_VAR(tpmTemp, 64 * 8); // allocate some space for a test value
106      crypt_uword_t           i;
107      TPM2B_TYPE(TEST, 16);
108      TPM2B_TEST              test = {{16, {0x0F, 0x0E, 0x0D, 0x0C,
109                                            0x0B, 0x0A, 0x09, 0x08,
110                                            0x07, 0x06, 0x05, 0x04,
111                                            0x03, 0x02, 0x01, 0x00}}};
112      // Convert the test TPM2B to a bigNum
113      BnFrom2B(tpmTemp, &test.b);
114      // Convert the test TPM2B to an OpenSSL BIGNUM
115      BN_bin2bn(test.t.buffer, test.t.size, osslTemp);
116      // Make sure the values are consistent
117      cAssert(osslTemp->top == (int)tpmTemp->size);
118      for(i = 0; i < tpmTemp->size; i++)
119          cAssert(osslTemp->d[0] == tpmTemp->d[0]);
120      OSSL_LEAVE();
121  }
122  #endif
```

B.2.3.2.3.2. BnModMult()

Does multiply and divide returning the remainder of the divide.

```
123  LIB_EXPORT BOOL
124  BnModMult(
125      bigNum              result,
126      bigConst            op1,
127      bigConst            op2,
128      bigConst            modulus
129      )
130  {
131      OSSL_ENTER();
132      BIG_INITIALIZED(bnResult, result);
133      BIG_INITIALIZED(bnOp1, op1);
134      BIG_INITIALIZED(bnOp2, op2);
135      BIG_INITIALIZED(bnMod, modulus);
136      BIG_VAR(bnTemp, (LARGEST_NUMBER_BITS * 4));
137      BOOL                OK;
138      pAssert(BnGetAllocated(result) >= BnGetSize(modulus));
139      OK = BN_mul(bnTemp, bnOp1, bnOp2, CTX);
140      OK = OK && BN_div(NULL, bnResult, bnTemp, bnMod, CTX);
141      result->size = bnResult->top;
142      OsslToTpmBn(result, bnResult);
143      OSSL_LEAVE();
144      return OK;
145  }
```

B.2.3.2.3.3. BnMult()

Multiplies two numbers

```
146  LIB_EXPORT BOOL
147  BnMult(
148      bigNum              result,
149      bigConst            multiplicand,
150      bigConst            multiplier
151      )
```

```
152 {
153     OSSL_ENTER();
154     BN_VAR(temp, (LARGEST_NUMBER_BITS * 2));
155     BIG_INITIALIZED(bnTemp, temp);
156     BIG_INITIALIZED(bnA, multiplicand);
157     BIG_INITIALIZED(bnB, multiplier);
158     BOOL                OK;
159     pAssert(result->allocated >=
160             (BITS_TO_CRYPT_WORDS(BnSizeInBits(multiplicand)
161                                  + BnSizeInBits(multiplier))));
162     OK = BN_mul(bnTemp, bnA, bnB, CTX);
163     OsslToTpmBn(temp, bnTemp);
164     OSSL_LEAVE();
165     BnCopy(result, temp);
166     return OK;
167 }
```

B.2.3.2.3.4. BnDiv()

This function divides two *bigNum* values. The function always returns TRUE.

```
168 LIB_EXPORT BOOL
169 BnDiv(
170     bigNum              quotient,
171     bigNum              remainder,
172     bigConst            dividend,
173     bigConst            divisor
174     )
175 {
176     OSSL_ENTER();
177     BIG_INITIALIZED(bnQ, quotient);
178     BIG_INITIALIZED(bnR, remainder);
179     BIG_INITIALIZED(bnDend, dividend);
180     BIG_INITIALIZED(bnSor, divisor);
181     BOOL        OK;
182     pAssert(!BnEqualZero(divisor));
183     if(BnGetSize(dividend) < BnGetSize(divisor))
184     {
185         if(quotient)
186             BnSetWord(quotient, 0);
187         if(remainder)
188             BnCopy(remainder, dividend);
189         OK = TRUE;
190     }
191     else
192     {
193         pAssert((quotient == NULL)
194                 || (quotient->allocated >= (unsigned)(dividend->size
195                                                       - divisor->size)));
196         pAssert((remainder == NULL)
197                 || (remainder->allocated >= divisor->size));
198         OK = BN_div(bnQ, bnR, bnDend, bnSor, CTX);
199         OsslToTpmBn(quotient, bnQ);
200         OsslToTpmBn(remainder, bnR);
201     }
202     DEBUG_PRINT("In BnDiv:\n");
203     BIGNUM_PRINT("   bnDividend: ", bnDend, TRUE);
204     BIGNUM_PRINT("    bnDivisor: ", bnSor, TRUE);
205     BIGNUM_PRINT("   bnQuotient: ", bnQ, TRUE);
206     BIGNUM_PRINT("  bnRemainder: ", bnR, TRUE);
207     OSSL_LEAVE();
208     return OK;
209 }
210 #ifdef TPM_ALG_RSA
```


B.2.3.2.3.5. BnGcd()

Get the greatest common divisor of two numbers

```
211  LIB_EXPORT BOOL
212  BnGcd(
213      bigNum      gcd,            // OUT: the common divisor
214      bigConst    number1,        // IN:
215      bigConst    number2         // IN:
216      )
217  {
218      OSSL_ENTER();
219      BIG_INITIALIZED(bnGcd, gcd);
220      BIG_INITIALIZED(bn1, number1);
221      BIG_INITIALIZED(bn2, number2);
222      BOOL            OK;
223      pAssert(gcd != NULL);
224      OK = BN_gcd(bnGcd, bn1, bn2, CTX);
225      OsslToTpmBn(gcd, bnGcd);
226      gcd->size = bnGcd->top;
227      OSSL_LEAVE();
228      return OK;
229  }
```

B.2.3.2.3.6. BnModExp()

Do modular exponentiation using *bigNum* values. The conversion from a bignum_t to a *bigNum* is trivial as they are based on the same structure

```
230  LIB_EXPORT BOOL
231  BnModExp(
232      bigNum              result,         // OUT: the result
233      bigConst            number,         // IN: number to exponentiate
234      bigConst            exponent,       // IN:
235      bigConst            modulus         // IN:
236      )
237  {
238      OSSL_ENTER();
239      BIG_INITIALIZED(bnResult, result);
240      BIG_INITIALIZED(bnN, number);
241      BIG_INITIALIZED(bnE, exponent);
242      BIG_INITIALIZED(bnM, modulus);
243      BOOL            OK;
244  //
245      OK = BN_mod_exp(bnResult, bnN, bnE, bnM, CTX);
246      OsslToTpmBn(result, bnResult);
247      OSSL_LEAVE();
248      return OK;
249  }
```

B.2.3.2.3.7. BnModInverse()

Modular multiplicative inverse

```
250  LIB_EXPORT BOOL
251  BnModInverse(
252      bigNum              result,
253      bigConst            number,
254      bigConst            modulus
255      )
256  {
257      OSSL_ENTER();
```



```
258         BIG_INITIALIZED(bnResult, result);
259         BIG_INITIALIZED(bnN, number);
260         BIG_INITIALIZED(bnM, modulus);
261         BOOL                OK;
262         OK = (BN_mod_inverse(bnResult, bnN, bnM, CTX) != NULL);
263         OsslToTpmBn(result, bnResult);
264         OSSL_LEAVE();
265         return OK;
266     }
267     #endif // TPM_ALG_RSA
268     #ifdef TPM_ALG_ECC
```

B.2.3.2.3.8. PointFromOssl()

Function to copy the point result from an OSSL function to a *bigNum*

```
269     static BOOL
270     PointFromOssl(
271         bigPoint         pOut,      // OUT: resulting point
272         EC_POINT        *pIn,       // IN: the point to return
273         bigCurve         E          // IN: the curve
274         )
275     {
276         BIGNUM          *x = NULL;
277         BIGNUM          *y = NULL;
278         BOOL             OK;
279         BN_CTX_start(E->CTX);
280     //
281         x = BN_CTX_get(E->CTX);
282         y = BN_CTX_get(E->CTX);
283         if(y == NULL)
284             FAIL(FATAL_ERROR_ALLOCATION);
285         // If this returns false, then the point is at infinity
286         OK = EC_POINT_get_affine_coordinates_GFp(E->G, pIn, x, y, E->CTX);
287         if(OK)
288         {
289             OsslToTpmBn(pOut->x, x);
290             OsslToTpmBn(pOut->y, y);
291             BnSetWord(pOut->z, 1);
292         }
293         else
294             BnSetWord(pOut->z, 0);
295         BN_CTX_end(E->CTX);
296         return OK;
297     }
```

B.2.3.2.3.9. EcPointInitialized()

Allocate and initialize a point.

```
298     static EC_POINT *
299     EcPointInitialized(
300         pointConst          initializer,
301         bigCurve            E
302         )
303     {
304         BIG_INITIALIZED(bnX, (initializer != NULL) ? initializer->x : NULL);
305         BIG_INITIALIZED(bnY, (initializer != NULL) ? initializer->y : NULL);
306         EC_POINT            *P = (initializer != NULL && E != NULL)
307                                  ? EC_POINT_new(E->G) : NULL;
308         pAssert(E != NULL);
309         if(P != NULL)
310             EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, E->CTX);
```

```
311         return P;
312     }
```

B.2.3.2.3.10. BnCurveInitialize()

This function initializes the OpenSSL() group definition

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*

```
313     bigCurve
314     BnCurveInitialize(
315         bigCurve          E,          // IN: curve structure to initialize
316         TPM_ECC_CURVE     curveId     // IN: curve identifier
317         )
318     {
319         EC_GROUP                *group = NULL;
320         EC_POINT                *P = NULL;
321         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
322         BN_CTX                  *CTX = NULL;
323         BIG_INITIALIZED(bnP, C != NULL ? C->prime : NULL);
324         BIG_INITIALIZED(bnA, C != NULL ? C->a : NULL);
325         BIG_INITIALIZED(bnB, C != NULL ? C->b : NULL);
326         BIG_INITIALIZED(bnX, C != NULL ? C->base.x : NULL);
327         BIG_INITIALIZED(bnY, C != NULL ? C->base.y : NULL);
328         BIG_INITIALIZED(bnN, C != NULL ? C->order : NULL);
329         BIG_INITIALIZED(bnH, C != NULL ? C->h : NULL);
330         int                      OK = (C != NULL);
331     //
332         OK = OK && ((CTX = OsslContextEnter()) != NULL);
333         // initialize EC group, associate a generator point and initialize the point
334         // from the parameter data
335         // Create a group structure
336         OK = OK && (group = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, CTX)) != NULL;
337         // Allocate a point in the group that will be used in setting the
338         // generator. This is not needed after the generator is set.
339         OK = OK && ((P = EC_POINT_new(group)) != NULL);
340         // Need to use this in case Montgomery method is being used
341         OK = OK
342             && EC_POINT_set_affine_coordinates_GFp(group, P, bnX, bnY, CTX);
343         // Now set the generator
344         OK = OK && EC_GROUP_set_generator(group, P, bnN, bnH);
345         if(P != NULL)
346             EC_POINT_free(P);
347         if(!OK && group != NULL)
348         {
349             EC_GROUP_free(group);
350             group = NULL;
351         }
352         if(!OK && CTX != NULL)
353         {
354             OsslContextLeave(CTX);
355             CTX = NULL;
356         }
357         E->G = group;
358         E->CTX = CTX;
359         E->C = C;
360         return OK ? E : NULL;
361     }
```

B.2.3.2.3.11. BnEccModMult()

This function does a point multiply of the form R = [d]S

Return Value	Meaning
FALSE	failure in operation; treat as result being point at infinity

```
362  LIB_EXPORT BOOL
363  BnEccModMult(
364      bigPoint             R,          // OUT: computed point
365      pointConst           S,          // IN: point to multiply by 'd' (optional)
366      bigConst             d,          // IN: scalar for [d]S
367      bigCurve             E
368      )
369  {
370      EC_POINT            *pR = EC_POINT_new(E->G);
371      EC_POINT            *pS = EcPointInitialized(S, E);
372      BIG_INITIALIZED(bnD, d);
373      if(S == NULL)
374          EC_POINT_mul(E->G, pR, bnD, NULL, NULL, E->CTX);
375      else
376          EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
377      PointFromOssl(R, pR, E);
378      EC_POINT_free(pR);
379      EC_POINT_free(pS);
380      return !BnEqualZero(R->z);
381  }
```

B.2.3.2.3.12. BnEccModMult2()

This function does a point multiply of the form R = [d]G + [u]Q

Return Value	Meaning
FALSE	failure in operation; treat as result being point at infinity

```
382  LIB_EXPORT BOOL
383  BnEccModMult2(
384      bigPoint             R,          // OUT: computed point
385      pointConst           S,          // IN: optional point
386      bigConst             d,          // IN: scalar for [d]S or [d]G
387      pointConst           Q,          // IN: second point
388      bigConst             u,          // IN: second scalar
389      bigCurve             E           // IN: curve
390      )
391  {
392      EC_POINT            *pR = EC_POINT_new(E->G);
393      EC_POINT            *pS = EcPointInitialized(S, E);
394      BIG_INITIALIZED(bnD, d);
395      EC_POINT            *pQ = EcPointInitialized(Q, E);
396      BIG_INITIALIZED(bnU, u);
397      if(S == NULL || S == (pointConst)&E->C->base)
398          EC_POINT_mul(E->G, pR, bnD, pQ, bnU, E->CTX);
399      else
400      {
401          const EC_POINT        *points[2];
402          const BIGNUM          *scalars[2];
403          points[0] = pS;
404          points[1] = pQ;
405          scalars[0] = bnD;
406          scalars[1] = bnU;
407          EC_POINTs_mul(E->G, pR, NULL, 2, points, scalars, E->CTX);
```



```
408         }
409         PointFromOssl(R, pR, E);
410         EC_POINT_free(pR);
411         EC_POINT_free(pS);
412         EC_POINT_free(pQ);
413         return !BnEqualZero(R->z);
414     }
```

B.2.3.2.4. BnEccAdd()

This function does addition of two points.

Return Value	Meaning
FALSE	failure in operation; treat as result being point at infinity

```
415     LIB_EXPORT BOOL
416     BnEccAdd(
417         bigPoint            R,          // OUT: computed point
418         pointConst          S,          // IN: point to multiply by 'd'
419         pointConst          Q,          // IN: second point
420         bigCurve            E           // IN: curve
421         )
422     {
423         EC_POINT           *pR = EC_POINT_new(E->G);
424         EC_POINT           *pS = EcPointInitialized(S, E);
425         EC_POINT           *pQ = EcPointInitialized(Q, E);
426     //
427         EC_POINT_add(E->G, pR, pS, pQ, E->CTX);
428         PointFromOssl(R, pR, E);
429         EC_POINT_free(pR);
430         EC_POINT_free(pS);
431         EC_POINT_free(pQ);
432         return !BnEqualZero(R->z);
433     }
434     #endif // TPM_ALG_ECC
435     #endif // MATHLIB OSSL
```


B.2.3.3. TpmToOsslSupport.c

B.2.3.3.1. Introduction

The functions in this file are used for initialization of the interface to the OpenSSL() library.

B.2.3.3.2. Defines and Includes

```
#include "Tpm.h"
#if MATH_LIB == OSSL
```

Used to pass the pointers to the correct sub-keys

```
typedef const BYTE *desKeyPointers[3];
```

B.2.3.3.2.1. SupportLibInit()

This does any initialization required by the support library.

```
LIB_EXPORT int
SupportLibInit(
    void
    )
{
#ifdef LIBRARY_COMPATIBILITY_CHECK
    MathLibraryCompatibilityCheck();
#endif
    return TRUE;
}
```

B.2.3.3.2.2. OsslContextEnter()

This function is used to initialize an OpenSSL() context at the start of a function that will call to an OpenSSL() math function.

```
BN_CTX *
OsslContextEnter(
    void
    )
{
    BN_CTX              *context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    return context;
}
```

B.2.3.3.2.3. OsslContextLeave()

This is the companion function to OsslContextEnter().

```
void
OsslContextLeave(
    BN_CTX          *context
    )
{
    if(context != NULL)
    {
```

```
32          BN_CTX_end(context);
33          BN_CTX_free(context);
34      }
35  }
36  #endif // MATH_LIB == OSSL
```


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"
2   #include "Platform_fp.h"
```

C.2.3. Functions

C.2.3.1. _plat__IsCanceled()

Check if the cancel flag is set

Return Value	Meaning
TRUE(1)	if cancel flag is set
FALSE(0)	if cancel flag is not set

```
3   LIB_EXPORT int
4   _plat__IsCanceled(
5       void
6       )
7   {
8       // return cancel flag
9       return s_isCanceled;
10  }
```

C.2.3.2. _plat__SetCancel()

Set cancel flag.

```
11  LIB_EXPORT void
12  _plat__SetCancel(
13      void
14      )
15  {
16      s_isCanceled = TRUE;
17      return;
18  }
```


C.2.3.3. _plat__ClearCancel()

Clear cancel flag

```
19  LIB_EXPORT void
20  _plat__ClearCancel(
21      void
22      )
23  {
24      s_isCanceled = FALSE;
25      return;
26  }
```


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 "PlatformData.h"
#include "Platform_fp.h"
#include "TpmFail_fp.h"
#include <assert.h>
```

C.3.3. Simulator Functions

C.3.3.1. Introduction

This set of functions is intended to be called by the simulator environment in order to simulate hardware events.

C.3.3.2. _plat__TimerReset()

This function sets current system clock time as t0 for counting TPM time. This function is called at a power on event to reset the clock.

```
LIB_EXPORT void
_plat__TimerReset(
    void
    )
{
    s_realTimePrevious = clock();
    s_tpmTime = 0;
    s_adjustRate = CLOCK_NOMINAL;
    s_timerReset = TRUE;
    s_timerStopped = TRUE;
    return;
}
```

C.3.3.3. _plat__TimerRestart()

This function should be called in order to simulate the restart of the timer should it be stopped while power is still applied.

```
LIB_EXPORT void
_plat__TimerRestart(
    void
    )
{
    s_timerStopped = TRUE;
    return;
}
```


C.3.4. Functions Used by TPM

C.3.4.1. Introduction

These functions are called by the TPM code. They should be replaced by appropriated hardware functions.

C.3.4.2. _plat__TimerRead()

This function provides access to the tick timer of the platform. The TPM code uses this value to drive the TPM Clock.

The tick timer is supposed to run when power is applied to the device. This timer should not be reset by time events including _TPM_Init(). It should only be reset when TPM power is re-applied.

If the TPM is run in a protected environment, that environment may provide the tick time to the TPM as long as the time provided by the environment is not allowed to go backwards. If the time provided by the system can go backwards during a power discontinuity, then the _plat__Signal_PowerOn() should call _plat__TimerReset().

The code in this function should be replaced by a read of a hardware tick timer.

```
25  LIB_EXPORT uint64_t
26  _plat__TimerRead(
27      void
28      )
29  {
30  #ifdef HARDWARE_CLOCK
31  #error      "need a defintion for reading the hardware clock"
32      return HARDWARE_CLOCK
33  #else
34  #define BILLION     1000000000
35  #define MILLION     1000000
36  #define THOUSAND    1000
37      clock_t         timeDiff;
38      uint64_t        adjusted;
39  #   define  TOP     (THOUSAND * CLOCK_NOMINAL)
40  #   define  BOTTOM  ((uint64_t)s_adjustRate * CLOCKS_PER_SEC)
41      // Save the value previously read from the system clock
42      timeDiff = s_realTimePrevious;
43      // update with the current value of the system clock
44      s_realTimePrevious = clock();
45      // In the place below when we "put back" the unused part of the timeDiff
46      // it is possible that we can put back more than we take out. That is, we could
47      // take out 1000 mSec, rate adjust it and put back 1001 mS. This means that
48      // on a subsequent call, time may not have caught up. Rather than trying
49      // to rate adjust this, just stop time. This only occurs in a simulation so
50      // time for more than one command being the same should not be an issue.
51      if(timeDiff >= s_realTimePrevious)
52      {
53          s_realTimePrevious = timeDiff;
54          return s_tpmTime;
55      }
56      // Compute the amount of time since the last call to the system clock
57      timeDiff = s_realTimePrevious - timeDiff;
58      // Do the time rate adjustment and conversion from CLOCKS_PER_SEC to mSec
59      adjusted = (((uint64_t)timeDiff * TOP) / BOTTOM);
60      s_tpmTime += (clock_t)adjusted;
61      // Might have some rounding error that would loose CLOCKS. See what is not
62      // being used. As mentioned above, this could result in putting back more than
63      // is taken out
64      adjusted = (adjusted * BOTTOM) / TOP;
65      // If adjusted is not the same as timeDiff, then there is some rounding
```



```
66      // error that needs to be pushed back into the previous sample.
67      // NOTE: the following is so that the fact that everything is signed will not
68      // matter.
69      s_realTimePrevious = (clock_t)((int64_t)s_realTimePrevious - adjusted);
70      s_realTimePrevious += timeDiff;
71  #ifdef  DEBUGGING_TIME
72      // Put this in so that TPM time will pass much faster than real time when
73      // doing debug.
74      // A value of 1000 for DEBUG_TIME_MULTIPLER will make each ms into a second
75      // A good value might be 100
76      return (s_tpmTime * DEBUG_TIME_MULTIPLIER);
77  #endif
78      return s_tpmTime;
79  #endif
80  }
```

C.3.4.3. _plat__TimerWasReset()

This function is used to interrogate the flag indicating if the tick timer has been reset.

If the *resetFlag* parameter is SET, then the flag will be CLEAR before the function returns.

```
81  LIB_EXPORT BOOL
82  _plat__TimerWasReset(
83     void
84      )
85  {
86      BOOL         retVal = s_timerReset;
87      s_timerReset = FALSE;
88      return retVal;
89  }
```

C.3.4.4. _plat__TimerWasStopped()

This function is used to interrogate the flag indicating if the tick timer has been stopped. If so, this is typically a reason to roll the nonce.

This function will CLEAR the *s_timerStopped* flag before returning. This provides functionality that is similar to status register that is cleared when read. This is the model used here because it is the one that has the most impact on the TPM code as the flag can only be accessed by one entity in the TPM. Any other implementation of the hardware can be made to look like a read-once register.

```
90  LIB_EXPORT BOOL
91  _plat__TimerWasStopped(
92      void
93      )
94  {
95      BOOL         retVal = s_timerStopped;
96      s_timerStopped = FALSE;
97      return retVal;
98  }
```

C.3.4.5. _plat__ClockAdjustRate()

Adjust the clock rate

```
99  LIB_EXPORT void
100 _plat__ClockAdjustRate(
101     int              adjust         // IN: the adjust number.  It could be positive
102                                     //     or negative
103     )
```



```
104 {
105     // We expect the caller should only use a fixed set of constant values to
106     // adjust the rate
107     switch(adjust)
108     {
109         case CLOCK_ADJUST_COARSE:
110             s_adjustRate += CLOCK_ADJUST_COARSE;
111             break;
112         case -CLOCK_ADJUST_COARSE:
113             s_adjustRate -= CLOCK_ADJUST_COARSE;
114             break;
115         case CLOCK_ADJUST_MEDIUM:
116             s_adjustRate += CLOCK_ADJUST_MEDIUM;
117             break;
118         case -CLOCK_ADJUST_MEDIUM:
119             s_adjustRate -= CLOCK_ADJUST_MEDIUM;
120             break;
121         case CLOCK_ADJUST_FINE:
122             s_adjustRate += CLOCK_ADJUST_FINE;
123             break;
124         case -CLOCK_ADJUST_FINE:
125             s_adjustRate -= CLOCK_ADJUST_FINE;
126             break;
127         default:
128             // ignore any other values;
129             break;
130     }
131     if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
132         s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
133     if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
134         s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;
135     return;
136 }
```


C.4 Entropy.c

C.4.1. Includes

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <memory.h>
#include "PlatformData.h"
#include "Platform_fp.h"
```

C.4.2. Local values

This is the last 32-bits of hardware entropy produced. We have to check to see that two consecutive 32-bit values are not the same because (according to FIPS 140-2, annex C

> "If each call to a RNG produces blocks of n bits (where n > 15), the first n-bit block generated after power-up, initialization, or reset shall not be used, but shall be saved for comparison with the next n-bit block to be generated. Each subsequent generation of an n-bit block shall be compared with the previously generated block. The test shall fail if any two compared n-bit blocks are equal."

```
extern uint32_t        lastEntropy;
extern int             firstValue;
```

C.4.3. _plat__GetEntropy()

This function is used to get available hardware entropy. In a hardware implementation of this function, there would be no call to the system to get entropy. If the caller does not ask for any entropy, then this is a startup indication and *firstValue* should be reset.

Return Value	Meaning
< 0	hardware failure of the entropy generator, this is sticky
>= 0	the returned amount of entropy (bytes)

```
LIB_EXPORT int32_t
_plat__GetEntropy(
    unsigned char       *entropy,           // output buffer
    uint32_t             amount             // amount requested
    )
{
    uint32_t            rndNum;
    int                 OK = 1;
    if(amount == 0)
    {
        firstValue = 1;
        return 0;
    }
    // Only provide entropy 32 bits at a time to test the ability
    // of the caller to deal with partial results.
    rndNum = rand();
    if(firstValue)
        firstValue = 0;
    else
        OK = (rndNum != lastEntropy);
    if(OK)
    {
        lastEntropy = rndNum;
        if(amount > sizeof(rndNum))
            amount = sizeof(rndNum);
```



```
33              memcpy(entropy, &rndNum, amount);
34          }
35          return (OK) ? (int32_t)amount : -1;
36      }
```

C.5 LocalityPlat.c

C.5.1. Includes

```
#include "PlatformData.h"
#include "Platform_fp.h"
```

C.5.2. Functions

C.5.2.1. _plat__LocalityGet()

Get the most recent command locality in locality value form. This is an integer value for locality and not a locality structure The locality can be 0-4 or 32-255. 5-31 is not allowed.

```
LIB_EXPORT unsigned char
_plat__LocalityGet(
    void
    )
{
    return s_locality;
}
```

C.5.2.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
LIB_EXPORT void
_plat__LocalitySet(
    unsigned char    locality
    )
{
    if(locality > 4 && locality < 32)
        locality = 0;
    s_locality = locality;
    return;
}
```


C.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 <assert.h>
#include "PlatformData.h"
#include "Platform_fp.h"
```

C.6.3. Functions

C.6.3.1. _plat__NvErrors()

This function is used by the simulator to set the error flags in the NV subsystem to simulate an error in the NV loading process

```
LIB_EXPORT void
_plat__NvErrors(
    int             recoverable,
    int             unrecoverable
    )
{
    s_NV_unrecoverable = unrecoverable;
    s_NV_recoverable = recoverable;
}
```

C.6.3.2. _plat__NVEnable()

Enable NV memory.

This version just pulls in data from a file. In a real TPM, with NV on chip, this function would verify the integrity of the saved context. If the NV memory was not on chip but was in something like RPMB, the NV state would be read in, decrypted and integrity checked.

The recovery from an integrity failure depends on where the error occurred. It it was in the state that is discarded by TPM Reset, then the error is recoverable if the TPM is reset. Otherwise, the TPM must go into failure mode.

Return Value	Meaning
0	if success
> 0	if receive recoverable error
<0	if unrecoverable error

```
LIB_EXPORT int
_plat__NVEnable(
    void            *platParameter  // IN: platform specific parameters
    )
{
    NOT_REFERENCED(platParameter);          // to keep compiler quiet
    // Start assuming everything is OK
```

```
22      s_NV_unrecoverable = FALSE;
23      s_NV_recoverable = FALSE;
24  #ifdef FILE_BACKED_NV
25      if(s_NVFile != NULL)
26          return 0;
27      // Try to open an exist NVChip file for read/write
28  #if defined _MSC_VER && 1
29      if(0 != fopen_s(&s_NVFile, "NVChip", "r+b"))
30          s_NVFile = NULL;
31  #else
32      s_NVFile = fopen("NVChip", "r+b");
33  #endif
34      if(NULL != s_NVFile)
35      {
36          // See if the NVChip file is empty
37          fseek(s_NVFile, 0, SEEK_END);
38          if(0 == ftell(s_NVFile))
39              s_NVFile = NULL;
40      }
41      if(s_NVFile == NULL)
42      {
43          // Initialize all the byte in the new file to 0
44          memset(s_NV, 0, NV_MEMORY_SIZE);
45          // If NVChip file does not exist, try to create it for read/write
46  #if defined _MSC_VER && 1
47          if(0 != fopen_s(&s_NVFile, "NVChip", "w+b"))
48              s_NVFile = NULL;
49  #else
50          s_NVFile = fopen("NVChip", "w+b");
51  #endif
52          if(s_NVFile != NULL)
53          {
54          // Start initialize at the end of new file
55              fseek(s_NVFile, 0, SEEK_END);
56              // Write 0s to NVChip file
57              fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);
58          }
59      }
60      else
61      {
62          // If NVChip file exist, assume the size is correct
63          fseek(s_NVFile, 0, SEEK_END);
64          assert(ftell(s_NVFile) == NV_MEMORY_SIZE);
65          // read NV file data to memory
66          fseek(s_NVFile, 0, SEEK_SET);
67          fread(s_NV, NV_MEMORY_SIZE, 1, s_NVFile);
68      }
69  #endif
70      // NV contents have been read and the error checks have been performed. For
71      // simulation purposes, use the signaling interface to indicate if an error is
72      // to be simulated and the type of the error.
73      if(s_NV_unrecoverable)
74          return -1;
75      return s_NV_recoverable;
76  }
```

C.6.3.3. _plat__NVDisable()

Disable NV memory

```
77  LIB_EXPORT void
78  _plat__NVDisable(
79      void
80      )
```

```
{
#ifdef  FILE_BACKED_NV
    assert(s_NVFile != NULL);
    // Close NV file
    fclose(s_NVFile);
    // Set file handle to NULL
    s_NVFile = NULL;
#endif
    return;
}
```

C.6.3.4. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
LIB_EXPORT int
_plat__IsNvAvailable(
    void
    )
{
    // NV is not available if the TPM is in failure mode
    if(!s_NvIsAvailable)
        return 1;
#ifdef FILE_BACKED_NV
    if(s_NVFile == NULL)
        return 1;
#endif
    return 0;
}
```

C.6.3.5. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```
LIB_EXPORT void
_plat__NvMemoryRead(
    unsigned int     startOffset,   // IN: read start
    unsigned int     size,          // IN: size of bytes to read
    void            *data           // OUT: data buffer
    )
{
    assert(startOffset + size <= NV_MEMORY_SIZE);
    // Copy data from RAM
    memcpy(data, &s_NV[startOffset], size);
    return;
}
```

C.6.3.6. _plat__NvIsDifferent()

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE(1)	the NV location is different from the test value
FALSE(0)	the NV location is the same as the test value

```
117  LIB_EXPORT int
118  _plat__NvIsDifferent(
119      unsigned int     startOffset,   // IN: read start
120      unsigned int     size,          // IN: size of bytes to read
121      void            *data           // IN: data buffer
122      )
123  {
124      return (memcmp(&s_NV[startOffset], data, size) != 0);
125  }
```

C.6.3.7. _plat__NvMemoryWrite()

This function is used to update NV memory. The **write** is to a memory copy of NV. At the end of the current command, any changes are written to the actual NV memory.

NOTE: A useful optimization would be for this code to compare the current contents of NV with the local copy and note the blocks that have changed. Then only write those blocks when _plat__NvCommit() is called.

```
126  LIB_EXPORT void
127  _plat__NvMemoryWrite(
128      unsigned int     startOffset,   // IN: write start
129      unsigned int     size,          // IN: size of bytes to write
130      void            *data           // OUT: data buffer
131      )
132  {
133      assert(startOffset + size <= NV_MEMORY_SIZE);
134      // Copy the data to the NV image
135      memcpy(&s_NV[startOffset], data, size);
136  }
```

C.6.3.8. _plat__NvMemoryClear()

Function is used to set a range of NV memory bytes to an implementation-dependent value. The value represents the erase state of the memory.

```
137  LIB_EXPORT void
138  _plat__NvMemoryClear(
139      unsigned int     start,         // IN: clear start
140      unsigned int     size           // IN: number of bytes to clear
141      )
142  {
143      assert(start + size <= NV_MEMORY_SIZE);
144      // In this implementation, assume that the errase value for NV is all 1s
145      memset(&s_NV[start], 0xff, size);
146  }
```

C.6.3.9. _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

```
147  LIB_EXPORT void
148  _plat__NvMemoryMove(
149      unsigned int     sourceOffset,  // IN: source offset
```

```
150         unsigned int     destOffset,    // IN: destination offset
151         unsigned int     size           // IN: size of data being moved
152         )
153     {
154         assert(sourceOffset + size <= NV_MEMORY_SIZE);
155         assert(destOffset + size <= NV_MEMORY_SIZE);
156         // Move data in RAM
157         memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);
158         return;
159     }
```

C.6.3.10. _plat__NvCommit()

Update NV chip

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
160     LIB_EXPORT int
161     _plat__NvCommit(
162         void
163         )
164     {
165     #ifdef FILE_BACKED_NV
166         // If NV file is not available, return failure
167         if(s_NVFile == NULL)
168             return 1;
169         // Write RAM data to NV
170         fseek(s_NVFile, 0, SEEK_SET);
171         fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NVFile);
172         return 0;
173     #else
174         return 0;
175     #endif
176     }
```

C.6.3.11. _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

```
177     LIB_EXPORT void
178     _plat__SetNvAvail(
179         void
180         )
181     {
182         s_NvIsAvailable = TRUE;
183         return;
184     }
```

C.6.3.12. _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

```
185     LIB_EXPORT void
186     _plat__ClearNvAvail(
187         void
```



```
188         )
189     {
190         s_NvIsAvailable = FALSE;
191         return;
192     }
```

C.7 PowerPlat.c

C.7.1. Includes and Function Prototypes

```
#include    "PlatformData.h"
#include    "Platform_fp.h"
#include    "_TPM_Init_fp.h"
```

C.7.2. Functions

C.7.2.1. _plat__Signal_PowerOn()

Signal platform power on

```
LIB_EXPORT int
_plat__Signal_PowerOn(
    void
    )
{
    // Reset the timer
    _plat__TimerReset();
   // Need to indicate that we lost power
    s_powerLost = TRUE;
    return 0;
}
```

C.7.2.2. _plat__WasPowerLost()

Test whether power was lost before a _TPM_Init().

This function will clear the **hardware** indication of power loss before return. This means that there can only be one spot in the TPM code where this value gets read. This method is used here as it is the most difficult to manage in the TPM code and, if the hardware actually works this way, it is hard to make it look like anything else. So, the burden is placed on the TPM code rather than the platform code

Return Value	Meaning
TRUE(1)	power was lost
FALSE(0)	power was not lost

```
LIB_EXPORT int
_plat__WasPowerLost(
    void
    )
{
    BOOL        retVal = s_powerLost;
    s_powerLost = FALSE;
    return retVal;
}
```

C.7.2.3. _plat_Signal_Reset()

This a TPM reset without a power loss.

```
LIB_EXPORT int
_plat__Signal_Reset(
    void
```

```
27      )
28  {
29      // Initialize locality
30      s_locality = 0;
31      // Command cancel
32      s_isCanceled = FALSE;
33      _TPM_Init();
34      // if we are doing reset but did not have a power failure, then we should
35      // not need to reload NV ...
36      return 0;
37  }
```

C.7.2.4. _plat__Signal_PowerOff()

Signal platform power off

```
38  LIB_EXPORT void
39  _plat__Signal_PowerOff(
40      void
41      )
42  {
43      // Prepare NV memory for power off
44      _plat__NVDisable();
45      return;
46  }
```

C.8 Platform_fp.h

```
#ifndef    _PLATFORM_FP_H_
#define    _PLATFORM_FP_H_
```

C.8.1. From Cancel.c

C.8.1.1. _plat__IsCanceled()

Check if the cancel flag is set

Return Value	Meaning
TRUE(1)	if cancel flag is set
FALSE(0)	if cancel flag is not set

```
LIB_EXPORT int
_plat__IsCanceled(
    void
    );
```

Set cancel flag.

```
LIB_EXPORT void
_plat__SetCancel(
    void
    );
```

C.8.1.2. _plat__ClearCancel()

Clear cancel flag

```
LIB_EXPORT void
_plat__ClearCancel(
    void
    );
```

C.8.2. From Clock.c

C.8.2.1. _plat__TimerReset()

This function sets current system clock time as t0 for counting TPM time. This function is called at a power on event to reset the clock.

```
LIB_EXPORT void
_plat__TimerReset(
    void
    );
```

C.8.2.2. _plat__TimerRestart()

This function should be called in order to simulate the restart of the timer should it be stopped while power is still applied.

```
LIB_EXPORT void
```



```
20  _plat__TimerRestart(
21      void
22      );
```

C.8.2.3. _plat__TimerRead()

This function provides access to the tick timer of the platform. The TPM code uses this value to drive the TPM Clock.

The tick timer is supposed to run when power is applied to the device. This timer should not be reset by time events including _TPM_Init(). It should only be reset when TPM power is re-applied.

If the TPM is run in a protected environment, that environment may provide the tick time to the TPM as long as the time provided by the environment is not allowed to go backwards. If the time provided by the system can go backwards during a power discontinuity, then the _plat__Signal_PowerOn() should call _plat__TimerReset().

The code in this function should be replaced by a read of a hardware tick timer.

```
23  LIB_EXPORT uint64_t
24  _plat__TimerRead(
25      void
26      );
```

C.8.2.4. _plat__TimerWasReset()

This function is used to interrogate the flag indicating if the tick timer has been reset.

If the *resetFlag* parameter is SET, then the flag will be CLEAR before the function returns.

```
27  LIB_EXPORT BOOL
28  _plat__TimerWasReset(
29     void
30      );
```

C.8.2.5. _plat__TimerWasStopped()

This function is used to interrogate the flag indicating if the tick timer has been stopped. If so, this is typically a reason to roll the nonce.

This function will CLEAR the *s_timerStopped* flag before returning. This provides functionality that is similar to status register that is cleared when read. This is the model used here because it is the one that has the most impact on the TPM code as the flag can only be accessed by one entity in the TPM. Any other implementation of the hardware can be made to look like a read-once register.

```
31  LIB_EXPORT BOOL
32  _plat__TimerWasStopped(
33      void
34      );
```

C.8.2.6. _plat__ClockAdjustRate()

Adjust the clock rate

```
35  LIB_EXPORT void
36  _plat__ClockAdjustRate(
37      int             adjust          // IN: the adjust number.  It could be positive
38                                      //     or negative
39      );
```

C.8.3. From Entropy.c

Return Value	Meaning
< 0	hardware failure of the entropy generator, this is sticky
>= 0	the returned amount of entropy (bytes)

```
40  LIB_EXPORT int32_t
41  _plat__GetEntropy(
42      unsigned char       *entropy,           // output buffer
43      uint32_t             amount             // amount requested
44      );
```

C.8.4. From Fail.c

C.8.4.1. _plat__Fail()

```
45  LIB_EXPORT NORETURN void
46  _plat__Fail(
47      void
48      );
```

C.8.5. From LocalityPlat.c

C.8.5.1. _plat__LocalityGet()

Get the most recent command locality in locality value form. This is an integer value for locality and not a locality structure The locality can be 0-4 or 32-255. 5-31 is not allowed.

```
49  LIB_EXPORT unsigned char
50  _plat__LocalityGet(
51      void
52      );
```

C.8.5.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
53  LIB_EXPORT void
54  _plat__LocalitySet(
55      unsigned char    locality
56      );
```

C.8.6. From NVMem.c

C.8.6.1. _plat__NvErrors()

This function is used by the simulator to set the error flags in the NV subsystem to simulate an error in the NV loading process

```
57  LIB_EXPORT void
58  _plat__NvErrors(
59      int              recoverable,
60      int            unrecoverable
61      );
```


C.8.6.2. _plat__NVEnable()

Enable NV memory.

This version just pulls in data from a file. In a real TPM, with NV on chip, this function would verify the integrity of the saved context. If the NV memory was not on chip but was in something like RPMB, the NV state would be read in, decrypted and integrity checked.

The recovery from an integrity failure depends on where the error occurred. It it was in the state that is discarded by TPM Reset, then the error is recoverable if the TPM is reset. Otherwise, the TPM must go into failure mode.

Return Value	Meaning
0	if success
> 0	if receive recoverable error
<0	if unrecoverable error

```
62  LIB_EXPORT int
63  _plat__NVEnable(
64      void            *platParameter  // IN: platform specific parameters
65      );
```

C.8.6.3. _plat__NVDisable()

Disable NV memory

```
66  LIB_EXPORT void
67  _plat__NVDisable(
68      void
69      );
```

C.8.6.4. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
70  LIB_EXPORT int
71  _plat__IsNvAvailable(
72      void
73      );
```

C.8.6.5. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```
74  LIB_EXPORT void
75  _plat__NvMemoryRead(
76      unsigned int     startOffset,   // IN: read start
77      unsigned int     size,          // IN: size of bytes to read
78      void            *data           // OUT: data buffer
79      );
```


C.8.6.6. _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(1)	the NV location is different from the test value
FALSE(0)	the NV location is the same as the test value

```
80  LIB_EXPORT int
81  _plat__NvIsDifferent(
82      unsigned int     startOffset,   // IN: read start
83      unsigned int     size,          // IN: size of bytes to read
84      void            *data           // IN: data buffer
85      );
```

C.8.6.7. _plat__NvMemoryWrite()

This function is used to update NV memory. The **write** is to a memory copy of NV. At the end of the current command, any changes are written to the actual NV memory.

NOTE: A useful optimization would be for this code to compare the current contents of NV with the local copy and note the blocks that have changed. Then only write those blocks when _plat__NvCommit() is called.

```
86  LIB_EXPORT void
87  _plat__NvMemoryWrite(
88      unsigned int     startOffset,   // IN: write start
89      unsigned int     size,          // IN: size of bytes to write
90      void            *data           // OUT: data buffer
91      );
```

C.8.6.8. _plat__NvMemoryClear()

Function is used to set a range of NV memory bytes to an implementation-dependent value. The value represents the erase state of the memory.

```
92  LIB_EXPORT void
93  _plat__NvMemoryClear(
94      unsigned int     start,         // IN: clear start
95      unsigned int     size           // IN: number of bytes to clear
96      );
```

C.8.6.9. _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

```
97  LIB_EXPORT void
98  _plat__NvMemoryMove(
99      unsigned int     sourceOffset,  // IN: source offset
100     unsigned int     destOffset,    // IN: destination offset
101     unsigned int     size           // IN: size of data being moved
102     );
```

C.8.6.10. _plat__NvCommit()

Update NV chip

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
103 LIB_EXPORT int
104 _plat__NvCommit(
105     void
106     );
```

C.8.6.11. _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

```
107 LIB_EXPORT void
108 _plat__SetNvAvail(
109     void
110     );
```

C.8.6.12. _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

```
111 LIB_EXPORT void
112 _plat__ClearNvAvail(
113     void
114     );
```

C.8.7. From PlatformData.c

C.8.8. From PowerPlat.c

C.8.8.1. _plat__Signal_PowerOn()

Signal platform power on

```
115 LIB_EXPORT int
116 _plat__Signal_PowerOn(
117     void
118     );
```

C.8.8.2. _plat__WasPowerLost()

Test whether power was lost before a _TPM_Init().

This function will clear the **hardware** indication of power loss before return. This means that there can only be one spot in the TPM code where this value gets read. This method is used here as it is the most difficult to manage in the TPM code and, if the hardware actually works this way, it is hard to make it look like anything else. So, the burden is placed on the TPM code rather than the platform code

Return Value	Meaning
TRUE(1)	power was lost
FALSE(0)	power was not lost

```
119  LIB_EXPORT int
120  _plat__WasPowerLost(
121      void
122      );
```

C.8.8.3. _plat_Signal_Reset()

This a TPM reset without a power loss.

```
123  LIB_EXPORT int
124  _plat__Signal_Reset(
125      void
126      );
```

C.8.8.4. _plat__Signal_PowerOff()

Signal platform power off

```
127  LIB_EXPORT void
128  _plat__Signal_PowerOff(
129      void
130      );
```

C.8.9. From PPPlat.c

C.8.10. Functions

C.8.10.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

Return Value	Meaning
TRUE(1)	if physical presence is signaled
FALSE(0)	if physical presence is not signaled

```
131  LIB_EXPORT int
132  _plat__PhysicalPresenceAsserted(
133      void
134      );
```

C.8.10.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```
135  LIB_EXPORT void
136  _plat__Signal_PhysicalPresenceOn(
137      void
138      );
```

C.8.10.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
139  LIB_EXPORT void
140  _plat__Signal_PhysicalPresenceOff(
141      void
142      );
```

C.8.11. From RunCommand.c

C.8.11.1. _plat__RunCommand()

This version of RunCommand() will set up a jum_buf and call ExecuteCommand(). If the command executes without failing, it will return and RunCommand() will return. If there is a failure in the command, then _plat__Fail() is called and it will longjump back to RunCommand() which will call ExecuteCommand() again. However, this time, the TPM will be in failure mode so ExecuteCommand() will simply build a failure response and return.

```
143  LIB_EXPORT void
144  _plat__RunCommand(
145      uint32_t         requestSize,   // IN: command buffer size
146      unsigned char   *request,       // IN: command buffer
147      uint32_t        *responseSize,  // IN/OUT: response buffer size
148      unsigned char   **response      // IN/OUT: response buffer
149      );
```

C.8.11.2. _plat__Fail()

This is the platform depended failure exit for the TPM.

```
150  LIB_EXPORT NORETURN void
151  _plat__Fail(
152      void
153      );
```

C.8.12. From Unique.c

C.8.13. _plat__GetUnique()

This function is used to access the platform-specific unique value. This function places the unique value in the provided buffer *(b*) and returns the number of bytes transferred. The function will not copy more data than *bSize*.

NOTE: If a platform unique value has unequal distribution of uniqueness and *bSize* is smaller than the size of the unique value, the *bSize* portion with the most uniqueness should be returned.

```
154  LIB_EXPORT uint32_t
155  _plat__GetUnique(
156      uint32_t             which,         // authorities (0) or details
157      uint32_t             bSize,         // size of the buffer
158      unsigned char       *b              // output buffer
159      );
160  #endif  // _PLATFORM_FP_H_
```

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.

```
1   #ifndef _PLATFORM_DATA_H_
2   #define _PLATFORM_DATA_H_
3   #include      "Implementation.h"
```

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

```
4   extern int      s_isCanceled;
5   #include    <time.h>
6   typedef struct {
7           time_t tv_sec;  // Seconds - >= 0
8           long   tv_nsec; // Nanoseconds - [0, 999999999]
9   } timespec_t;
10  #ifndef HARDWARE_CLOCK
```

This is the value returned the last time that the system clock was read. This is only relevant for a simulator or virtual TPM.

```
11  extern clock_t        s_realTimePrevious;
```

This is the rate adjusted value that is the equivalent of what would be read from a hardware register that produced rate adjusted time.

```
12  extern clock_t        s_tpmTime;
13  #endif // HARDWARE_CLOCK
```

This value indicates that the timer was reset

```
14  extern BOOL             s_timerReset;
```

This value indicates that the timer was stopped. It causes a clock discontinuity.

```
15  extern BOOL             s_timerStopped;
```

Assume that the nominal divisor is 30000

```
16  #define     CLOCK_NOMINAL           30000
```

A 1% change in rate is 300 counts

```
17  #define     CLOCK_ADJUST_COARSE     300
```

A .1 change in rate is 30 counts

```
18  #define     CLOCK_ADJUST_MEDIUM     30
```

A minimum change in rate is 1 count

```
19  #define     CLOCK_ADJUST_FINE       1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

```
20  #define     CLOCK_ADJUST_LIMIT      5000
```


This variable records the time when _plat__TimerReset() is called. This mechanism allow us to subtract the time when TPM is power off from the total time reported by clock() function

```
21  extern uint64_t        s_initClock;
```

This variable records the timer adjustment factor.

```
22  extern unsigned int         s_adjustRate;
```

From LocalityPlat.c Locality of current command

```
23  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.

```
24  #define FILE_BACKED_NV
25  #if defined FILE_BACKED_NV
26  #include <stdio.h>
```

A file to emulate NV storage

```
27  extern FILE*             s_NVFile;
28  #endif
29  extern unsigned char     s_NV[NV_MEMORY_SIZE];
30  extern BOOL              s_NvIsAvailable;
31  extern BOOL              s_NV_unrecoverable;
32  extern BOOL              s_NV_recoverable;
```

From PPPlat.c Physical presence. It is initialized to FALSE

```
33  extern BOOL     s_physicalPresence;
```

From Power

```
34  extern BOOL        s_powerLost;
```

From Entropy.c

```
35  extern uint32_t        lastEntropy;
36  extern int             firstValue;
37  #endif // _PLATFORM_DATA_H_
```

C.10 PlatformData.c

C.10.1. Description

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

C.10.2. Includes

```
#include    "Implementation.h"
#include    "PlatformData.h"
```

From Cancel.c

```
BOOL                s_isCanceled;
```

From Clock.c

```
unsigned int        s_adjustRate;
BOOL                s_timerReset;
BOOL                s_timerStopped;
#ifndef HARDWARE_CLOCK
#include    <time.h>
clock_t             s_realTimePrevious;
clock_t             s_tpmTime;
#endif
```

From LocalityPlat.c

```
unsigned char       s_locality;
```

From Power.c

```
BOOL                s_powerLost;
```

From Entropy.c

```
uint32_t            lastEntropy;
int                 firstValue;
```

From NVMem.c

```
#ifdef  VTPM
#   undef FILE_BACKED_NV
#endif
#ifdef FILE_BACKED_NV
FILE                *s_NVFile = NULL;
#endif
unsigned char       s_NV[NV_MEMORY_SIZE];
BOOL                s_NvIsAvailable;
BOOL                s_NV_unrecoverable;
BOOL                s_NV_recoverable;
```

From PPPlat.c

```
BOOL  s_physicalPresence;
```


C.11 PPPlat.c

C.11.1. Description

This module simulates the physical present interface pins on the TPM.

C.11.2. Includes

```
#include "PlatformData.h"
#include "Platform_fp.h"
```

C.11.3. Functions

C.11.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

Return Value	Meaning
TRUE(1)	if physical presence is signaled
FALSE(0)	if physical presence is not signaled

```
LIB_EXPORT int
_plat__PhysicalPresenceAsserted(
    void
    )
{
    // Do not know how to check physical presence without real hardware.
    // so always return TRUE;
    return s_physicalPresence;
}
```

C.11.3.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```
LIB_EXPORT void
_plat__Signal_PhysicalPresenceOn(
    void
    )
{
    s_physicalPresence = TRUE;
    return;
}
```

C.11.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
LIB_EXPORT void
_plat__Signal_PhysicalPresenceOff(
    void
    )
{
    s_physicalPresence = FALSE;
    return;
```



```
27 }
```

C.12 RunCommand.c

C.12.1. Introduction

This module provides the platform specific entry and fail processing. The _plat__RunCommand() function is used to call to ExecuteCommand() in the TPM code. This function does whatever processing is necessary to set up the platform in anticipation of the call to the TPM including settup for error processing.

The _plat__Fail() function is called when there is a failure in the TPM. The TPM code will have set the flag to indicate that the TPM is in failure mode. This call will then recursively call ExecuteCommand() in order to build the failure mode response. When ExecuteCommand() returns to _plat__Fail(), the platform will do some platform specif operation to return to the environment in which the TPM is executing. For a simulator, setjmp/longjmp is used. For an OS, a system exit to the OS would be appropriate.

C.12.2. Includes and locals

```
#include "PlatformData.h"
#include "Platform_fp.h"
#include <setjmp.h>
#include <ExecCommand_fp.h>
jmp_buf              s_jumpBuffer;
```

C.12.3. Functions

C.12.3.1. _plat__RunCommand()

This version of RunCommand() will set up a jum_buf and call ExecuteCommand(). If the command executes without failing, it will return and RunCommand() will return. If there is a failure in the command, then _plat__Fail() is called and it will longjump back to RunCommand() which will call ExecuteCommand() again. However, this time, the TPM will be in failure mode so ExecuteCommand() will simply build a failure response and return.

```
LIB_EXPORT void
_plat__RunCommand(
    uint32_t         requestSize,   // IN: command buffer size
    unsigned char   *request,       // IN: command buffer
    uint32_t        *responseSize,  // IN/OUT: response buffer size
    unsigned char   **response      // IN/OUT: response buffer
    )
{
    setjmp(s_jumpBuffer);
    ExecuteCommand(requestSize, request, responseSize, response);
}
```

C.12.3.2. _plat__Fail()

This is the platform depended failure exit for the TPM.

```
LIB_EXPORT NORETURN void
_plat__Fail(
    void
    )
{
    longjmp(&s_jumpBuffer[0], 1);
}
```


C.13 Unique.c

C.13.1. Introduction

In some implementations of the TPM, the hardware can provide a secret value to the TPM. This secret value is statistically unique to the instance of the TPM. Typical uses of this value are to provide personalization to the random number generation and as a shared secret between the TPM and the manufacturer.

C.13.2. Includes

```
#include "PlatformData.h"
#include "Platform_fp.h"
const char notReallyUnique[] =
"This is not really a unique value. A real unique value should"
" be generated by the platform.";
```

C.13.3. _plat__GetUnique()

This function is used to access the platform-specific unique value. This function places the unique value in the provided buffer *(b*) and returns the number of bytes transferred. The function will not copy more data than *bSize*.

NOTE: If a platform unique value has unequal distribution of uniqueness and *bSize* is smaller than the size of the unique value, the *bSize* portion with the most uniqueness should be returned.

```
LIB_EXPORT uint32_t
_plat__GetUnique(
    uint32_t             which,         // authorities (0) or details
    uint32_t             bSize,         // size of the buffer
    unsigned char       *b              // output buffer
    )
{
    const char          *from = notReallyUnique;
    uint32_t             retVal = 0;
    if(which == 0) // the authorities value
    {
        for(retVal = 0;
        *from != 0 && retVal < bSize;
            retVal++)
        {
            *b++ = *from++;
        }
    }
    else
    {
#define uSize  sizeof(notReallyUnique)
        b = &b[((bSize < uSize) ? bSize : uSize) - 1];
        for(retVal = 0;
        *from != 0 && retVal < bSize;
            retVal++)
        {
            *b-- = *from++;
        }
    }
    return retVal;
}
```


Annex D
(informative)
Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in TPM 2.0 Part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.

D.2 Simulator_fp.h

```
1   #ifndef    _SIMULATOR_FP_H_
2   #define    _SIMULATOR_FP_H_
```

D.2.1. From TcpServer.c

D.2.1.1. PlatformServer()

This function processes incoming platform requests.

```
3   BOOL
4   PlatformServer(
5       SOCKET          s
6       );
```

D.2.1.2. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

```
7   DWORD WINAPI
8   PlatformSvcRoutine(
9       LPVOID          port
10      );
```

D.2.1.3. PlatformSignalService()

This function starts a new thread waiting for platform signals. Platform signals are processed one at a time in the order in which they are received.

```
11  int
12  PlatformSignalService(
13      int             PortNumber
14      );
```

D.2.1.4. RegularCommandService()

This function services regular commands.

```
15  int
16  RegularCommandService(
17      int             PortNumber
```



```
18      );
```

D.2.1.5. StartTcpServer()

Main entry-point to the TCP server. The server listens on port specified. Note that there is no way to specify the network interface in this implementation.

```
19  int
20  StartTcpServer(
21      int             PortNumber
22      );
```

D.2.1.6. ReadBytes()

This function reads the indicated number of bytes *(NumBytes*) into buffer from the indicated socket.

```
23  BOOL
24  ReadBytes(
25      SOCKET          s,
26      char           *buffer,
27      int             NumBytes
28      );
```

D.2.1.7. WriteBytes()

This function will send the indicated number of bytes *(NumBytes*) to the indicated socket

```
29  BOOL
30  WriteBytes(
31      SOCKET          s,
32      char           *buffer,
33      int             NumBytes
34      );
```

D.2.1.8. WriteUINT32()

Send 4 bytes containing hton(1)

```
35  BOOL
36  WriteUINT32(
37      SOCKET          s,
38      uint32_t        val
39      );
```

D.2.1.9. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
40  BOOL
41  ReadVarBytes(
42      SOCKET          s,
43      char           *buffer,
44      uint32_t       *BytesReceived,
45      int             MaxLen
46      );
```

D.2.1.10. WriteVarBytes()

Send a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
47  BOOL
48  WriteVarBytes(
49      SOCKET          s,
50      char           *buffer,
51      int             BytesToSend
52      );
```

D.2.1.11. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```
53  BOOL
54  TpmServer(
55      SOCKET          s
56      );
```

D.2.2. From TPMCmdp.c

D.2.2.1. Signal_PowerOn()

This function processes a power-on indication. Among other things, it calls the _TPM_Init() handler.

```
57  void
58  _rpc__Signal_PowerOn(
59      BOOL        isReset
60      );
```

D.2.2.2. Signal_Restart()

This function processes the clock restart indication. All it does is call the platform function.

```
61  void
62  _rpc__Signal_Restart(
63      void
64      );
```

D.2.2.3. Signal_PowerOff()

This function processes the power off indication. Its primary function is to set a flag indicating that the next power on indication should cause _TPM_Init() to be called.

```
65  void
66  _rpc__Signal_PowerOff(
67      void
68      );
```

D.2.2.4. _rpc__ForceFailureMode()

This function is used to debug the Failure Mode logic of the TPM. It will set a flag in the TPM code such that the next call to TPM2_SelfTest() will result in a failure, putting the TPM into Failure Mode.


```
69  void
70  _rpc__ForceFailureMode(
71      void
72      );
```

D.2.2.5. _rpc__Signal_PhysicalPresenceOn()

This function is called to simulate activation of the physical presence **pin**.

```
73  void
74  _rpc__Signal_PhysicalPresenceOn(
75      void
76      );
```

D.2.2.6. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence **pin**.

```
77  void
78  _rpc__Signal_PhysicalPresenceOff(
79      void
80      );
```

D.2.2.7. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start() event. It will call

```
81  void
82  _rpc__Signal_Hash_Start(
83      void
84      );
```

D.2.2.8. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data() event.

```
85  void
86  _rpc__Signal_Hash_Data(
87      _IN_BUFFER       input
88      );
```

D.2.2.9. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End() event.

```
89  void
90  _rpc__Signal_HashEnd(
91      void
92      );
```

Command interface Entry of a RPC call

```
93  void
94  _rpc__Send_Command(
95      unsigned char    locality,
96      _IN_BUFFER       request,
97      _OUT_BUFFER     *response
```

```
98      );
```

D.2.2.10. _rpc__Signal_CancelOn()

This function is used to turn on the indication to cancel a command in process. An executing command is not interrupted. The command code may periodically check this indication to see if it should abort the current command processing and returned TPM_RC_CANCELLED.

```
99  void
100 _rpc__Signal_CancelOn(
101     void
102     );
```

D.2.2.11. _rpc__Signal_CancelOff()

This function is used to turn off the indication to cancel a command in process.

```
103 void
104 _rpc__Signal_CancelOff(
105     void
106     );
```

D.2.2.12. _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.

```
107 void
108 _rpc__Signal_NvOn(
109     void
110     );
```

D.2.2.13. _rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

```
111 void
112 _rpc__Signal_NvOff(
113     void
114     );
```

D.2.2.14. _rpc__RsaKeyCacheControl()

This function is used to enable/disable the use of the RSA key cache during simulation.

```
115 void
116 _rpc__RsaKeyCacheControl(
117     int             state
118     );
```

D.2.2.15. _rpc__Shutdown()

This function is used to stop the TPM simulator.

```
119 void
120 _rpc__Shutdown(
```



```
121         void
122         );
```

D.2.3. From TPMCmds.c

D.2.3.1. main()

This is the main entry point for the simulator.

```
123     int
124     main(
125         int             argc,
126         char            *argv[]
127         );
128     #endif  // _SIMULATOR_FP_H_
```

D.3 TpmTcpProtocol.h

D.3.1. 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.3.2. Typedefs and Defines

```
#ifndef     TCP_TPM_PROTOCOL_H
#define     TCP_TPM_PROTOCOL_H
```

TPM Commands. All commands acknowledge processing by returning a UINT32 == 0 except where noted

```
#define TPM_SIGNAL_POWER_ON         1
#define TPM_SIGNAL_POWER_OFF        2
#define TPM_SIGNAL_PHYS_PRES_ON     3
#define TPM_SIGNAL_PHYS_PRES_OFF    4
#define TPM_SIGNAL_HASH_START       5
#define TPM_SIGNAL_HASH_DATA        6
        // {UINT32 BufferSize, BYTE[BufferSize] Buffer}
#define TPM_SIGNAL_HASH_END         7
#define TPM_SEND_COMMAND            8
        // {BYTE Locality, UINT32 InBufferSize, BYTE[InBufferSize] InBuffer} ->
        //     {UINT32 OutBufferSize, BYTE[OutBufferSize] OutBuffer}
#define TPM_SIGNAL_CANCEL_ON        9
#define TPM_SIGNAL_CANCEL_OFF       10
#define TPM_SIGNAL_NV_ON            11
#define TPM_SIGNAL_NV_OFF           12
#define TPM_SIGNAL_KEY_CACHE_ON     13
#define TPM_SIGNAL_KEY_CACHE_OFF    14
#define TPM_REMOTE_HANDSHAKE        15
#define TPM_SET_ALTERNATIVE_RESULT  16
#define TPM_SIGNAL_RESET            17
#define TPM_SIGNAL_RESTART          18
#define TPM_SESSION_END             20
#define TPM_STOP                    21
#define TPM_GET_COMMAND_RESPONSE_SIZES  25
#define TPM_TEST_FAILURE_MODE       30
enum TpmEndPointInfo
{
    tpmPlatformAvailable = 0x01,
    tpmUsesTbs = 0x02,
    tpmInRawMode = 0x04,
    tpmSupportsPP = 0x08
};
// Existing RPC interface type definitions retained so that the implementation
// can be re-used
typedef struct in_buffer
{
    unsigned long BufferSize;
```

```
40      unsigned char *Buffer;
41  } _IN_BUFFER;
42  typedef unsigned char *_OUTPUT_BUFFER;
43  typedef struct out_buffer
44  {
45      uint32_t         BufferSize;
46      _OUTPUT_BUFFER   Buffer;
47  } _OUT_BUFFER;
48  #ifndef WIN32
49  typedef unsigned long        DWORD;
50  typedef void                *LPVOID;
51  #undef WINAPI
52  typedef
53  #endif
54  #endif
```


D.4 TcpServer.c

D.4.1. Description

This file contains the socket interface to a TPM simulator.

D.4.2. Includes, Locals, Defines and Function Prototypes

```
#include "TpmBuildSwitches.h"
#include <stdio.h>
#ifdef _MSC_VER
#include <windows.h>
#include <winsock.h>
#else
typedef SOCKET  int
#endif
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
//#include "BaseTypes.h"
#include "TpmTcpProtocol.h"
#include "Manufacture_fp.h"
#include "Simulator_fp.h"
```

To access key cache control in TPM

```
void RsaKeyCacheControl(int state);
#ifndef __IGNORE_STATE__
static uint32_t ServerVersion = 1;
#define MAX_BUFFER 1048576
char InputBuffer[MAX_BUFFER];       //The input data buffer for the simulator.
char OutputBuffer[MAX_BUFFER];      //The output data buffer for the simulator.
struct
{
    uint32_t    largestCommandSize;
    uint32_t    largestCommand;
    uint32_t    largestResponseSize;
    uint32_t    largestResponse;
} CommandResponseSizes = {0};
#endif // __IGNORE_STATE___
```

D.4.3. Functions

D.4.3.1. CreateSocket()

This function creates a socket listening on *PortNumber*.

```
static int
CreateSocket(
    int                 PortNumber,
    SOCKET              *listenSocket
    )
{
    WSADATA             wsaData;
    struct              sockaddr_in MyAddress;
    int res;
    // Initialize Winsock
    res = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if(res != 0)
    {
```

```
43          printf("WSAStartup failed with error: %d\n", res);
44          return -1;
45      }
46      // create listening socket
47      *listenSocket = socket(PF_INET, SOCK_STREAM, 0);
48      if(INVALID_SOCKET == *listenSocket)
49      {
50          printf("Cannot create server listen socket.  Error is 0x%x\n",
51                 WSAGetLastError());
52          return -1;
53      }
54      // bind the listening socket to the specified port
55      ZeroMemory(&MyAddress, sizeof(MyAddress));
56      MyAddress.sin_port = htons((short)PortNumber);
57      MyAddress.sin_family = AF_INET;
58      res = bind(*listenSocket, (struct sockaddr*) &MyAddress, sizeof(MyAddress));
59      if(res == SOCKET_ERROR)
60      {
61          printf("Bind error.  Error is 0x%x\n", WSAGetLastError());
62          return -1;
63      };
64      // listen/wait for server connections
65      res = listen(*listenSocket, 3);
66      if(res == SOCKET_ERROR)
67      {
68          printf("Listen error.  Error is 0x%x\n", WSAGetLastError());
69          return -1;
70      };
71      return 0;
72  }
```

D.4.3.2. PlatformServer()

This function processes incoming platform requests.

```
73  BOOL
74  PlatformServer(
75      SOCKET          s
76      )
77  {
78      BOOL                OK = TRUE;
79      uint32_t            Command;
80      for(;;)
81      {
82          OK = ReadBytes(s, (char*)&Command, 4);
83          // client disconnected (or other error).  We stop processing this client
84          // and return to our caller who can stop the server or listen for another
85          // connection.
86          if(!OK) return TRUE;
87          Command = ntohl(Command);
88          switch(Command)
89          {
90              case TPM_SIGNAL_POWER_ON:
91                  _rpc__Signal_PowerOn(FALSE);
92                  break;
93              case TPM_SIGNAL_POWER_OFF:
94                  _rpc__Signal_PowerOff();
95                  break;
96              case TPM_SIGNAL_RESET:
97                  _rpc__Signal_PowerOn(TRUE);
98                  break;
99              case TPM_SIGNAL_RESTART:
100                 _rpc__Signal_Restart();
101                 break;
```



```
102             case TPM_SIGNAL_PHYS_PRES_ON:
103                 _rpc__Signal_PhysicalPresenceOn();
104                 break;
105             case TPM_SIGNAL_PHYS_PRES_OFF:
106                 _rpc__Signal_PhysicalPresenceOff();
107                 break;
108             case TPM_SIGNAL_CANCEL_ON:
109                 _rpc__Signal_CancelOn();
110                 break;
111             case TPM_SIGNAL_CANCEL_OFF:
112                 _rpc__Signal_CancelOff();
113                 break;
114             case TPM_SIGNAL_NV_ON:
115                 _rpc__Signal_NvOn();
116                 break;
117             case TPM_SIGNAL_NV_OFF:
118                 _rpc__Signal_NvOff();
119                 break;
120             case TPM_SIGNAL_KEY_CACHE_ON:
121                 _rpc__RsaKeyCacheControl(TRUE);
122                 break;
123             case TPM_SIGNAL_KEY_CACHE_OFF:
124                 _rpc__RsaKeyCacheControl(FALSE);
125                 break;
126             case TPM_SESSION_END:
127                 // Client signaled end-of-session
128                 TpmEndSimulation();
129                 return TRUE;
130             case TPM_STOP:
131                 // Client requested the simulator to exit
132                 return FALSE;
133             case TPM_TEST_FAILURE_MODE:
134                 _rpc__ForceFailureMode();
135                 break;
136             case TPM_GET_COMMAND_RESPONSE_SIZES:
137                 OK = WriteVarBytes(s, (char *)&CommandResponseSizes,
138                                    sizeof(CommandResponseSizes));
139                 memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
140                 if(!OK)
141                     return TRUE;
142                 break;
143             default:
144                 printf("Unrecognized platform interface command %d\n",
145                        (int)Command);
146                 WriteUINT32(s, 1);
147                 return TRUE;
148         }
149         WriteUINT32(s, 0);
150     }
151     return FALSE;
152 }
```

D.4.3.3. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

```
153 DWORD WINAPI
154 PlatformSvcRoutine(
155     LPVOID          port
156     )
157 {
158     int                 PortNumber = (int)(INT_PTR)port;
159     SOCKET              listenSocket, serverSocket;
160     struct              sockaddr_in HerAddress;
```

```
161         int                 res;
162         int                 length;
163         BOOL                continueServing;
164         res = CreateSocket(PortNumber, &listenSocket);
165         if(res != 0)
166         {
167             printf("Create platform service socket fail\n");
168             return res;
169         }
170         // Loop accepting connections one-by-one until we are killed or asked to stop
171         // Note the platform service is single-threaded so we don't listen for a new
172         // connection until the prior connection drops.
173         do
174         {
175             printf("Platform server listening on port %d\n", PortNumber);
176             // blocking accept
177             length = sizeof(HerAddress);
178             serverSocket = accept(listenSocket,
179                                   (struct sockaddr*) &HerAddress,
180                                   &length);
181             if(serverSocket == SOCKET_ERROR)
182             {
183                 printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
184                 return -1;
185             };
186             printf("Client accepted\n");
187             // normal behavior on client disconnection is to wait for a new client
188             // to connect
189             continueServing = PlatformServer(serverSocket);
190             closesocket(serverSocket);
191         } while(continueServing);
192         return 0;
193     }
```

D.4.3.4. PlatformSignalService()

This function starts a new thread waiting for platform signals. Platform signals are processed one at a time in the order in which they are received.

```
194     int
195     PlatformSignalService(
196         int             PortNumber
197         )
198     {
199         HANDLE              hPlatformSvc;
200         int                 ThreadId;
201         int                 port = PortNumber;
202         // Create service thread for platform signals
203         hPlatformSvc = CreateThread(NULL, 0,
204                                     (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
205                                     (LPVOID)(INT_PTR)port, 0, (LPDWORD)&ThreadId);
206         if(hPlatformSvc == NULL)
207         {
208             printf("Thread Creation failed\n");
209             return -1;
210         }
211         return 0;
212     }
```

D.4.3.5. RegularCommandService()

This function services regular commands.

```
213 int
214 RegularCommandService(
215     int             PortNumber
216     )
217 {
218     SOCKET              listenSocket;
219     SOCKET              serverSocket;
220     struct              sockaddr_in HerAddress;
221     int res, length;
222     BOOL continueServing;
223     res = CreateSocket(PortNumber, &listenSocket);
224     if(res != 0)
225     {
226         printf("Create platform service socket fail\n");
227         return res;
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         // blocking accept
236         length = sizeof(HerAddress);
237         serverSocket = accept(listenSocket,
238                               (struct sockaddr*) &HerAddress,
239                               &length);
240         if(serverSocket == SOCKET_ERROR)
241         {
242             printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
243             return -1;
244         };
245         printf("Client accepted\n");
246         // normal behavior on client disconnection is to wait for a new client
247         // to connect
248         continueServing = TpmServer(serverSocket);
249         closesocket(serverSocket);
250     } while(continueServing);
251     return 0;
252 }
```

D.4.3.6. StartTcpServer()

Main entry-point to the TCP server. The server listens on port specified. Note that there is no way to specify the network interface in this implementation.

```
253 int
254 StartTcpServer(
255     int             PortNumber
256     )
257 {
258     int                 res;
259     // Start Platform Signal Processing Service
260     res = PlatformSignalService(PortNumber + 1);
261     if(res != 0)
262     {
263         printf("PlatformSignalService failed\n");
264         return res;
265     }
266     // Start Regular/DRTM TPM command service
267     res = RegularCommandService(PortNumber);
268     if(res != 0)
269     {
270         printf("RegularCommandService failed\n");
```

```
271             return res;
272         }
273         return 0;
274     }
```

D.4.3.7. ReadBytes()

This function reads the indicated number of bytes *(NumBytes*) into buffer from the indicated socket.

```
275     BOOL
276     ReadBytes(
277         SOCKET          s,
278         char           *buffer,
279         int             NumBytes
280         )
281     {
282         int                 res;
283         int                 numGot = 0;
284         while(numGot < NumBytes)
285         {
286             res = recv(s, buffer + numGot, NumBytes - numGot, 0);
287             if(res == -1)
288             {
289                 printf("Receive error.  Error is 0x%x\n", WSAGetLastError());
290                 return FALSE;
291             }
292             if(res == 0)
293             {
294                 return FALSE;
295             }
296             numGot += res;
297         }
298         return TRUE;
299     }
```

D.4.3.8. WriteBytes()

This function will send the indicated number of bytes *(NumBytes*) to the indicated socket

```
300     BOOL
301     WriteBytes(
302         SOCKET          s,
303         char           *buffer,
304         int             NumBytes
305         )
306     {
307         int                 res;
308         int                 numSent = 0;
309         while(numSent < NumBytes)
310         {
311             res = send(s, buffer + numSent, NumBytes - numSent, 0);
312             if(res == -1)
313             {
314                 if(WSAGetLastError() == 0x2745)
315                 {
316                     printf("Client disconnected\n");
317                 }
318                 else
319                 {
320                     printf("Send error.  Error is 0x%x\n", WSAGetLastError());
321                 }
322                 return FALSE;
323             }
```

```
324             numSent += res;
325         }
326         return TRUE;
327     }
```

D.4.3.9. WriteUINT32()

Send 4 bytes containing hton(1)

```
328     BOOL
329     WriteUINT32(
330         SOCKET          s,
331         uint32_t        val
332         )
333     {
334         UINT32 netVal = htonl(val);
335         return WriteBytes(s, (char*)&netVal, 4);
336     }
```

D.4.3.10. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
337     BOOL
338     ReadVarBytes(
339         SOCKET          s,
340         char            *buffer,
341         uint32_t        *BytesReceived,
342         int             MaxLen
343         )
344     {
345         int                 length;
346         BOOL                res;
347         res = ReadBytes(s, (char*)&length, 4);
348         if(!res) return res;
349         length = ntohl(length);
350         *BytesReceived = length;
351         if(length > MaxLen)
352         {
353             printf("Buffer too big.  Client says %d\n", length);
354             return FALSE;
355         }
356         if(length == 0) return TRUE;
357         res = ReadBytes(s, buffer, length);
358         if(!res) return res;
359         return TRUE;
360     }
```

D.4.3.11. WriteVarBytes()

Send a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```
361     BOOL
362     WriteVarBytes(
363         SOCKET          s,
364         char            *buffer,
365         int             BytesToSend
366         )
367     {
```

```
368         uint32_t            netLength = htonl(BytesToSend);
369         BOOL res;
370         res = WriteBytes(s, (char*)&netLength, 4);
371         if(!res) return res;
372         res = WriteBytes(s, buffer, BytesToSend);
373         if(!res) return res;
374         return TRUE;
375     }
```

D.4.3.12. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```
376     BOOL
377     TpmServer(
378         SOCKET          s
379         )
380     {
381         uint32_t            length;
382         uint32_t            Command;
383         BYTE                locality;
384         BOOL                OK;
385         int                 result;
386         int                 clientVersion;
387         _IN_BUFFER          InBuffer;
388         _OUT_BUFFER         OutBuffer;
389         for(;;)
390         {
391             OK = ReadBytes(s, (char*)&Command, 4);
392             // client disconnected (or other error).  We stop processing this client
393             // and return to our caller who can stop the server or listen for another
394             // connection.
395             if(!OK)
396                 return TRUE;
397             Command = ntohl(Command);
398             switch(Command)
399             {
400                 case TPM_SIGNAL_HASH_START:
401                     _rpc__Signal_Hash_Start();
402                     break;
403                 case TPM_SIGNAL_HASH_END:
404                     _rpc__Signal_HashEnd();
405                     break;
406                 case TPM_SIGNAL_HASH_DATA:
407                     OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
408                     if(!OK) return TRUE;
409                     InBuffer.Buffer = (BYTE*)InputBuffer;
410                     InBuffer.BufferSize = length;
411                     _rpc__Signal_Hash_Data(InBuffer);
412                     break;
413                 case TPM_SEND_COMMAND:
414                     OK = ReadBytes(s, (char*)&locality, 1);
415                     if(!OK)
416                         return TRUE;
417                     OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
418                     if(!OK)
419                         return TRUE;
420                     InBuffer.Buffer = (BYTE*)InputBuffer;
421                     InBuffer.BufferSize = length;
422                     OutBuffer.BufferSize = MAX_BUFFER;
423                     OutBuffer.Buffer = (_OUTPUT_BUFFER)OutputBuffer;
424                     // record the number of bytes in the command if it is the largest
425                     // we have seen so far.
426                     if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
```

```
427                 {
428                     CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
429                     memcpy(&CommandResponseSizes.largestCommand,
430                            &InputBuffer[6], sizeof(UINT32));
431                 }
432                 _rpc__Send_Command(locality, InBuffer, &OutBuffer);
433                 // record the number of bytes in the response if it is the largest
434                 // we have seen so far.
435                 if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
436                 {
437                     CommandResponseSizes.largestResponseSize
438                         = OutBuffer.BufferSize;
439                     memcpy(&CommandResponseSizes.largestResponse,
440                            &OutputBuffer[6], sizeof(UINT32));
441                 }
442                 OK = WriteVarBytes(s,
443                                    (char*)OutBuffer.Buffer,
444                                    OutBuffer.BufferSize);
445                 if(!OK)
446                     return TRUE;
447                 break;
448             case TPM_REMOTE_HANDSHAKE:
449                 OK = ReadBytes(s, (char*)&clientVersion, 4);
450                 if(!OK)
451                     return TRUE;
452                 if(clientVersion == 0)
453                 {
454                     printf("Unsupported client version (0).\n");
455                     return TRUE;
456                 }
457                 OK &= WriteUINT32(s, ServerVersion);
458                 OK &= WriteUINT32(s, tpmInRawMode
459                                   | tpmPlatformAvailable | tpmSupportsPP);
460                 break;
461             case TPM_SET_ALTERNATIVE_RESULT:
462                 OK = ReadBytes(s, (char*)&result, 4);
463                 if(!OK)
464                     return TRUE;
465                 // Alternative result is not applicable to the simulator.
466                 break;
467             case TPM_SESSION_END:
468                 // Client signaled end-of-session
469                 return TRUE;
470             case TPM_STOP:
471                 // Client requested the simulator to exit
472                 return FALSE;
473             default:
474                 printf("Unrecognized TPM interface command %d\n", (int)Command);
475                 return TRUE;
476         }
477         OK = WriteUINT32(s, 0);
478         if(!OK)
479             return TRUE;
480     }
481     return FALSE;
482 }
```


D.5 TPMCmdp.c

D.5.1. 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.5.2. Includes and Data Definitions

```
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>
#include "TpmBuildSwitches.h"
#include <windows.h>
#include <winsock.h>
#include "Platform_fp.h"
#include "ExecCommand_fp.h"
#include "Manufacture_fp.h"
#include "_TPM_Init_fp.h"
#include "_TPM_Hash_Start_fp.h"
#include "_TPM_Hash_Data_fp.h"
#include "_TPM_Hash_End_fp.h"
#include "TpmFail_fp.h"
#include "TpmTcpProtocol.h"
#include "Simulator_fp.h"
static BOOL     s_isPowerOn = FALSE;
```

D.5.3. Functions

D.5.3.1. Signal_PowerOn()

This function processes a power-on indication. Among other things, it calls the _TPM_Init() handler.

```
void
_rpc__Signal_PowerOn(
    BOOL        isReset
    )
{
    // if power is on and this is not a call to do TPM reset then return
    if(s_isPowerOn && !isReset)
        return;
    // If this is a reset but power is not on, then return
    if(isReset && !s_isPowerOn)
        return;
    // Unless this is just a reset, pass power on signal to platform
    if(!isReset)
        _plat__Signal_PowerOn();
    // Power on and reset both lead to _TPM_Init()
    _plat__Signal_Reset();
    // Set state as power on
    s_isPowerOn = TRUE;
}
```

D.5.3.2. Signal_Restart()

This function processes the clock restart indication. All it does is call the platform function.

```
void
_rpc__Signal_Restart(
    void
    )
{
    _plat__TimerRestart();
}
```

D.5.3.3. Signal_PowerOff()

This function processes the power off indication. Its primary function is to set a flag indicating that the next power on indication should cause _TPM_Init() to be called.

```
void
_rpc__Signal_PowerOff(
    void
    )
{
    if(!s_isPowerOn) return;
    // Pass power off signal to platform
    _plat__Signal_PowerOff();
    s_isPowerOn = FALSE;
    return;
}
```

D.5.3.4. _rpc__ForceFailureMode()

This function is used to debug the Failure Mode logic of the TPM. It will set a flag in the TPM code such that the next call to TPM2_SelfTest() will result in a failure, putting the TPM into Failure Mode.

```
void
_rpc__ForceFailureMode(
    void
    )
{
    SetForceFailureMode();
}
```

D.5.3.5. _rpc__Signal_PhysicalPresenceOn()

This function is called to simulate activation of the physical presence **pin**.

```
void
_rpc__Signal_PhysicalPresenceOn(
    void
    )
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;
    // Pass physical presence on to platform
    _plat__Signal_PhysicalPresenceOn();
    return;
}
```

D.5.3.6. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence **pin**.

```
void
```



```
74  _rpc__Signal_PhysicalPresenceOff(
75      void
76      )
77  {
78      // If TPM is power off, reject this signal
79      if(!s_isPowerOn) return;
80      // Pass physical presence off to platform
81      _plat__Signal_PhysicalPresenceOff();
82      return;
83  }
```

D.5.3.7. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start() event. It will call

```
84  void
85  _rpc__Signal_Hash_Start(
86      void
87      )
88  {
89      // If TPM is power off, reject this signal
90      if(!s_isPowerOn) return;
91      // Pass _TPM_Hash_Start signal to TPM
92      _TPM_Hash_Start();
93      return;
94  }
```

D.5.3.8. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data() event.

```
95  void
96  _rpc__Signal_Hash_Data(
97      _IN_BUFFER       input
98      )
99  {
100     // If TPM is power off, reject this signal
101     if(!s_isPowerOn) return;
102     // Pass _TPM_Hash_Data signal to TPM
103     _TPM_Hash_Data(input.BufferSize, input.Buffer);
104     return;
105 }
```

D.5.3.9. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End() event.

```
106 void
107 _rpc__Signal_HashEnd(
108     void
109     )
110 {
111     // If TPM is power off, reject this signal
112     if(!s_isPowerOn) return;
113     // Pass _TPM_HashEnd signal to TPM
114     _TPM_Hash_End();
115     return;
116 }
```

Command interface Entry of a RPC call


```
117  void
118  _rpc__Send_Command(
119      unsigned char    locality,
120      _IN_BUFFER       request,
121      _OUT_BUFFER     *response
122      )
123  {
124      // If TPM is power off, reject any commands.
125      if(!s_isPowerOn)
126      {
127          response->BufferSize = 0;
128          return;
129      }
130      // Set the locality of the command so that it doesn't change during the command
131      _plat__LocalitySet(locality);
132      // Do implementation-specific command dispatch
133      _plat__RunCommand(request.BufferSize, request.Buffer,
134                        &response->BufferSize, &response->Buffer);
135      return;
136  }
```

D.5.3.10. _rpc__Signal_CancelOn()

This function is used to turn on the indication to cancel a command in process. An executing command is not interrupted. The command code may periodically check this indication to see if it should abort the current command processing and returned TPM_RC_CANCELLED.

```
137  void
138  _rpc__Signal_CancelOn(
139      void
140      )
141  {
142      // If TPM is power off, reject this signal
143      if(!s_isPowerOn) return;
144      // Set the platform canceling flag.
145      _plat__SetCancel();
146      return;
147  }
```

D.5.3.11. _rpc__Signal_CancelOff()

This function is used to turn off the indication to cancel a command in process.

```
148  void
149  _rpc__Signal_CancelOff(
150      void
151      )
152  {
153      // If TPM is power off, reject this signal
154      if(!s_isPowerOn) return;
155      // Set the platform canceling flag.
156      _plat__ClearCancel();
157      return;
158  }
```

D.5.3.12. _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.

```
159  void
```



```
160 _rpc__Signal_NvOn(
161     void
162     )
163 {
164     // If TPM is power off, reject this signal
165     if(!s_isPowerOn) return;
166     _plat__SetNvAvail();
167     return;
168 }
```

D.5.3.13. _rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

```
169 void
170 _rpc__Signal_NvOff(
171     void
172     )
173 {
174     // If TPM is power off, reject this signal
175     if(!s_isPowerOn) return;
176     _plat__ClearNvAvail();
177     return;
178 }
179 void RsaKeyCacheControl(int state);
```

D.5.3.14. _rpc__RsaKeyCacheControl()

This function is used to enable/disable the use of the RSA key cache during simulation.

```
180 void
181 _rpc__RsaKeyCacheControl(
182     int             state
183     )
184 {
185 #ifdef USE_RSA_KEY_CACHE
186     RsaKeyCacheControl(state);
187 #endif
188 }
```

D.5.3.15. _rpc__Shutdown()

This function is used to stop the TPM simulator.

```
189 void
190 _rpc__Shutdown(
191     void
192     )
193 {
194     RPC_STATUS status;
195     // Stop TPM
196     TPM_TearDown();
197     status = RpcMgmtStopServerListening(NULL);
198     if(status != RPC_S_OK)
199     {
200         printf("RpcMgmtStopServerListening returned: 0x%x\n", status);
201         exit(status);
202     }
203     status = RpcServerUnregisterIf(NULL, NULL, FALSE);
204     if(status != RPC_S_OK)
205     {
```

```
206             printf("RpcServerUnregisterIf returned 0x%x\n", status);
207             exit(status);
208         }
209         return;
210     }
```

D.6 TPMCmds.c

D.6.1. Description

This file contains the entry point for the simulator.

D.6.2. Includes, Defines, Data Definitions, and Function Prototypes

```
#include "TpmBuildSwitches.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <ctype.h>
#include <string.h>
#include <windows.h>
#include <winsock.h>
#include "TpmTcpProtocol.h"
#include "Manufacture_fp.h"
#include "Platform_fp.h"
#include "Simulator_fp.h"
#define PURPOSE \
"TPM Reference Simulator.\nCopyright Microsoft 2010, 2011.\n"
#define DEFAULT_TPM_PORT 2321
void* MainPointer;
```

D.6.3. Functions

D.6.3.1. Usage()

This function prints the proper calling sequence for the simulator.

```
static void
Usage(
    char                *pszProgramName
    )
{
    fprintf(stderr, "%s", PURPOSE);
    fprintf(stderr, "Usage:\n");
    fprintf(stderr, "%s          - Starts the TPM server listening on port %d\n",
              pszProgramName, DEFAULT_TPM_PORT);
    fprintf(stderr,
              "%s PortNum - Starts the TPM server listening on port PortNum\n",
              pszProgramName);
    fprintf(stderr, "%s ?       - This message\n", pszProgramName);
    exit(1);
}
```

D.6.3.2. main()

This is the main entry point for the simulator.

main: register the interface, start listening for clients

```
int
main(
    int              argc,
    char            *argv[]
    )
{
```



```
38      int portNum = DEFAULT_TPM_PORT;
39      if(argc > 2)
40      {
41          Usage(argv[0]);
42      }
43      if(argc == 2)
44      {
45          if(strcmp(argv[1], "?") == 0)
46          {
47              Usage(argv[0]);
48          }
49          portNum = atoi(argv[1]);
50          if(portNum <= 0 || portNum > 65535)
51          {
52              Usage(argv[0]);
53          }
54      }
55      _plat__NVEnable(NULL);
56      if(TPM_Manufacture(1) != 0)
57      {
58          exit(1);
59      }
60      // Coverage test - repeated manufacturing attempt
61      if(TPM_Manufacture(0) != 1)
62      {
63          exit(2);
64      }
65      // Coverage test - re-manufacturing
66      TPM_TearDown();
67      if(TPM_Manufacture(1) != 0)
68      {
69          exit(3);
70      }
71      // Disable NV memory
72      _plat__NVDisable();
73      StartTcpServer(portNum);
74      return EXIT_SUCCESS;
75  }
```

