Trusted Platform Module Library
Part 4: Supporting Routines

Family "2.0"

Level 00 Revision 01.50

September 18, 2018

Committee Draft

Contact: admin@trustedcomputinggroup.org

Work in Progress

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

TCG Public Review

TCG

Licenses and Notices

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 4.4.

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_FP_H_
3   #define _CREATE_FP_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_FP_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

```
#ifndef _BASETYPES_H
#define _BASETYPES_H
```

NULL definition

```
#ifndef NULL
#define NULL                (0)
#endif
typedef uint8_t             UINT8;
typedef uint8_t             BYTE;
typedef int8_t              INT8;
typedef int                 BOOL;
typedef uint16_t            UINT16;
typedef int16_t             INT16;
typedef uint32_t            UINT32;
typedef int32_t             INT32;
typedef uint64_t            UINT64;
typedef int64_t             INT64;
#endif // _BASETYPES_H
```

5.3 Capabilities.h

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

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

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

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

5.4 CommandAttributeData.h

This file should only be included by CommandCodeAttibutes.c

```
1   #ifdef _COMMAND_CODE_ATTRIBUTES_
2   #include "CommandAttributes.h"
3   #if COMPRESSED_LISTS
4   #   define      PAD_LIST    0
5   #else
6   #   define      PAD_LIST    1
7   #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

```
8   const TPMA_CC    s_ccAttr [] = {
9   #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
10          TPMA_CC_INITIALIZER(0x011F, 0, 1, 0, 0, 2, 0, 0, 0),
11  #endif
12  #if (PAD_LIST || CC_EvictControl)
13          TPMA_CC_INITIALIZER(0x0120, 0, 1, 0, 0, 2, 0, 0, 0),
14  #endif
15  #if (PAD_LIST || CC_HierarchyControl)
16          TPMA_CC_INITIALIZER(0x0121, 0, 1, 1, 0, 1, 0, 0, 0),
17  #endif
18  #if (PAD_LIST || CC_NV_UndefineSpace)
19          TPMA_CC_INITIALIZER(0x0122, 0, 1, 0, 0, 2, 0, 0, 0),
20  #endif
21  #if (PAD_LIST )
22          TPMA_CC_INITIALIZER(0x0123, 0, 0, 0, 0, 0, 0, 0, 0),
23  #endif
24  #if (PAD_LIST || CC_ChangeEPS)
25          TPMA_CC_INITIALIZER(0x0124, 0, 1, 1, 0, 1, 0, 0, 0),
26  #endif
27  #if (PAD_LIST || CC_ChangePPS)
28          TPMA_CC_INITIALIZER(0x0125, 0, 1, 1, 0, 1, 0, 0, 0),
29  #endif
30  #if (PAD_LIST || CC_Clear)
31          TPMA_CC_INITIALIZER(0x0126, 0, 1, 1, 0, 1, 0, 0, 0),
32  #endif
33  #if (PAD_LIST || CC_ClearControl)
34          TPMA_CC_INITIALIZER(0x0127, 0, 1, 0, 0, 1, 0, 0, 0),
35  #endif
36  #if (PAD_LIST || CC_ClockSet)
37          TPMA_CC_INITIALIZER(0x0128, 0, 1, 0, 0, 1, 0, 0, 0),
38  #endif
39  #if (PAD_LIST || CC_HierarchyChangeAuth)
40          TPMA_CC_INITIALIZER(0x0129, 0, 1, 0, 0, 1, 0, 0, 0),
41  #endif
42  #if (PAD_LIST || CC_NV_DefineSpace)
43          TPMA_CC_INITIALIZER(0x012A, 0, 1, 0, 0, 1, 0, 0, 0),
44  #endif
45  #if (PAD_LIST || CC_PCR_Allocate)
46          TPMA_CC_INITIALIZER(0x012B, 0, 1, 0, 0, 1, 0, 0, 0),
47  #endif
48  #if (PAD_LIST || CC_PCR_SetAuthPolicy)
49          TPMA_CC_INITIALIZER(0x012C, 0, 1, 0, 0, 1, 0, 0, 0),
50  #endif
51  #if (PAD_LIST || CC_PP_Commands)
52          TPMA_CC_INITIALIZER(0x012D, 0, 1, 0, 0, 1, 0, 0, 0),
53  #endif
54  #if (PAD_LIST || CC_SetPrimaryPolicy)
55          TPMA_CC_INITIALIZER(0x012E, 0, 1, 0, 0, 1, 0, 0, 0),
56  #endif
```

```
57   #if (PAD_LIST || CC_FieldUpgradeStart)
58           TPMA_CC_INITIALIZER(0x012F, 0, 0, 0, 0, 2, 0, 0, 0),
59   #endif
60   #if (PAD_LIST || CC_ClockRateAdjust)
61           TPMA_CC_INITIALIZER(0x0130, 0, 0, 0, 0, 1, 0, 0, 0),
62   #endif
63   #if (PAD_LIST || CC_CreatePrimary)
64           TPMA_CC_INITIALIZER(0x0131, 0, 0, 0, 0, 1, 1, 0, 0),
65   #endif
66   #if (PAD_LIST || CC_NV_GlobalWriteLock)
67           TPMA_CC_INITIALIZER(0x0132, 0, 1, 0, 0, 1, 0, 0, 0),
68   #endif
69   #if (PAD_LIST || CC_GetCommandAuditDigest)
70           TPMA_CC_INITIALIZER(0x0133, 0, 1, 0, 0, 2, 0, 0, 0),
71   #endif
72   #if (PAD_LIST || CC_NV_Increment)
73           TPMA_CC_INITIALIZER(0x0134, 0, 1, 0, 0, 2, 0, 0, 0),
74   #endif
75   #if (PAD_LIST || CC_NV_SetBits)
76           TPMA_CC_INITIALIZER(0x0135, 0, 1, 0, 0, 2, 0, 0, 0),
77   #endif
78   #if (PAD_LIST || CC_NV_Extend)
79           TPMA_CC_INITIALIZER(0x0136, 0, 1, 0, 0, 2, 0, 0, 0),
80   #endif
81   #if (PAD_LIST || CC_NV_Write)
82           TPMA_CC_INITIALIZER(0x0137, 0, 1, 0, 0, 2, 0, 0, 0),
83   #endif
84   #if (PAD_LIST || CC_NV_WriteLock)
85           TPMA_CC_INITIALIZER(0x0138, 0, 1, 0, 0, 2, 0, 0, 0),
86   #endif
87   #if (PAD_LIST || CC_DictionaryAttackLockReset)
88           TPMA_CC_INITIALIZER(0x0139, 0, 1, 0, 0, 1, 0, 0, 0),
89   #endif
90   #if (PAD_LIST || CC_DictionaryAttackParameters)
91           TPMA_CC_INITIALIZER(0x013A, 0, 1, 0, 0, 1, 0, 0, 0),
92   #endif
93   #if (PAD_LIST || CC_NV_ChangeAuth)
94           TPMA_CC_INITIALIZER(0x013B, 0, 1, 0, 0, 1, 0, 0, 0),
95   #endif
96   #if (PAD_LIST || CC_PCR_Event)
97           TPMA_CC_INITIALIZER(0x013C, 0, 1, 0, 0, 1, 0, 0, 0),
98   #endif
99   #if (PAD_LIST || CC_PCR_Reset)
100          TPMA_CC_INITIALIZER(0x013D, 0, 1, 0, 0, 1, 0, 0, 0),
101  #endif
102  #if (PAD_LIST || CC_SequenceComplete)
103          TPMA_CC_INITIALIZER(0x013E, 0, 0, 0, 1, 1, 0, 0, 0),
104  #endif
105  #if (PAD_LIST || CC_SetAlgorithmSet)
106          TPMA_CC_INITIALIZER(0x013F, 0, 1, 0, 0, 1, 0, 0, 0),
107  #endif
108  #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
109          TPMA_CC_INITIALIZER(0x0140, 0, 1, 0, 0, 1, 0, 0, 0),
110  #endif
111  #if (PAD_LIST || CC_FieldUpgradeData)
112          TPMA_CC_INITIALIZER(0x0141, 0, 1, 0, 0, 0, 0, 0, 0),
113  #endif
114  #if (PAD_LIST || CC_IncrementalSelfTest)
115          TPMA_CC_INITIALIZER(0x0142, 0, 1, 0, 0, 0, 0, 0, 0),
116  #endif
117  #if (PAD_LIST || CC_SelfTest)
118          TPMA_CC_INITIALIZER(0x0143, 0, 1, 0, 0, 0, 0, 0, 0),
119  #endif
120  #if (PAD_LIST || CC_Startup)
121          TPMA_CC_INITIALIZER(0x0144, 0, 1, 0, 0, 0, 0, 0, 0),
122  #endif
```



```
123   #if (PAD_LIST || CC_Shutdown)
124           TPMA_CC_INITIALIZER(0x0145, 0, 1, 0, 0, 0, 0, 0, 0),
125   #endif
126   #if (PAD_LIST || CC_StirRandom)
127           TPMA_CC_INITIALIZER(0x0146, 0, 1, 0, 0, 0, 0, 0, 0),
128   #endif
129   #if (PAD_LIST || CC_ActivateCredential)
130           TPMA_CC_INITIALIZER(0x0147, 0, 0, 0, 0, 2, 0, 0, 0),
131   #endif
132   #if (PAD_LIST || CC_Certify)
133           TPMA_CC_INITIALIZER(0x0148, 0, 0, 0, 0, 2, 0, 0, 0),
134   #endif
135   #if (PAD_LIST || CC_PolicyNV)
136           TPMA_CC_INITIALIZER(0x0149, 0, 0, 0, 0, 3, 0, 0, 0),
137   #endif
138   #if (PAD_LIST || CC_CertifyCreation)
139           TPMA_CC_INITIALIZER(0x014A, 0, 0, 0, 0, 2, 0, 0, 0),
140   #endif
141   #if (PAD_LIST || CC_Duplicate)
142           TPMA_CC_INITIALIZER(0x014B, 0, 0, 0, 0, 2, 0, 0, 0),
143   #endif
144   #if (PAD_LIST || CC_GetTime)
145           TPMA_CC_INITIALIZER(0x014C, 0, 0, 0, 0, 2, 0, 0, 0),
146   #endif
147   #if (PAD_LIST || CC_GetSessionAuditDigest)
148           TPMA_CC_INITIALIZER(0x014D, 0, 0, 0, 0, 3, 0, 0, 0),
149   #endif
150   #if (PAD_LIST || CC_NV_Read)
151           TPMA_CC_INITIALIZER(0x014E, 0, 0, 0, 0, 2, 0, 0, 0),
152   #endif
153   #if (PAD_LIST || CC_NV_ReadLock)
154           TPMA_CC_INITIALIZER(0x014F, 0, 1, 0, 0, 2, 0, 0, 0),
155   #endif
156   #if (PAD_LIST || CC_ObjectChangeAuth)
157           TPMA_CC_INITIALIZER(0x0150, 0, 0, 0, 0, 2, 0, 0, 0),
158   #endif
159   #if (PAD_LIST || CC_PolicySecret)
160           TPMA_CC_INITIALIZER(0x0151, 0, 0, 0, 0, 2, 0, 0, 0),
161   #endif
162   #if (PAD_LIST || CC_Rewrap)
163           TPMA_CC_INITIALIZER(0x0152, 0, 0, 0, 0, 2, 0, 0, 0),
164   #endif
165   #if (PAD_LIST || CC_Create)
166           TPMA_CC_INITIALIZER(0x0153, 0, 0, 0, 0, 1, 0, 0, 0),
167   #endif
168   #if (PAD_LIST || CC_ECDH_ZGen)
169           TPMA_CC_INITIALIZER(0x0154, 0, 0, 0, 0, 1, 0, 0, 0),
170   #endif
171   #if (PAD_LIST || (CC_HMAC || CC_MAC))
172           TPMA_CC_INITIALIZER(0x0155, 0, 0, 0, 0, 1, 0, 0, 0),
173   #endif
174   #if (PAD_LIST || CC_Import)
175           TPMA_CC_INITIALIZER(0x0156, 0, 0, 0, 0, 1, 0, 0, 0),
176   #endif
177   #if (PAD_LIST || CC_Load)
178           TPMA_CC_INITIALIZER(0x0157, 0, 0, 0, 0, 1, 1, 0, 0),
179   #endif
180   #if (PAD_LIST || CC_Quote)
181           TPMA_CC_INITIALIZER(0x0158, 0, 0, 0, 0, 1, 0, 0, 0),
182   #endif
183   #if (PAD_LIST || CC_RSA_Decrypt)
184           TPMA_CC_INITIALIZER(0x0159, 0, 0, 0, 0, 1, 0, 0, 0),
185   #endif
186   #if (PAD_LIST )
187           TPMA_CC_INITIALIZER(0x015A, 0, 0, 0, 0, 0, 0, 0, 0),
188   #endif
```

```
189   #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
190           TPMA_CC_INITIALIZER(0x015B, 0, 0, 0, 0, 1, 1, 0, 0),
191   #endif
192   #if (PAD_LIST || CC_SequenceUpdate)
193           TPMA_CC_INITIALIZER(0x015C, 0, 0, 0, 0, 1, 0, 0, 0),
194   #endif
195   #if (PAD_LIST || CC_Sign)
196           TPMA_CC_INITIALIZER(0x015D, 0, 0, 0, 0, 1, 0, 0, 0),
197   #endif
198   #if (PAD_LIST || CC_Unseal)
199           TPMA_CC_INITIALIZER(0x015E, 0, 0, 0, 0, 1, 0, 0, 0),
200   #endif
201   #if (PAD_LIST )
202           TPMA_CC_INITIALIZER(0x015F, 0, 0, 0, 0, 0, 0, 0, 0),
203   #endif
204   #if (PAD_LIST || CC_PolicySigned)
205           TPMA_CC_INITIALIZER(0x0160, 0, 0, 0, 0, 2, 0, 0, 0),
206   #endif
207   #if (PAD_LIST || CC_ContextLoad)
208           TPMA_CC_INITIALIZER(0x0161, 0, 0, 0, 0, 0, 1, 0, 0),
209   #endif
210   #if (PAD_LIST || CC_ContextSave)
211           TPMA_CC_INITIALIZER(0x0162, 0, 0, 0, 0, 1, 0, 0, 0),
212   #endif
213   #if (PAD_LIST || CC_ECDH_KeyGen)
214           TPMA_CC_INITIALIZER(0x0163, 0, 0, 0, 0, 1, 0, 0, 0),
215   #endif
216   #if (PAD_LIST || CC_EncryptDecrypt)
217           TPMA_CC_INITIALIZER(0x0164, 0, 0, 0, 0, 1, 0, 0, 0),
218   #endif
219   #if (PAD_LIST || CC_FlushContext)
220           TPMA_CC_INITIALIZER(0x0165, 0, 0, 0, 0, 0, 0, 0, 0),
221   #endif
222   #if (PAD_LIST )
223           TPMA_CC_INITIALIZER(0x0166, 0, 0, 0, 0, 0, 0, 0, 0),
224   #endif
225   #if (PAD_LIST || CC_LoadExternal)
226           TPMA_CC_INITIALIZER(0x0167, 0, 0, 0, 0, 0, 1, 0, 0),
227   #endif
228   #if (PAD_LIST || CC_MakeCredential)
229           TPMA_CC_INITIALIZER(0x0168, 0, 0, 0, 0, 1, 0, 0, 0),
230   #endif
231   #if (PAD_LIST || CC_NV_ReadPublic)
232           TPMA_CC_INITIALIZER(0x0169, 0, 0, 0, 0, 1, 0, 0, 0),
233   #endif
234   #if (PAD_LIST || CC_PolicyAuthorize)
235           TPMA_CC_INITIALIZER(0x016A, 0, 0, 0, 0, 1, 0, 0, 0),
236   #endif
237   #if (PAD_LIST || CC_PolicyAuthValue)
238           TPMA_CC_INITIALIZER(0x016B, 0, 0, 0, 0, 1, 0, 0, 0),
239   #endif
240   #if (PAD_LIST || CC_PolicyCommandCode)
241           TPMA_CC_INITIALIZER(0x016C, 0, 0, 0, 0, 1, 0, 0, 0),
242   #endif
243   #if (PAD_LIST || CC_PolicyCounterTimer)
244           TPMA_CC_INITIALIZER(0x016D, 0, 0, 0, 0, 1, 0, 0, 0),
245   #endif
246   #if (PAD_LIST || CC_PolicyCpHash)
247           TPMA_CC_INITIALIZER(0x016E, 0, 0, 0, 0, 1, 0, 0, 0),
248   #endif
249   #if (PAD_LIST || CC_PolicyLocality)
250           TPMA_CC_INITIALIZER(0x016F, 0, 0, 0, 0, 1, 0, 0, 0),
251   #endif
252   #if (PAD_LIST || CC_PolicyNameHash)
253           TPMA_CC_INITIALIZER(0x0170, 0, 0, 0, 0, 1, 0, 0, 0),
254   #endif
```



```
255   #if (PAD_LIST || CC_PolicyOR)
256           TPMA_CC_INITIALIZER(0x0171, 0, 0, 0, 0, 1, 0, 0, 0),
257   #endif
258   #if (PAD_LIST || CC_PolicyTicket)
259           TPMA_CC_INITIALIZER(0x0172, 0, 0, 0, 0, 1, 0, 0, 0),
260   #endif
261   #if (PAD_LIST || CC_ReadPublic)
262           TPMA_CC_INITIALIZER(0x0173, 0, 0, 0, 0, 1, 0, 0, 0),
263   #endif
264   #if (PAD_LIST || CC_RSA_Encrypt)
265           TPMA_CC_INITIALIZER(0x0174, 0, 0, 0, 0, 1, 0, 0, 0),
266   #endif
267   #if (PAD_LIST )
268           TPMA_CC_INITIALIZER(0x0175, 0, 0, 0, 0, 0, 0, 0, 0),
269   #endif
270   #if (PAD_LIST || CC_StartAuthSession)
271           TPMA_CC_INITIALIZER(0x0176, 0, 0, 0, 0, 2, 1, 0, 0),
272   #endif
273   #if (PAD_LIST || CC_VerifySignature)
274           TPMA_CC_INITIALIZER(0x0177, 0, 0, 0, 0, 1, 0, 0, 0),
275   #endif
276   #if (PAD_LIST || CC_ECC_Parameters)
277           TPMA_CC_INITIALIZER(0x0178, 0, 0, 0, 0, 0, 0, 0, 0),
278   #endif
279   #if (PAD_LIST || CC_FirmwareRead)
280           TPMA_CC_INITIALIZER(0x0179, 0, 0, 0, 0, 0, 0, 0, 0),
281   #endif
282   #if (PAD_LIST || CC_GetCapability)
283           TPMA_CC_INITIALIZER(0x017A, 0, 0, 0, 0, 0, 0, 0, 0),
284   #endif
285   #if (PAD_LIST || CC_GetRandom)
286           TPMA_CC_INITIALIZER(0x017B, 0, 0, 0, 0, 0, 0, 0, 0),
287   #endif
288   #if (PAD_LIST || CC_GetTestResult)
289           TPMA_CC_INITIALIZER(0x017C, 0, 0, 0, 0, 0, 0, 0, 0),
290   #endif
291   #if (PAD_LIST || CC_Hash)
292           TPMA_CC_INITIALIZER(0x017D, 0, 0, 0, 0, 0, 0, 0, 0),
293   #endif
294   #if (PAD_LIST || CC_PCR_Read)
295           TPMA_CC_INITIALIZER(0x017E, 0, 0, 0, 0, 0, 0, 0, 0),
296   #endif
297   #if (PAD_LIST || CC_PolicyPCR)
298           TPMA_CC_INITIALIZER(0x017F, 0, 0, 0, 0, 1, 0, 0, 0),
299   #endif
300   #if (PAD_LIST || CC_PolicyRestart)
301           TPMA_CC_INITIALIZER(0x0180, 0, 0, 0, 0, 1, 0, 0, 0),
302   #endif
303   #if (PAD_LIST || CC_ReadClock)
304           TPMA_CC_INITIALIZER(0x0181, 0, 0, 0, 0, 0, 0, 0, 0),
305   #endif
306   #if (PAD_LIST || CC_PCR_Extend)
307           TPMA_CC_INITIALIZER(0x0182, 0, 1, 0, 0, 1, 0, 0, 0),
308   #endif
309   #if (PAD_LIST || CC_PCR_SetAuthValue)
310           TPMA_CC_INITIALIZER(0x0183, 0, 0, 0, 0, 1, 0, 0, 0),
311   #endif
312   #if (PAD_LIST || CC_NV_Certify)
313           TPMA_CC_INITIALIZER(0x0184, 0, 0, 0, 0, 3, 0, 0, 0),
314   #endif
315   #if (PAD_LIST || CC_EventSequenceComplete)
316           TPMA_CC_INITIALIZER(0x0185, 0, 1, 0, 1, 2, 0, 0, 0),
317   #endif
318   #if (PAD_LIST || CC_HashSequenceStart)
319           TPMA_CC_INITIALIZER(0x0186, 0, 0, 0, 0, 0, 1, 0, 0),
320   #endif
```

```
321  #if (PAD_LIST || CC_PolicyPhysicalPresence)
322          TPMA_CC_INITIALIZER(0x0187, 0, 0, 0, 0, 1, 0, 0, 0),
323  #endif
324  #if (PAD_LIST || CC_PolicyDuplicationSelect)
325          TPMA_CC_INITIALIZER(0x0188, 0, 0, 0, 0, 1, 0, 0, 0),
326  #endif
327  #if (PAD_LIST || CC_PolicyGetDigest)
328          TPMA_CC_INITIALIZER(0x0189, 0, 0, 0, 0, 1, 0, 0, 0),
329  #endif
330  #if (PAD_LIST || CC_TestParms)
331          TPMA_CC_INITIALIZER(0x018A, 0, 0, 0, 0, 0, 0, 0, 0),
332  #endif
333  #if (PAD_LIST || CC_Commit)
334          TPMA_CC_INITIALIZER(0x018B, 0, 0, 0, 0, 1, 0, 0, 0),
335  #endif
336  #if (PAD_LIST || CC_PolicyPassword)
337          TPMA_CC_INITIALIZER(0x018C, 0, 0, 0, 0, 1, 0, 0, 0),
338  #endif
339  #if (PAD_LIST || CC_ZGen_2Phase)
340          TPMA_CC_INITIALIZER(0x018D, 0, 0, 0, 0, 1, 0, 0, 0),
341  #endif
342  #if (PAD_LIST || CC_EC_Ephemeral)
343          TPMA_CC_INITIALIZER(0x018E, 0, 0, 0, 0, 0, 0, 0, 0),
344  #endif
345  #if (PAD_LIST || CC_PolicyNvWritten)
346          TPMA_CC_INITIALIZER(0x018F, 0, 0, 0, 0, 1, 0, 0, 0),
347  #endif
348  #if (PAD_LIST || CC_PolicyTemplate)
349          TPMA_CC_INITIALIZER(0x0190, 0, 0, 0, 0, 1, 0, 0, 0),
350  #endif
351  #if (PAD_LIST || CC_CreateLoaded)
352          TPMA_CC_INITIALIZER(0x0191, 0, 0, 0, 0, 1, 1, 0, 0),
353  #endif
354  #if (PAD_LIST || CC_PolicyAuthorizeNV)
355          TPMA_CC_INITIALIZER(0x0192, 0, 0, 0, 0, 3, 0, 0, 0),
356  #endif
357  #if (PAD_LIST || CC_EncryptDecrypt2)
358          TPMA_CC_INITIALIZER(0x0193, 0, 0, 0, 0, 1, 0, 0, 0),
359  #endif
360  #if (PAD_LIST || CC_AC_GetCapability)
361          TPMA_CC_INITIALIZER(0x0194, 0, 0, 0, 0, 1, 0, 0, 0),
362  #endif
363  #if (PAD_LIST || CC_AC_Send)
364          TPMA_CC_INITIALIZER(0x0195, 0, 0, 0, 0, 3, 0, 0, 0),
365  #endif
366  #if (PAD_LIST || CC_Policy_AC_SendSelect)
367          TPMA_CC_INITIALIZER(0x0196, 0, 0, 0, 0, 1, 0, 0, 0),
368  #endif
369  #if (PAD_LIST || CC_Vendor_TCG_Test)
370          TPMA_CC_INITIALIZER(0x0000, 0, 0, 0, 0, 0, 0, 1, 0),
371  #endif
372          {0}
373  };
```

This is the command code attribute structure.

```
374  const COMMAND_ATTRIBUTES    s_commandAttributes [] = {
375  #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
376          (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpaceSpecial     *  // 0x011F
377              (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),
378  #endif
379  #if (PAD_LIST || CC_EvictControl)
380          (COMMAND_ATTRIBUTES)(CC_EvictControl                *  // 0x0120
381              (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
382  #endif
```



```
383   #if (PAD_LIST || CC_HierarchyControl)
384           (COMMAND_ATTRIBUTES)(CC_HierarchyControl            *  // 0x0121
385               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
386   #endif
387   #if (PAD_LIST || CC_NV_UndefineSpace)
388           (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpace            *  // 0x0122
389               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
390   #endif
391   #if (PAD_LIST )
392           (COMMAND_ATTRIBUTES)(0),                               // 0x0123
393   #endif
394   #if (PAD_LIST || CC_ChangeEPS)
395           (COMMAND_ATTRIBUTES)(CC_ChangeEPS                   *  // 0x0124
396               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
397   #endif
398   #if (PAD_LIST || CC_ChangePPS)
399           (COMMAND_ATTRIBUTES)(CC_ChangePPS                   *  // 0x0125
400               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
401   #endif
402   #if (PAD_LIST || CC_Clear)
403           (COMMAND_ATTRIBUTES)(CC_Clear                       *  // 0x0126
404               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
405   #endif
406   #if (PAD_LIST || CC_ClearControl)
407           (COMMAND_ATTRIBUTES)(CC_ClearControl                *  // 0x0127
408               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
409   #endif
410   #if (PAD_LIST || CC_ClockSet)
411           (COMMAND_ATTRIBUTES)(CC_ClockSet                    *  // 0x0128
412               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
413   #endif
414   #if (PAD_LIST || CC_HierarchyChangeAuth)
415           (COMMAND_ATTRIBUTES)(CC_HierarchyChangeAuth         *  // 0x0129
416               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
417   #endif
418   #if (PAD_LIST || CC_NV_DefineSpace)
419           (COMMAND_ATTRIBUTES)(CC_NV_DefineSpace              *  // 0x012A
420               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
421   #endif
422   #if (PAD_LIST || CC_PCR_Allocate)
423           (COMMAND_ATTRIBUTES)(CC_PCR_Allocate                *  // 0x012B
424               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
425   #endif
426   #if (PAD_LIST || CC_PCR_SetAuthPolicy)
427           (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthPolicy           *  // 0x012C
428               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
429   #endif
430   #if (PAD_LIST || CC_PP_Commands)
431           (COMMAND_ATTRIBUTES)(CC_PP_Commands                 *  // 0x012D
432               (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),
433   #endif
434   #if (PAD_LIST || CC_SetPrimaryPolicy)
435           (COMMAND_ATTRIBUTES)(CC_SetPrimaryPolicy            *  // 0x012E
436               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
437   #endif
438   #if (PAD_LIST || CC_FieldUpgradeStart)
439           (COMMAND_ATTRIBUTES)(CC_FieldUpgradeStart           *  // 0x012F
440               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),
441   #endif
442   #if (PAD_LIST || CC_ClockRateAdjust)
443           (COMMAND_ATTRIBUTES)(CC_ClockRateAdjust             *  // 0x0130
444               (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
445   #endif
446   #if (PAD_LIST || CC_CreatePrimary)
447           (COMMAND_ATTRIBUTES)(CC_CreatePrimary               *  // 0x0131
448               (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
```

```
449  #endif
450  #if (PAD_LIST || CC_NV_GlobalWriteLock)
451          (COMMAND_ATTRIBUTES)(CC_NV_GlobalWriteLock          *  // 0x0132
452              (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
453  #endif
454  #if (PAD_LIST || CC_GetCommandAuditDigest)
455          (COMMAND_ATTRIBUTES)(CC_GetCommandAuditDigest       *  // 0x0133
456              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
457  #endif
458  #if (PAD_LIST || CC_NV_Increment)
459          (COMMAND_ATTRIBUTES)(CC_NV_Increment                *  // 0x0134
460              (IS_IMPLEMENTED+HANDLE_1_USER)),
461  #endif
462  #if (PAD_LIST || CC_NV_SetBits)
463          (COMMAND_ATTRIBUTES)(CC_NV_SetBits                  *  // 0x0135
464              (IS_IMPLEMENTED+HANDLE_1_USER)),
465  #endif
466  #if (PAD_LIST || CC_NV_Extend)
467          (COMMAND_ATTRIBUTES)(CC_NV_Extend                   *  // 0x0136
468              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
469  #endif
470  #if (PAD_LIST || CC_NV_Write)
471          (COMMAND_ATTRIBUTES)(CC_NV_Write                    *  // 0x0137
472              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
473  #endif
474  #if (PAD_LIST || CC_NV_WriteLock)
475          (COMMAND_ATTRIBUTES)(CC_NV_WriteLock                *  // 0x0138
476              (IS_IMPLEMENTED+HANDLE_1_USER)),
477  #endif
478  #if (PAD_LIST || CC_DictionaryAttackLockReset)
479          (COMMAND_ATTRIBUTES)(CC_DictionaryAttackLockReset   *  // 0x0139
480              (IS_IMPLEMENTED+HANDLE_1_USER)),
481  #endif
482  #if (PAD_LIST || CC_DictionaryAttackParameters)
483          (COMMAND_ATTRIBUTES)(CC_DictionaryAttackParameters  *  // 0x013A
484              (IS_IMPLEMENTED+HANDLE_1_USER)),
485  #endif
486  #if (PAD_LIST || CC_NV_ChangeAuth)
487          (COMMAND_ATTRIBUTES)(CC_NV_ChangeAuth               *  // 0x013B
488              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),
489  #endif
490  #if (PAD_LIST || CC_PCR_Event)
491          (COMMAND_ATTRIBUTES)(CC_PCR_Event                   *  // 0x013C
492              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
493  #endif
494  #if (PAD_LIST || CC_PCR_Reset)
495          (COMMAND_ATTRIBUTES)(CC_PCR_Reset                   *  // 0x013D
496              (IS_IMPLEMENTED+HANDLE_1_USER)),
497  #endif
498  #if (PAD_LIST || CC_SequenceComplete)
499          (COMMAND_ATTRIBUTES)(CC_SequenceComplete            *  // 0x013E
500              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
501  #endif
502  #if (PAD_LIST || CC_SetAlgorithmSet)
503          (COMMAND_ATTRIBUTES)(CC_SetAlgorithmSet             *  // 0x013F
504              (IS_IMPLEMENTED+HANDLE_1_USER)),
505  #endif
506  #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
507          (COMMAND_ATTRIBUTES)(CC_SetCommandCodeAuditStatus   *  // 0x0140
508              (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
509  #endif
510  #if (PAD_LIST || CC_FieldUpgradeData)
511          (COMMAND_ATTRIBUTES)(CC_FieldUpgradeData            *  // 0x0141
512              (IS_IMPLEMENTED+DECRYPT_2)),
513  #endif
514  #if (PAD_LIST || CC_IncrementalSelfTest)
```



```
515         (COMMAND_ATTRIBUTES)(CC_IncrementalSelfTest         *  // 0x0142
516             (IS_IMPLEMENTED)),
517 #endif
518 #if (PAD_LIST || CC_SelfTest)
519         (COMMAND_ATTRIBUTES)(CC_SelfTest                    *  // 0x0143
520             (IS_IMPLEMENTED)),
521 #endif
522 #if (PAD_LIST || CC_Startup)
523         (COMMAND_ATTRIBUTES)(CC_Startup                     *  // 0x0144
524             (IS_IMPLEMENTED+NO_SESSIONS)),
525 #endif
526 #if (PAD_LIST || CC_Shutdown)
527         (COMMAND_ATTRIBUTES)(CC_Shutdown                    *  // 0x0145
528             (IS_IMPLEMENTED)),
529 #endif
530 #if (PAD_LIST || CC_StirRandom)
531         (COMMAND_ATTRIBUTES)(CC_StirRandom                  *  // 0x0146
532             (IS_IMPLEMENTED+DECRYPT_2)),
533 #endif
534 #if (PAD_LIST || CC_ActivateCredential)
535         (COMMAND_ATTRIBUTES)(CC_ActivateCredential          *  // 0x0147
536             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
537 #endif
538 #if (PAD_LIST || CC_Certify)
539         (COMMAND_ATTRIBUTES)(CC_Certify                     *  // 0x0148
540             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
541 #endif
542 #if (PAD_LIST || CC_PolicyNV)
543         (COMMAND_ATTRIBUTES)(CC_PolicyNV                    *  // 0x0149
544             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL)),
545 #endif
546 #if (PAD_LIST || CC_CertifyCreation)
547         (COMMAND_ATTRIBUTES)(CC_CertifyCreation             *  // 0x014A
548             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
549 #endif
550 #if (PAD_LIST || CC_Duplicate)
551         (COMMAND_ATTRIBUTES)(CC_Duplicate                   *  // 0x014B
552             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),
553 #endif
554 #if (PAD_LIST || CC_GetTime)
555         (COMMAND_ATTRIBUTES)(CC_GetTime                     *  // 0x014C
556             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
557 #endif
558 #if (PAD_LIST || CC_GetSessionAuditDigest)
559         (COMMAND_ATTRIBUTES)(CC_GetSessionAuditDigest       *  // 0x014D
560             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
561 #endif
562 #if (PAD_LIST || CC_NV_Read)
563         (COMMAND_ATTRIBUTES)(CC_NV_Read                     *  // 0x014E
564             (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
565 #endif
566 #if (PAD_LIST || CC_NV_ReadLock)
567         (COMMAND_ATTRIBUTES)(CC_NV_ReadLock                 *  // 0x014F
568             (IS_IMPLEMENTED+HANDLE_1_USER)),
569 #endif
570 #if (PAD_LIST || CC_ObjectChangeAuth)
571         (COMMAND_ATTRIBUTES)(CC_ObjectChangeAuth            *  // 0x0150
572             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),
573 #endif
574 #if (PAD_LIST || CC_PolicySecret)
575         (COMMAND_ATTRIBUTES)(CC_PolicySecret                *  // 0x0151
576             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL+ENCRYPT_2)),
577 #endif
578 #if (PAD_LIST || CC_Rewrap)
579         (COMMAND_ATTRIBUTES)(CC_Rewrap                      *  // 0x0152
580             (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
```

```
581  #endif
582  #if (PAD_LIST || CC_Create)
583          (COMMAND_ATTRIBUTES)(CC_Create                        *  // 0x0153
584              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
585  #endif
586  #if (PAD_LIST || CC_ECDH_ZGen)
587          (COMMAND_ATTRIBUTES)(CC_ECDH_ZGen                     *  // 0x0154
588              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
589  #endif
590  #if (PAD_LIST || (CC_HMAC || CC_MAC))
591          (COMMAND_ATTRIBUTES)((CC_HMAC || CC_MAC)              *  // 0x0155
592              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
593  #endif
594  #if (PAD_LIST || CC_Import)
595          (COMMAND_ATTRIBUTES)(CC_Import                        *  // 0x0156
596              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
597  #endif
598  #if (PAD_LIST || CC_Load)
599          (COMMAND_ATTRIBUTES)(CC_Load                          *  // 0x0157
600              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),
601  #endif
602  #if (PAD_LIST || CC_Quote)
603          (COMMAND_ATTRIBUTES)(CC_Quote                         *  // 0x0158
604              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
605  #endif
606  #if (PAD_LIST || CC_RSA_Decrypt)
607          (COMMAND_ATTRIBUTES)(CC_RSA_Decrypt                   *  // 0x0159
608              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
609  #endif
610  #if (PAD_LIST )
611          (COMMAND_ATTRIBUTES)(0),                                 // 0x015A
612  #endif
613  #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
614          (COMMAND_ATTRIBUTES)((CC_HMAC_Start || CC_MAC_Start) *  // 0x015B
615              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),
616  #endif
617  #if (PAD_LIST || CC_SequenceUpdate)
618          (COMMAND_ATTRIBUTES)(CC_SequenceUpdate                *  // 0x015C
619              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
620  #endif
621  #if (PAD_LIST || CC_Sign)
622          (COMMAND_ATTRIBUTES)(CC_Sign                          *  // 0x015D
623              (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
624  #endif
625  #if (PAD_LIST || CC_Unseal)
626          (COMMAND_ATTRIBUTES)(CC_Unseal                        *  // 0x015E
627              (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
628  #endif
629  #if (PAD_LIST )
630          (COMMAND_ATTRIBUTES)(0),                                 // 0x015F
631  #endif
632  #if (PAD_LIST || CC_PolicySigned)
633          (COMMAND_ATTRIBUTES)(CC_PolicySigned                  *  // 0x0160
634              (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL+ENCRYPT_2)),
635  #endif
636  #if (PAD_LIST || CC_ContextLoad)
637          (COMMAND_ATTRIBUTES)(CC_ContextLoad                   *  // 0x0161
638              (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
639  #endif
640  #if (PAD_LIST || CC_ContextSave)
641          (COMMAND_ATTRIBUTES)(CC_ContextSave                   *  // 0x0162
642              (IS_IMPLEMENTED+NO_SESSIONS)),
643  #endif
644  #if (PAD_LIST || CC_ECDH_KeyGen)
645          (COMMAND_ATTRIBUTES)(CC_ECDH_KeyGen                   *  // 0x0163
646              (IS_IMPLEMENTED+ENCRYPT_2)),
```



```
647  #endif
648  #if (PAD_LIST || CC_EncryptDecrypt)
649          (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt                *  // 0x0164
650              (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
651  #endif
652  #if (PAD_LIST || CC_FlushContext)
653          (COMMAND_ATTRIBUTES)(CC_FlushContext                  *  // 0x0165
654              (IS_IMPLEMENTED+NO_SESSIONS)),
655  #endif
656  #if (PAD_LIST )
657          (COMMAND_ATTRIBUTES)(0),                                 // 0x0166
658  #endif
659  #if (PAD_LIST || CC_LoadExternal)
660          (COMMAND_ATTRIBUTES)(CC_LoadExternal                  *  // 0x0167
661              (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
662  #endif
663  #if (PAD_LIST || CC_MakeCredential)
664          (COMMAND_ATTRIBUTES)(CC_MakeCredential                *  // 0x0168
665              (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
666  #endif
667  #if (PAD_LIST || CC_NV_ReadPublic)
668          (COMMAND_ATTRIBUTES)(CC_NV_ReadPublic                 *  // 0x0169
669              (IS_IMPLEMENTED+ENCRYPT_2)),
670  #endif
671  #if (PAD_LIST || CC_PolicyAuthorize)
672          (COMMAND_ATTRIBUTES)(CC_PolicyAuthorize               *  // 0x016A
673              (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
674  #endif
675  #if (PAD_LIST || CC_PolicyAuthValue)
676          (COMMAND_ATTRIBUTES)(CC_PolicyAuthValue               *  // 0x016B
677              (IS_IMPLEMENTED+ALLOW_TRIAL)),
678  #endif
679  #if (PAD_LIST || CC_PolicyCommandCode)
680          (COMMAND_ATTRIBUTES)(CC_PolicyCommandCode             *  // 0x016C
681              (IS_IMPLEMENTED+ALLOW_TRIAL)),
682  #endif
683  #if (PAD_LIST || CC_PolicyCounterTimer)
684          (COMMAND_ATTRIBUTES)(CC_PolicyCounterTimer            *  // 0x016D
685              (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
686  #endif
687  #if (PAD_LIST || CC_PolicyCpHash)
688          (COMMAND_ATTRIBUTES)(CC_PolicyCpHash                  *  // 0x016E
689              (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
690  #endif
691  #if (PAD_LIST || CC_PolicyLocality)
692          (COMMAND_ATTRIBUTES)(CC_PolicyLocality                *  // 0x016F
693              (IS_IMPLEMENTED+ALLOW_TRIAL)),
694  #endif
695  #if (PAD_LIST || CC_PolicyNameHash)
696          (COMMAND_ATTRIBUTES)(CC_PolicyNameHash                *  // 0x0170
697              (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
698  #endif
699  #if (PAD_LIST || CC_PolicyOR)
700          (COMMAND_ATTRIBUTES)(CC_PolicyOR                      *  // 0x0171
701              (IS_IMPLEMENTED+ALLOW_TRIAL)),
702  #endif
703  #if (PAD_LIST || CC_PolicyTicket)
704          (COMMAND_ATTRIBUTES)(CC_PolicyTicket                  *  // 0x0172
705              (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
706  #endif
707  #if (PAD_LIST || CC_ReadPublic)
708          (COMMAND_ATTRIBUTES)(CC_ReadPublic                    *  // 0x0173
709              (IS_IMPLEMENTED+ENCRYPT_2)),
710  #endif
711  #if (PAD_LIST || CC_RSA_Encrypt)
712          (COMMAND_ATTRIBUTES)(CC_RSA_Encrypt                   *  // 0x0174
```

```
713                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
714     #endif
715     #if (PAD_LIST )
716             (COMMAND_ATTRIBUTES)(0),                                // 0x0175
717     #endif
718     #if (PAD_LIST || CC_StartAuthSession)
719             (COMMAND_ATTRIBUTES)(CC_StartAuthSession            *  // 0x0176
720                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
721     #endif
722     #if (PAD_LIST || CC_VerifySignature)
723             (COMMAND_ATTRIBUTES)(CC_VerifySignature             *  // 0x0177
724                 (IS_IMPLEMENTED+DECRYPT_2)),
725     #endif
726     #if (PAD_LIST || CC_ECC_Parameters)
727             (COMMAND_ATTRIBUTES)(CC_ECC_Parameters              *  // 0x0178
728                 (IS_IMPLEMENTED)),
729     #endif
730     #if (PAD_LIST || CC_FirmwareRead)
731             (COMMAND_ATTRIBUTES)(CC_FirmwareRead                *  // 0x0179
732                 (IS_IMPLEMENTED+ENCRYPT_2)),
733     #endif
734     #if (PAD_LIST || CC_GetCapability)
735             (COMMAND_ATTRIBUTES)(CC_GetCapability               *  // 0x017A
736                 (IS_IMPLEMENTED)),
737     #endif
738     #if (PAD_LIST || CC_GetRandom)
739             (COMMAND_ATTRIBUTES)(CC_GetRandom                   *  // 0x017B
740                 (IS_IMPLEMENTED+ENCRYPT_2)),
741     #endif
742     #if (PAD_LIST || CC_GetTestResult)
743             (COMMAND_ATTRIBUTES)(CC_GetTestResult               *  // 0x017C
744                 (IS_IMPLEMENTED+ENCRYPT_2)),
745     #endif
746     #if (PAD_LIST || CC_Hash)
747             (COMMAND_ATTRIBUTES)(CC_Hash                        *  // 0x017D
748                 (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
749     #endif
750     #if (PAD_LIST || CC_PCR_Read)
751             (COMMAND_ATTRIBUTES)(CC_PCR_Read                    *  // 0x017E
752                 (IS_IMPLEMENTED)),
753     #endif
754     #if (PAD_LIST || CC_PolicyPCR)
755             (COMMAND_ATTRIBUTES)(CC_PolicyPCR                   *  // 0x017F
756                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
757     #endif
758     #if (PAD_LIST || CC_PolicyRestart)
759             (COMMAND_ATTRIBUTES)(CC_PolicyRestart               *  // 0x0180
760                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
761     #endif
762     #if (PAD_LIST || CC_ReadClock)
763             (COMMAND_ATTRIBUTES)(CC_ReadClock                   *  // 0x0181
764                 (IS_IMPLEMENTED)),
765     #endif
766     #if (PAD_LIST || CC_PCR_Extend)
767             (COMMAND_ATTRIBUTES)(CC_PCR_Extend                  *  // 0x0182
768                 (IS_IMPLEMENTED+HANDLE_1_USER)),
769     #endif
770     #if (PAD_LIST || CC_PCR_SetAuthValue)
771             (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthValue            *  // 0x0183
772                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
773     #endif
774     #if (PAD_LIST || CC_NV_Certify)
775             (COMMAND_ATTRIBUTES)(CC_NV_Certify                  *  // 0x0184
776                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
777     #endif
778     #if (PAD_LIST || CC_EventSequenceComplete)
```



```
779             (COMMAND_ATTRIBUTES)(CC_EventSequenceComplete       *  // 0x0185
780                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),
781     #endif
782     #if (PAD_LIST || CC_HashSequenceStart)
783             (COMMAND_ATTRIBUTES)(CC_HashSequenceStart           *  // 0x0186
784                 (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
785     #endif
786     #if (PAD_LIST || CC_PolicyPhysicalPresence)
787             (COMMAND_ATTRIBUTES)(CC_PolicyPhysicalPresence      *  // 0x0187
788                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
789     #endif
790     #if (PAD_LIST || CC_PolicyDuplicationSelect)
791             (COMMAND_ATTRIBUTES)(CC_PolicyDuplicationSelect     *  // 0x0188
792                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
793     #endif
794     #if (PAD_LIST || CC_PolicyGetDigest)
795             (COMMAND_ATTRIBUTES)(CC_PolicyGetDigest             *  // 0x0189
796                 (IS_IMPLEMENTED+ALLOW_TRIAL+ENCRYPT_2)),
797     #endif
798     #if (PAD_LIST || CC_TestParms)
799             (COMMAND_ATTRIBUTES)(CC_TestParms                   *  // 0x018A
800                 (IS_IMPLEMENTED)),
801     #endif
802     #if (PAD_LIST || CC_Commit)
803             (COMMAND_ATTRIBUTES)(CC_Commit                      *  // 0x018B
804                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
805     #endif
806     #if (PAD_LIST || CC_PolicyPassword)
807             (COMMAND_ATTRIBUTES)(CC_PolicyPassword              *  // 0x018C
808                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
809     #endif
810     #if (PAD_LIST || CC_ZGen_2Phase)
811             (COMMAND_ATTRIBUTES)(CC_ZGen_2Phase                 *  // 0x018D
812                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
813     #endif
814     #if (PAD_LIST || CC_EC_Ephemeral)
815             (COMMAND_ATTRIBUTES)(CC_EC_Ephemeral                *  // 0x018E
816                 (IS_IMPLEMENTED+ENCRYPT_2)),
817     #endif
818     #if (PAD_LIST || CC_PolicyNvWritten)
819             (COMMAND_ATTRIBUTES)(CC_PolicyNvWritten             *  // 0x018F
820                 (IS_IMPLEMENTED+ALLOW_TRIAL)),
821     #endif
822     #if (PAD_LIST || CC_PolicyTemplate)
823             (COMMAND_ATTRIBUTES)(CC_PolicyTemplate              *  // 0x0190
824                 (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
825     #endif
826     #if (PAD_LIST || CC_CreateLoaded)
827             (COMMAND_ATTRIBUTES)(CC_CreateLoaded                *  // 0x0191
828                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
829     #endif
830     #if (PAD_LIST || CC_PolicyAuthorizeNV)
831             (COMMAND_ATTRIBUTES)(CC_PolicyAuthorizeNV           *  // 0x0192
832                 (IS_IMPLEMENTED+HANDLE_1_USER+ALLOW_TRIAL)),
833     #endif
834     #if (PAD_LIST || CC_EncryptDecrypt2)
835             (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt2             *  // 0x0193
836                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
837     #endif
838     #if (PAD_LIST || CC_AC_GetCapability)
839             (COMMAND_ATTRIBUTES)(CC_AC_GetCapability            *  // 0x0194
840                 (IS_IMPLEMENTED)),
841     #endif
842     #if (PAD_LIST || CC_AC_Send)
843             (COMMAND_ATTRIBUTES)(CC_AC_Send                     *  // 0x0195
844                 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+HANDLE_2_USER)),
```

```
845    #endif
846    #if (PAD_LIST || CC_Policy_AC_SendSelect)
847            (COMMAND_ATTRIBUTES)(CC_Policy_AC_SendSelect        *  // 0x0196
848                (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
849    #endif
850    #if (PAD_LIST || CC_Vendor_TCG_Test)
851            (COMMAND_ATTRIBUTES)(CC_Vendor_TCG_Test             *  // 0x0000
852                (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
853    #endif
854            0
855    };
856    #endif  // _COMMAND_CODE_ATTRIBUTES_
```

5.5 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
```

5.6 CommandDispatchData.h

This file should only be included by CommandCodeAttibutes.c

```
1   #ifdef _COMMAND_TABLE_DISPATCH_
```

Define the stop value

```
2   #define END_OF_LIST     0xff
3   #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

```
4   const UNMARSHAL_t UnmarshalArray[] = {
5   #define TPMI_DH_CONTEXT_H_UNMARSHAL         0
6               (UNMARSHAL_t)TPMI_DH_CONTEXT_Unmarshal,
7   #define TPMI_RH_AC_H_UNMARSHAL              (TPMI_DH_CONTEXT_H_UNMARSHAL + 1)
8               (UNMARSHAL_t)TPMI_RH_AC_Unmarshal,
9   #define TPMI_RH_CLEAR_H_UNMARSHAL           (TPMI_RH_AC_H_UNMARSHAL + 1)
10              (UNMARSHAL_t)TPMI_RH_CLEAR_Unmarshal,
11  #define TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL  (TPMI_RH_CLEAR_H_UNMARSHAL + 1)
12              (UNMARSHAL_t)TPMI_RH_HIERARCHY_AUTH_Unmarshal,
13  #define TPMI_RH_LOCKOUT_H_UNMARSHAL         (TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL + 1)
14              (UNMARSHAL_t)TPMI_RH_LOCKOUT_Unmarshal,
15  #define TPMI_RH_NV_AUTH_H_UNMARSHAL         (TPMI_RH_LOCKOUT_H_UNMARSHAL + 1)
16              (UNMARSHAL_t)TPMI_RH_NV_AUTH_Unmarshal,
17  #define TPMI_RH_NV_INDEX_H_UNMARSHAL        (TPMI_RH_NV_AUTH_H_UNMARSHAL + 1)
18              (UNMARSHAL_t)TPMI_RH_NV_INDEX_Unmarshal,
19  #define TPMI_RH_PLATFORM_H_UNMARSHAL        (TPMI_RH_NV_INDEX_H_UNMARSHAL + 1)
20              (UNMARSHAL_t)TPMI_RH_PLATFORM_Unmarshal,
21  #define TPMI_RH_PROVISION_H_UNMARSHAL       (TPMI_RH_PLATFORM_H_UNMARSHAL + 1)
22              (UNMARSHAL_t)TPMI_RH_PROVISION_Unmarshal,
23  #define TPMI_SH_HMAC_H_UNMARSHAL            (TPMI_RH_PROVISION_H_UNMARSHAL + 1)
24              (UNMARSHAL_t)TPMI_SH_HMAC_Unmarshal,
25  #define TPMI_SH_POLICY_H_UNMARSHAL          (TPMI_SH_HMAC_H_UNMARSHAL + 1)
26              (UNMARSHAL_t)TPMI_SH_POLICY_Unmarshal,
27  // HANDLE_FIRST_FLAG_TYPE is the first handle that needs a flag when called.
28  #define HANDLE_FIRST_FLAG_TYPE              (TPMI_SH_POLICY_H_UNMARSHAL + 1)
29  #define TPMI_DH_ENTITY_H_UNMARSHAL          (TPMI_SH_POLICY_H_UNMARSHAL + 1)
30              (UNMARSHAL_t)TPMI_DH_ENTITY_Unmarshal,
31  #define TPMI_DH_OBJECT_H_UNMARSHAL          (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
32              (UNMARSHAL_t)TPMI_DH_OBJECT_Unmarshal,
33  #define TPMI_DH_PARENT_H_UNMARSHAL          (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
34              (UNMARSHAL_t)TPMI_DH_PARENT_Unmarshal,
35  #define TPMI_DH_PCR_H_UNMARSHAL             (TPMI_DH_PARENT_H_UNMARSHAL + 1)
36              (UNMARSHAL_t)TPMI_DH_PCR_Unmarshal,
37  #define TPMI_RH_ENDORSEMENT_H_UNMARSHAL     (TPMI_DH_PCR_H_UNMARSHAL + 1)
38              (UNMARSHAL_t)TPMI_RH_ENDORSEMENT_Unmarshal,
39  #define TPMI_RH_HIERARCHY_H_UNMARSHAL       (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
40              (UNMARSHAL_t)TPMI_RH_HIERARCHY_Unmarshal,
41  // PARAMETER_FIRST_TYPE marks the end of the handle list.
42  #define PARAMETER_FIRST_TYPE                (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
43  #define TPM2B_DATA_P_UNMARSHAL              (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
44              (UNMARSHAL_t)TPM2B_DATA_Unmarshal,
45  #define TPM2B_DIGEST_P_UNMARSHAL            (TPM2B_DATA_P_UNMARSHAL + 1)
46              (UNMARSHAL_t)TPM2B_DIGEST_Unmarshal,
47  #define TPM2B_ECC_PARAMETER_P_UNMARSHAL     (TPM2B_DIGEST_P_UNMARSHAL + 1)
48              (UNMARSHAL_t)TPM2B_ECC_PARAMETER_Unmarshal,
49  #define TPM2B_ECC_POINT_P_UNMARSHAL         (TPM2B_ECC_PARAMETER_P_UNMARSHAL + 1)
50              (UNMARSHAL_t)TPM2B_ECC_POINT_Unmarshal,
51  #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 TPM2B_TIMEOUT_P_UNMARSHAL            (TPM2B_TEMPLATE_P_UNMARSHAL + 1)
80              (UNMARSHAL_t)TPM2B_TIMEOUT_Unmarshal,
81  #define TPMI_DH_CONTEXT_P_UNMARSHAL          (TPM2B_TIMEOUT_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_AT_P_UNMARSHAL                   (TPMT_TK_VERIFIED_P_UNMARSHAL + 1)
112             (UNMARSHAL_t)TPM_AT_Unmarshal,
113 #define TPM_CAP_P_UNMARSHAL                  (TPM_AT_P_UNMARSHAL + 1)
114             (UNMARSHAL_t)TPM_CAP_Unmarshal,
115 #define TPM_CLOCK_ADJUST_P_UNMARSHAL         (TPM_CAP_P_UNMARSHAL + 1)
116             (UNMARSHAL_t)TPM_CLOCK_ADJUST_Unmarshal,
117 #define TPM_EO_P_UNMARSHAL                   (TPM_CLOCK_ADJUST_P_UNMARSHAL + 1)
```

```
118                 (UNMARSHAL_t)TPM_EO_Unmarshal,
119     #define TPM_SE_P_UNMARSHAL                  (TPM_EO_P_UNMARSHAL + 1)
120                 (UNMARSHAL_t)TPM_SE_Unmarshal,
121     #define TPM_SU_P_UNMARSHAL                  (TPM_SE_P_UNMARSHAL + 1)
122                 (UNMARSHAL_t)TPM_SU_Unmarshal,
123     #define UINT16_P_UNMARSHAL                  (TPM_SU_P_UNMARSHAL + 1)
124                 (UNMARSHAL_t)UINT16_Unmarshal,
125     #define UINT32_P_UNMARSHAL                  (UINT16_P_UNMARSHAL + 1)
126                 (UNMARSHAL_t)UINT32_Unmarshal,
127     #define UINT64_P_UNMARSHAL                  (UINT32_P_UNMARSHAL + 1)
128                 (UNMARSHAL_t)UINT64_Unmarshal,
129     #define UINT8_P_UNMARSHAL                   (UINT64_P_UNMARSHAL + 1)
130                 (UNMARSHAL_t)UINT8_Unmarshal,
131     // PARAMETER_FIRST_FLAG_TYPE is the first parameter to need a flag.
132     #define PARAMETER_FIRST_FLAG_TYPE           (UINT8_P_UNMARSHAL + 1)
133     #define TPM2B_PUBLIC_P_UNMARSHAL            (UINT8_P_UNMARSHAL + 1)
134                 (UNMARSHAL_t)TPM2B_PUBLIC_Unmarshal,
135     #define TPMI_ALG_CIPHER_MODE_P_UNMARSHAL    (TPM2B_PUBLIC_P_UNMARSHAL + 1)
136                 (UNMARSHAL_t)TPMI_ALG_CIPHER_MODE_Unmarshal,
137     #define TPMI_ALG_HASH_P_UNMARSHAL           (TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + 1)
138                 (UNMARSHAL_t)TPMI_ALG_HASH_Unmarshal,
139     #define TPMI_ALG_MAC_SCHEME_P_UNMARSHAL     (TPMI_ALG_HASH_P_UNMARSHAL + 1)
140                 (UNMARSHAL_t)TPMI_ALG_MAC_SCHEME_Unmarshal,
141     #define TPMI_DH_PCR_P_UNMARSHAL             (TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + 1)
142                 (UNMARSHAL_t)TPMI_DH_PCR_Unmarshal,
143     #define TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL   (TPMI_DH_PCR_P_UNMARSHAL + 1)
144                 (UNMARSHAL_t)TPMI_ECC_KEY_EXCHANGE_Unmarshal,
145     #define TPMI_RH_ENABLES_P_UNMARSHAL         (TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL + 1)
146                 (UNMARSHAL_t)TPMI_RH_ENABLES_Unmarshal,
147     #define TPMI_RH_HIERARCHY_P_UNMARSHAL       (TPMI_RH_ENABLES_P_UNMARSHAL + 1)
148                 (UNMARSHAL_t)TPMI_RH_HIERARCHY_Unmarshal,
149     #define TPMT_RSA_DECRYPT_P_UNMARSHAL        (TPMI_RH_HIERARCHY_P_UNMARSHAL + 1)
150                 (UNMARSHAL_t)TPMT_RSA_DECRYPT_Unmarshal,
151     #define TPMT_SIGNATURE_P_UNMARSHAL          (TPMT_RSA_DECRYPT_P_UNMARSHAL + 1)
152                 (UNMARSHAL_t)TPMT_SIGNATURE_Unmarshal,
153     #define TPMT_SIG_SCHEME_P_UNMARSHAL         (TPMT_SIGNATURE_P_UNMARSHAL + 1)
154                 (UNMARSHAL_t)TPMT_SIG_SCHEME_Unmarshal,
155     #define TPMT_SYM_DEF_P_UNMARSHAL            (TPMT_SIG_SCHEME_P_UNMARSHAL + 1)
156                 (UNMARSHAL_t)TPMT_SYM_DEF_Unmarshal,
157     #define TPMT_SYM_DEF_OBJECT_P_UNMARSHAL     (TPMT_SYM_DEF_P_UNMARSHAL + 1)
158                 (UNMARSHAL_t)TPMT_SYM_DEF_OBJECT_Unmarshal
159     // PAREMETER_LAST_TYPE is the end of the command parameter list.
160     #define PARAMETER_LAST_TYPE                 (TPMT_SYM_DEF_OBJECT_P_UNMARSHAL)
161     };
```

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

```
162     const MARSHAL_t MarshalArray[] = {
163     #define UINT32_H_MARSHAL                    0
164                 (MARSHAL_t)UINT32_Marshal,
165     // RESPONSE_PARAMETER_FIRST_TYPE marks the end of the response handles.
166     #define RESPONSE_PARAMETER_FIRST_TYPE       (UINT32_H_MARSHAL + 1)
167     #define TPM2B_ATTEST_P_MARSHAL              (UINT32_H_MARSHAL + 1)
168                 (MARSHAL_t)TPM2B_ATTEST_Marshal,
169     #define TPM2B_CREATION_DATA_P_MARSHAL       (TPM2B_ATTEST_P_MARSHAL + 1)
170                 (MARSHAL_t)TPM2B_CREATION_DATA_Marshal,
171     #define TPM2B_DATA_P_MARSHAL                (TPM2B_CREATION_DATA_P_MARSHAL + 1)
172                 (MARSHAL_t)TPM2B_DATA_Marshal,
173     #define TPM2B_DIGEST_P_MARSHAL              (TPM2B_DATA_P_MARSHAL + 1)
174                 (MARSHAL_t)TPM2B_DIGEST_Marshal,
175     #define TPM2B_ECC_POINT_P_MARSHAL           (TPM2B_DIGEST_P_MARSHAL + 1)
176                 (MARSHAL_t)TPM2B_ECC_POINT_Marshal,
177     #define TPM2B_ENCRYPTED_SECRET_P_MARSHAL    (TPM2B_ECC_POINT_P_MARSHAL + 1)
178                 (MARSHAL_t)TPM2B_ENCRYPTED_SECRET_Marshal,
```



```
179  #define TPM2B_ID_OBJECT_P_MARSHAL            (TPM2B_ENCRYPTED_SECRET_P_MARSHAL + 1)
180              (MARSHAL_t)TPM2B_ID_OBJECT_Marshal,
181  #define TPM2B_IV_P_MARSHAL                   (TPM2B_ID_OBJECT_P_MARSHAL + 1)
182              (MARSHAL_t)TPM2B_IV_Marshal,
183  #define TPM2B_MAX_BUFFER_P_MARSHAL           (TPM2B_IV_P_MARSHAL + 1)
184              (MARSHAL_t)TPM2B_MAX_BUFFER_Marshal,
185  #define TPM2B_MAX_NV_BUFFER_P_MARSHAL        (TPM2B_MAX_BUFFER_P_MARSHAL + 1)
186              (MARSHAL_t)TPM2B_MAX_NV_BUFFER_Marshal,
187  #define TPM2B_NAME_P_MARSHAL                 (TPM2B_MAX_NV_BUFFER_P_MARSHAL + 1)
188              (MARSHAL_t)TPM2B_NAME_Marshal,
189  #define TPM2B_NV_PUBLIC_P_MARSHAL            (TPM2B_NAME_P_MARSHAL + 1)
190              (MARSHAL_t)TPM2B_NV_PUBLIC_Marshal,
191  #define TPM2B_PRIVATE_P_MARSHAL              (TPM2B_NV_PUBLIC_P_MARSHAL + 1)
192              (MARSHAL_t)TPM2B_PRIVATE_Marshal,
193  #define TPM2B_PUBLIC_P_MARSHAL               (TPM2B_PRIVATE_P_MARSHAL + 1)
194              (MARSHAL_t)TPM2B_PUBLIC_Marshal,
195  #define TPM2B_PUBLIC_KEY_RSA_P_MARSHAL       (TPM2B_PUBLIC_P_MARSHAL + 1)
196              (MARSHAL_t)TPM2B_PUBLIC_KEY_RSA_Marshal,
197  #define TPM2B_SENSITIVE_DATA_P_MARSHAL       (TPM2B_PUBLIC_KEY_RSA_P_MARSHAL + 1)
198              (MARSHAL_t)TPM2B_SENSITIVE_DATA_Marshal,
199  #define TPM2B_TIMEOUT_P_MARSHAL              (TPM2B_SENSITIVE_DATA_P_MARSHAL + 1)
200              (MARSHAL_t)TPM2B_TIMEOUT_Marshal,
201  #define UINT8_P_MARSHAL                      (TPM2B_TIMEOUT_P_MARSHAL + 1)
202              (MARSHAL_t)UINT8_Marshal,
203  #define TPML_AC_CAPABILITIES_P_MARSHAL       (UINT8_P_MARSHAL + 1)
204              (MARSHAL_t)TPML_AC_CAPABILITIES_Marshal,
205  #define TPML_ALG_P_MARSHAL                   (TPML_AC_CAPABILITIES_P_MARSHAL + 1)
206              (MARSHAL_t)TPML_ALG_Marshal,
207  #define TPML_DIGEST_P_MARSHAL                (TPML_ALG_P_MARSHAL + 1)
208              (MARSHAL_t)TPML_DIGEST_Marshal,
209  #define TPML_DIGEST_VALUES_P_MARSHAL         (TPML_DIGEST_P_MARSHAL + 1)
210              (MARSHAL_t)TPML_DIGEST_VALUES_Marshal,
211  #define TPML_PCR_SELECTION_P_MARSHAL         (TPML_DIGEST_VALUES_P_MARSHAL + 1)
212              (MARSHAL_t)TPML_PCR_SELECTION_Marshal,
213  #define TPMS_AC_OUTPUT_P_MARSHAL             (TPML_PCR_SELECTION_P_MARSHAL + 1)
214              (MARSHAL_t)TPMS_AC_OUTPUT_Marshal,
215  #define TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL (TPMS_AC_OUTPUT_P_MARSHAL + 1)
216              (MARSHAL_t)TPMS_ALGORITHM_DETAIL_ECC_Marshal,
217  #define TPMS_CAPABILITY_DATA_P_MARSHAL       \
218              (TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL + 1)
219              (MARSHAL_t)TPMS_CAPABILITY_DATA_Marshal,
220  #define TPMS_CONTEXT_P_MARSHAL               (TPMS_CAPABILITY_DATA_P_MARSHAL + 1)
221              (MARSHAL_t)TPMS_CONTEXT_Marshal,
222  #define TPMS_TIME_INFO_P_MARSHAL             (TPMS_CONTEXT_P_MARSHAL + 1)
223              (MARSHAL_t)TPMS_TIME_INFO_Marshal,
224  #define TPMT_HA_P_MARSHAL                    (TPMS_TIME_INFO_P_MARSHAL + 1)
225              (MARSHAL_t)TPMT_HA_Marshal,
226  #define TPMT_SIGNATURE_P_MARSHAL             (TPMT_HA_P_MARSHAL + 1)
227              (MARSHAL_t)TPMT_SIGNATURE_Marshal,
228  #define TPMT_TK_AUTH_P_MARSHAL               (TPMT_SIGNATURE_P_MARSHAL + 1)
229              (MARSHAL_t)TPMT_TK_AUTH_Marshal,
230  #define TPMT_TK_CREATION_P_MARSHAL           (TPMT_TK_AUTH_P_MARSHAL + 1)
231              (MARSHAL_t)TPMT_TK_CREATION_Marshal,
232  #define TPMT_TK_HASHCHECK_P_MARSHAL          (TPMT_TK_CREATION_P_MARSHAL + 1)
233              (MARSHAL_t)TPMT_TK_HASHCHECK_Marshal,
234  #define TPMT_TK_VERIFIED_P_MARSHAL           (TPMT_TK_HASHCHECK_P_MARSHAL + 1)
235              (MARSHAL_t)TPMT_TK_VERIFIED_Marshal,
236  #define UINT32_P_MARSHAL                     (TPMT_TK_VERIFIED_P_MARSHAL + 1)
237              (MARSHAL_t)UINT32_Marshal,
238  #define UINT16_P_MARSHAL                     (UINT32_P_MARSHAL + 1)
239              (MARSHAL_t)UINT16_Marshal
240  // RESPONSE_PARAMETER_LAST_TYPE is the end of the response parameter list.
241  #define RESPONSE_PARAMETER_LAST_TYPE         (UINT16_P_MARSHAL)
242  };
```

This list of aliases allows the types in the _COMMAND_DESCRIPTOR_T to match the types in the command/response templates of part 3.

```
243  #define INT32_P_UNMARSHAL                    UINT32_P_UNMARSHAL
244  #define TPM2B_AUTH_P_UNMARSHAL               TPM2B_DIGEST_P_UNMARSHAL
245  #define TPM2B_NONCE_P_UNMARSHAL              TPM2B_DIGEST_P_UNMARSHAL
246  #define TPM2B_OPERAND_P_UNMARSHAL            TPM2B_DIGEST_P_UNMARSHAL
247  #define TPMA_LOCALITY_P_UNMARSHAL            UINT8_P_UNMARSHAL
248  #define TPM_CC_P_UNMARSHAL                   UINT32_P_UNMARSHAL
249  #define TPMI_DH_CONTEXT_H_MARSHAL            UINT32_H_MARSHAL
250  #define TPMI_DH_OBJECT_H_MARSHAL             UINT32_H_MARSHAL
251  #define TPMI_SH_AUTH_SESSION_H_MARSHAL       UINT32_H_MARSHAL
252  #define TPM_HANDLE_H_MARSHAL                 UINT32_H_MARSHAL
253  #define TPM2B_NONCE_P_MARSHAL                TPM2B_DIGEST_P_MARSHAL
254  #define TPMI_YES_NO_P_MARSHAL                UINT8_P_MARSHAL
255  #define TPM_RC_P_MARSHAL                     UINT32_P_MARSHAL
256  #if CC_Startup
257  #include "Startup_fp.h"
258  typedef TPM_RC  (Startup_Entry)(
259      Startup_In                  *in
260  );
261  typedef const struct {
262      Startup_Entry           *entry;
263      UINT16                  inSize;
264      UINT16                  outSize;
265      UINT16                  offsetOfTypes;
266      BYTE                    types[3];
267  } Startup_COMMAND_DESCRIPTOR_t;
268  Startup_COMMAND_DESCRIPTOR_t _StartupData = {
269      /* entry         */     &TPM2_Startup,
270      /* inSize        */     (UINT16)(sizeof(Startup_In)),
271      /* outSize       */     0,
272      /* offsetOfTypes */     offsetof(Startup_COMMAND_DESCRIPTOR_t, types),
273      /* offsets       */     // No parameter offsets;
274      /* types         */     {TPM_SU_P_UNMARSHAL,
275                               END_OF_LIST,
276                               END_OF_LIST}
277  };
278  #define _StartupDataAddress (&_StartupData)
279  #else
280  #define _StartupDataAddress 0
281  #endif // CC_Startup
282  #if CC_Shutdown
283  #include "Shutdown_fp.h"
284  typedef TPM_RC  (Shutdown_Entry)(
285      Shutdown_In                 *in
286  );
287  typedef const struct {
288      Shutdown_Entry          *entry;
289      UINT16                  inSize;
290      UINT16                  outSize;
291      UINT16                  offsetOfTypes;
292      BYTE                    types[3];
293  } Shutdown_COMMAND_DESCRIPTOR_t;
294  Shutdown_COMMAND_DESCRIPTOR_t _ShutdownData = {
295      /* entry         */     &TPM2_Shutdown,
296      /* inSize        */     (UINT16)(sizeof(Shutdown_In)),
297      /* outSize       */     0,
298      /* offsetOfTypes */     offsetof(Shutdown_COMMAND_DESCRIPTOR_t, types),
299      /* offsets       */     // No parameter offsets;
300      /* types         */     {TPM_SU_P_UNMARSHAL,
301                               END_OF_LIST,
302                               END_OF_LIST}
303  };
304  #define _ShutdownDataAddress (&_ShutdownData)
```



```
305   #else
306   #define _ShutdownDataAddress 0
307   #endif // CC_Shutdown
308   #if CC_SelfTest
309   #include "SelfTest_fp.h"
310   typedef TPM_RC  (SelfTest_Entry)(
311       SelfTest_In                 *in
312   );
313   typedef const struct {
314       SelfTest_Entry          *entry;
315       UINT16                  inSize;
316       UINT16                  outSize;
317       UINT16                  offsetOfTypes;
318       BYTE                    types[3];
319   } SelfTest_COMMAND_DESCRIPTOR_t;
320   SelfTest_COMMAND_DESCRIPTOR_t _SelfTestData = {
321       /* entry         */     &TPM2_SelfTest,
322       /* inSize        */     (UINT16)(sizeof(SelfTest_In)),
323       /* outSize       */     0,
324       /* offsetOfTypes */     offsetof(SelfTest_COMMAND_DESCRIPTOR_t, types),
325       /* offsets       */     // No parameter offsets;
326       /* types         */     {TPMI_YES_NO_P_UNMARSHAL,
327                                END_OF_LIST,
328                                END_OF_LIST}
329   };
330   #define _SelfTestDataAddress (&_SelfTestData)
331   #else
332   #define _SelfTestDataAddress 0
333   #endif // CC_SelfTest
334   #if CC_IncrementalSelfTest
335   #include "IncrementalSelfTest_fp.h"
336   typedef TPM_RC  (IncrementalSelfTest_Entry)(
337       IncrementalSelfTest_In          *in,
338       IncrementalSelfTest_Out         *out
339   );
340   typedef const struct {
341       IncrementalSelfTest_Entry   *entry;
342       UINT16                      inSize;
343       UINT16                      outSize;
344       UINT16                      offsetOfTypes;
345       BYTE                        types[4];
346   } IncrementalSelfTest_COMMAND_DESCRIPTOR_t;
347   IncrementalSelfTest_COMMAND_DESCRIPTOR_t _IncrementalSelfTestData = {
348       /* entry         */         &TPM2_IncrementalSelfTest,
349       /* inSize        */         (UINT16)(sizeof(IncrementalSelfTest_In)),
350       /* outSize       */         (UINT16)(sizeof(IncrementalSelfTest_Out)),
351       /* offsetOfTypes */         offsetof(IncrementalSelfTest_COMMAND_DESCRIPTOR_t,
      types),
352       /* offsets       */         // No parameter offsets;
353       /* types         */         {TPML_ALG_P_UNMARSHAL,
354                                    END_OF_LIST,
355                                    TPML_ALG_P_MARSHAL,
356                                    END_OF_LIST}
357   };
358   #define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
359   #else
360   #define _IncrementalSelfTestDataAddress 0
361   #endif // CC_IncrementalSelfTest
362   #if CC_GetTestResult
363   #include "GetTestResult_fp.h"
364   typedef TPM_RC  (GetTestResult_Entry)(
365       GetTestResult_Out           *out
366   );
367   typedef const struct {
368       GetTestResult_Entry     *entry;
369       UINT16                  inSize;
```

```
370         UINT16                  outSize;
371         UINT16                  offsetOfTypes;
372         UINT16                  paramOffsets[1];
373         BYTE                    types[4];
374     } GetTestResult_COMMAND_DESCRIPTOR_t;
375     GetTestResult_COMMAND_DESCRIPTOR_t _GetTestResultData = {
376         /* entry         */     &TPM2_GetTestResult,
377         /* inSize        */     0,
378         /* outSize       */     (UINT16)(sizeof(GetTestResult_Out)),
379         /* offsetOfTypes */     offsetof(GetTestResult_COMMAND_DESCRIPTOR_t, types),
380         /* offsets       */     {(UINT16)(offsetof(GetTestResult_Out, testResult))},
381         /* types         */     {END_OF_LIST,
382                                   TPM2B_MAX_BUFFER_P_MARSHAL,
383                                   TPM_RC_P_MARSHAL,
384                                   END_OF_LIST}
385     };
386     #define _GetTestResultDataAddress (&_GetTestResultData)
387     #else
388     #define _GetTestResultDataAddress 0
389     #endif // CC_GetTestResult
390     #if CC_StartAuthSession
391     #include "StartAuthSession_fp.h"
392     typedef TPM_RC  (StartAuthSession_Entry)(
393         StartAuthSession_In         *in,
394         StartAuthSession_Out        *out
395     );
396     typedef const struct {
397         StartAuthSession_Entry  *entry;
398         UINT16                  inSize;
399         UINT16                  outSize;
400         UINT16                  offsetOfTypes;
401         UINT16                  paramOffsets[7];
402         BYTE                    types[11];
403     } StartAuthSession_COMMAND_DESCRIPTOR_t;
404     StartAuthSession_COMMAND_DESCRIPTOR_t _StartAuthSessionData = {
405         /* entry         */     &TPM2_StartAuthSession,
406         /* inSize        */     (UINT16)(sizeof(StartAuthSession_In)),
407         /* outSize       */     (UINT16)(sizeof(StartAuthSession_Out)),
408         /* offsetOfTypes */     offsetof(StartAuthSession_COMMAND_DESCRIPTOR_t, types),
409         /* offsets       */     {(UINT16)(offsetof(StartAuthSession_In, bind)),
410                                   (UINT16)(offsetof(StartAuthSession_In, nonceCaller)),
411                                   (UINT16)(offsetof(StartAuthSession_In, encryptedSalt)),
412                                   (UINT16)(offsetof(StartAuthSession_In, sessionType)),
413                                   (UINT16)(offsetof(StartAuthSession_In, symmetric)),
414                                   (UINT16)(offsetof(StartAuthSession_In, authHash)),
415                                   (UINT16)(offsetof(StartAuthSession_Out, nonceTPM))},
416         /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
417                                   TPMI_DH_ENTITY_H_UNMARSHAL + ADD_FLAG,
418                                   TPM2B_NONCE_P_UNMARSHAL,
419                                   TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
420                                   TPM_SE_P_UNMARSHAL,
421                                   TPMT_SYM_DEF_P_UNMARSHAL + ADD_FLAG,
422                                   TPMI_ALG_HASH_P_UNMARSHAL,
423                                   END_OF_LIST,
424                                   TPMI_SH_AUTH_SESSION_H_MARSHAL,
425                                   TPM2B_NONCE_P_MARSHAL,
426                                   END_OF_LIST}
427     };
428     #define _StartAuthSessionDataAddress (&_StartAuthSessionData)
429     #else
430     #define _StartAuthSessionDataAddress 0
431     #endif // CC_StartAuthSession
432     #if CC_PolicyRestart
433     #include "PolicyRestart_fp.h"
434     typedef TPM_RC  (PolicyRestart_Entry)(
435         PolicyRestart_In            *in
```



```
436   );
437   typedef const struct {
438       PolicyRestart_Entry     *entry;
439       UINT16                  inSize;
440       UINT16                  outSize;
441       UINT16                  offsetOfTypes;
442       BYTE                    types[3];
443   } PolicyRestart_COMMAND_DESCRIPTOR_t;
444   PolicyRestart_COMMAND_DESCRIPTOR_t _PolicyRestartData = {
445       /* entry         */     &TPM2_PolicyRestart,
446       /* inSize        */     (UINT16)(sizeof(PolicyRestart_In)),
447       /* outSize       */     0,
448       /* offsetOfTypes */     offsetof(PolicyRestart_COMMAND_DESCRIPTOR_t, types),
449       /* offsets       */     // No parameter offsets;
450       /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
451                                END_OF_LIST,
452                                END_OF_LIST}
453   };
454   #define _PolicyRestartDataAddress (&_PolicyRestartData)
455   #else
456   #define _PolicyRestartDataAddress 0
457   #endif // CC_PolicyRestart
458   #if CC_Create
459   #include "Create_fp.h"
460   typedef TPM_RC  (Create_Entry)(
461       Create_In                   *in,
462       Create_Out                  *out
463   );
464   typedef const struct {
465       Create_Entry            *entry;
466       UINT16                  inSize;
467       UINT16                  outSize;
468       UINT16                  offsetOfTypes;
469       UINT16                  paramOffsets[8];
470       BYTE                    types[12];
471   } Create_COMMAND_DESCRIPTOR_t;
472   Create_COMMAND_DESCRIPTOR_t _CreateData = {
473       /* entry         */     &TPM2_Create,
474       /* inSize        */     (UINT16)(sizeof(Create_In)),
475       /* outSize       */     (UINT16)(sizeof(Create_Out)),
476       /* offsetOfTypes */     offsetof(Create_COMMAND_DESCRIPTOR_t, types),
477       /* offsets       */     {(UINT16)(offsetof(Create_In, inSensitive)),
478                                (UINT16)(offsetof(Create_In, inPublic)),
479                                (UINT16)(offsetof(Create_In, outsideInfo)),
480                                (UINT16)(offsetof(Create_In, creationPCR)),
481                                (UINT16)(offsetof(Create_Out, outPublic)),
482                                (UINT16)(offsetof(Create_Out, creationData)),
483                                (UINT16)(offsetof(Create_Out, creationHash)),
484                                (UINT16)(offsetof(Create_Out, creationTicket))},
485       /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
486                                TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
487                                TPM2B_PUBLIC_P_UNMARSHAL,
488                                TPM2B_DATA_P_UNMARSHAL,
489                                TPML_PCR_SELECTION_P_UNMARSHAL,
490                                END_OF_LIST,
491                                TPM2B_PRIVATE_P_MARSHAL,
492                                TPM2B_PUBLIC_P_MARSHAL,
493                                TPM2B_CREATION_DATA_P_MARSHAL,
494                                TPM2B_DIGEST_P_MARSHAL,
495                                TPMT_TK_CREATION_P_MARSHAL,
496                                END_OF_LIST}
497   };
498   #define _CreateDataAddress (&_CreateData)
499   #else
500   #define _CreateDataAddress 0
501   #endif // CC_Create
```

```
502  #if CC_Load
503  #include "Load_fp.h"
504  typedef TPM_RC  (Load_Entry)(
505      Load_In                     *in,
506      Load_Out                    *out
507  );
508  typedef const struct {
509      Load_Entry              *entry;
510      UINT16                  inSize;
511      UINT16                  outSize;
512      UINT16                  offsetOfTypes;
513      UINT16                  paramOffsets[3];
514      BYTE                    types[7];
515  } Load_COMMAND_DESCRIPTOR_t;
516  Load_COMMAND_DESCRIPTOR_t _LoadData = {
517      /* entry         */     &TPM2_Load,
518      /* inSize        */     (UINT16)(sizeof(Load_In)),
519      /* outSize       */     (UINT16)(sizeof(Load_Out)),
520      /* offsetOfTypes */     offsetof(Load_COMMAND_DESCRIPTOR_t, types),
521      /* offsets       */     {(UINT16)(offsetof(Load_In, inPrivate)),
522                               (UINT16)(offsetof(Load_In, inPublic)),
523                               (UINT16)(offsetof(Load_Out, name))},
524      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
525                               TPM2B_PRIVATE_P_UNMARSHAL,
526                               TPM2B_PUBLIC_P_UNMARSHAL,
527                               END_OF_LIST,
528                               TPM_HANDLE_H_MARSHAL,
529                               TPM2B_NAME_P_MARSHAL,
530                               END_OF_LIST}
531  };
532  #define _LoadDataAddress (&_LoadData)
533  #else
534  #define _LoadDataAddress 0
535  #endif // CC_Load
536  #if CC_LoadExternal
537  #include "LoadExternal_fp.h"
538  typedef TPM_RC  (LoadExternal_Entry)(
539      LoadExternal_In             *in,
540      LoadExternal_Out            *out
541  );
542  typedef const struct {
543      LoadExternal_Entry      *entry;
544      UINT16                  inSize;
545      UINT16                  outSize;
546      UINT16                  offsetOfTypes;
547      UINT16                  paramOffsets[3];
548      BYTE                    types[7];
549  } LoadExternal_COMMAND_DESCRIPTOR_t;
550  LoadExternal_COMMAND_DESCRIPTOR_t _LoadExternalData = {
551      /* entry         */     &TPM2_LoadExternal,
552      /* inSize        */     (UINT16)(sizeof(LoadExternal_In)),
553      /* outSize       */     (UINT16)(sizeof(LoadExternal_Out)),
554      /* offsetOfTypes */     offsetof(LoadExternal_COMMAND_DESCRIPTOR_t, types),
555      /* offsets       */     {(UINT16)(offsetof(LoadExternal_In, inPublic)),
556                               (UINT16)(offsetof(LoadExternal_In, hierarchy)),
557                               (UINT16)(offsetof(LoadExternal_Out, name))},
558      /* types         */     {TPM2B_SENSITIVE_P_UNMARSHAL,
559                               TPM2B_PUBLIC_P_UNMARSHAL + ADD_FLAG,
560                               TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
561                               END_OF_LIST,
562                               TPM_HANDLE_H_MARSHAL,
563                               TPM2B_NAME_P_MARSHAL,
564                               END_OF_LIST}
565  };
566  #define _LoadExternalDataAddress (&_LoadExternalData)
567  #else
```



```
568  #define _LoadExternalDataAddress 0
569  #endif // CC_LoadExternal
570  #if CC_ReadPublic
571  #include "ReadPublic_fp.h"
572  typedef TPM_RC  (ReadPublic_Entry)(
573      ReadPublic_In               *in,
574      ReadPublic_Out              *out
575  );
576  typedef const struct {
577      ReadPublic_Entry        *entry;
578      UINT16                  inSize;
579      UINT16                  outSize;
580      UINT16                  offsetOfTypes;
581      UINT16                  paramOffsets[2];
582      BYTE                    types[6];
583  } ReadPublic_COMMAND_DESCRIPTOR_t;
584  ReadPublic_COMMAND_DESCRIPTOR_t _ReadPublicData = {
585      /* entry         */     &TPM2_ReadPublic,
586      /* inSize        */     (UINT16)(sizeof(ReadPublic_In)),
587      /* outSize       */     (UINT16)(sizeof(ReadPublic_Out)),
588      /* offsetOfTypes */     offsetof(ReadPublic_COMMAND_DESCRIPTOR_t, types),
589      /* offsets       */     {(UINT16)(offsetof(ReadPublic_Out, name)),
590                              (UINT16)(offsetof(ReadPublic_Out, qualifiedName))},
591      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
592                              END_OF_LIST,
593                              TPM2B_PUBLIC_P_MARSHAL,
594                              TPM2B_NAME_P_MARSHAL,
595                              TPM2B_NAME_P_MARSHAL,
596                              END_OF_LIST}
597  };
598  #define _ReadPublicDataAddress (&_ReadPublicData)
599  #else
600  #define _ReadPublicDataAddress 0
601  #endif // CC_ReadPublic
602  #if CC_ActivateCredential
603  #include "ActivateCredential_fp.h"
604  typedef TPM_RC  (ActivateCredential_Entry)(
605      ActivateCredential_In           *in,
606      ActivateCredential_Out          *out
607  );
608  typedef const struct {
609      ActivateCredential_Entry    *entry;
610      UINT16                      inSize;
611      UINT16                      outSize;
612      UINT16                      offsetOfTypes;
613      UINT16                      paramOffsets[3];
614      BYTE                        types[7];
615  } ActivateCredential_COMMAND_DESCRIPTOR_t;
616  ActivateCredential_COMMAND_DESCRIPTOR_t _ActivateCredentialData = {
617      /* entry         */         &TPM2_ActivateCredential,
618      /* inSize        */         (UINT16)(sizeof(ActivateCredential_In)),
619      /* outSize       */         (UINT16)(sizeof(ActivateCredential_Out)),
620      /* offsetOfTypes */         offsetof(ActivateCredential_COMMAND_DESCRIPTOR_t,
     types),
621      /* offsets       */         {(UINT16)(offsetof(ActivateCredential_In, keyHandle)),
622                                  (UINT16)(offsetof(ActivateCredential_In,
     credentialBlob)),
623                                  (UINT16)(offsetof(ActivateCredential_In, secret))},
624      /* types         */         {TPMI_DH_OBJECT_H_UNMARSHAL,
625                                  TPMI_DH_OBJECT_H_UNMARSHAL,
626                                  TPM2B_ID_OBJECT_P_UNMARSHAL,
627                                  TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
628                                  END_OF_LIST,
629                                  TPM2B_DIGEST_P_MARSHAL,
630                                  END_OF_LIST}
631  };
```

```
632  #define _ActivateCredentialDataAddress (&_ActivateCredentialData)
633  #else
634  #define _ActivateCredentialDataAddress 0
635  #endif // CC_ActivateCredential
636  #if CC_MakeCredential
637  #include "MakeCredential_fp.h"
638  typedef TPM_RC  (MakeCredential_Entry)(
639      MakeCredential_In           *in,
640      MakeCredential_Out          *out
641  );
642  typedef const struct {
643      MakeCredential_Entry    *entry;
644      UINT16                  inSize;
645      UINT16                  outSize;
646      UINT16                  offsetOfTypes;
647      UINT16                  paramOffsets[3];
648      BYTE                    types[7];
649  } MakeCredential_COMMAND_DESCRIPTOR_t;
650  MakeCredential_COMMAND_DESCRIPTOR_t _MakeCredentialData = {
651      /* entry         */     &TPM2_MakeCredential,
652      /* inSize        */     (UINT16)(sizeof(MakeCredential_In)),
653      /* outSize       */     (UINT16)(sizeof(MakeCredential_Out)),
654      /* offsetOfTypes */     offsetof(MakeCredential_COMMAND_DESCRIPTOR_t, types),
655      /* offsets       */     {(UINT16)(offsetof(MakeCredential_In, credential)),
656                               (UINT16)(offsetof(MakeCredential_In, objectName)),
657                               (UINT16)(offsetof(MakeCredential_Out, secret))},
658      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
659                               TPM2B_DIGEST_P_UNMARSHAL,
660                               TPM2B_NAME_P_UNMARSHAL,
661                               END_OF_LIST,
662                               TPM2B_ID_OBJECT_P_MARSHAL,
663                               TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
664                               END_OF_LIST}
665  };
666  #define _MakeCredentialDataAddress (&_MakeCredentialData)
667  #else
668  #define _MakeCredentialDataAddress 0
669  #endif // CC_MakeCredential
670  #if CC_Unseal
671  #include "Unseal_fp.h"
672  typedef TPM_RC  (Unseal_Entry)(
673      Unseal_In                   *in,
674      Unseal_Out                  *out
675  );
676  typedef const struct {
677      Unseal_Entry            *entry;
678      UINT16                  inSize;
679      UINT16                  outSize;
680      UINT16                  offsetOfTypes;
681      BYTE                    types[4];
682  } Unseal_COMMAND_DESCRIPTOR_t;
683  Unseal_COMMAND_DESCRIPTOR_t _UnsealData = {
684      /* entry         */     &TPM2_Unseal,
685      /* inSize        */     (UINT16)(sizeof(Unseal_In)),
686      /* outSize       */     (UINT16)(sizeof(Unseal_Out)),
687      /* offsetOfTypes */     offsetof(Unseal_COMMAND_DESCRIPTOR_t, types),
688      /* offsets       */     // No parameter offsets;
689      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
690                               END_OF_LIST,
691                               TPM2B_SENSITIVE_DATA_P_MARSHAL,
692                               END_OF_LIST}
693  };
694  #define _UnsealDataAddress (&_UnsealData)
695  #else
696  #define _UnsealDataAddress 0
697  #endif // CC_Unseal
```



```
698  #if CC_ObjectChangeAuth
699  #include "ObjectChangeAuth_fp.h"
700  typedef TPM_RC  (ObjectChangeAuth_Entry)(
701      ObjectChangeAuth_In         *in,
702      ObjectChangeAuth_Out        *out
703  );
704  typedef const struct {
705      ObjectChangeAuth_Entry  *entry;
706      UINT16                  inSize;
707      UINT16                  outSize;
708      UINT16                  offsetOfTypes;
709      UINT16                  paramOffsets[2];
710      BYTE                    types[6];
711  } ObjectChangeAuth_COMMAND_DESCRIPTOR_t;
712  ObjectChangeAuth_COMMAND_DESCRIPTOR_t _ObjectChangeAuthData = {
713      /* entry         */     &TPM2_ObjectChangeAuth,
714      /* inSize        */     (UINT16)(sizeof(ObjectChangeAuth_In)),
715      /* outSize       */     (UINT16)(sizeof(ObjectChangeAuth_Out)),
716      /* offsetOfTypes */     offsetof(ObjectChangeAuth_COMMAND_DESCRIPTOR_t, types),
717      /* offsets       */     {(UINT16)(offsetof(ObjectChangeAuth_In, parentHandle)),
718                              (UINT16)(offsetof(ObjectChangeAuth_In, newAuth))},
719      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
720                              TPMI_DH_OBJECT_H_UNMARSHAL,
721                              TPM2B_AUTH_P_UNMARSHAL,
722                              END_OF_LIST,
723                              TPM2B_PRIVATE_P_MARSHAL,
724                              END_OF_LIST}
725  };
726  #define _ObjectChangeAuthDataAddress (&_ObjectChangeAuthData)
727  #else
728  #define _ObjectChangeAuthDataAddress 0
729  #endif // CC_ObjectChangeAuth
730  #if CC_CreateLoaded
731  #include "CreateLoaded_fp.h"
732  typedef TPM_RC  (CreateLoaded_Entry)(
733      CreateLoaded_In             *in,
734      CreateLoaded_Out            *out
735  );
736  typedef const struct {
737      CreateLoaded_Entry      *entry;
738      UINT16                  inSize;
739      UINT16                  outSize;
740      UINT16                  offsetOfTypes;
741      UINT16                  paramOffsets[5];
742      BYTE                    types[9];
743  } CreateLoaded_COMMAND_DESCRIPTOR_t;
744  CreateLoaded_COMMAND_DESCRIPTOR_t _CreateLoadedData = {
745      /* entry         */     &TPM2_CreateLoaded,
746      /* inSize        */     (UINT16)(sizeof(CreateLoaded_In)),
747      /* outSize       */     (UINT16)(sizeof(CreateLoaded_Out)),
748      /* offsetOfTypes */     offsetof(CreateLoaded_COMMAND_DESCRIPTOR_t, types),
749      /* offsets       */     {(UINT16)(offsetof(CreateLoaded_In, inSensitive)),
750                              (UINT16)(offsetof(CreateLoaded_In, inPublic)),
751                              (UINT16)(offsetof(CreateLoaded_Out, outPrivate)),
752                              (UINT16)(offsetof(CreateLoaded_Out, outPublic)),
753                              (UINT16)(offsetof(CreateLoaded_Out, name))},
754      /* types         */     {TPMI_DH_PARENT_H_UNMARSHAL + ADD_FLAG,
755                              TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
756                              TPM2B_TEMPLATE_P_UNMARSHAL,
757                              END_OF_LIST,
758                              TPM_HANDLE_H_MARSHAL,
759                              TPM2B_PRIVATE_P_MARSHAL,
760                              TPM2B_PUBLIC_P_MARSHAL,
761                              TPM2B_NAME_P_MARSHAL,
762                              END_OF_LIST}
763  };
```

```
764  #define _CreateLoadedDataAddress (&_CreateLoadedData)
765  #else
766  #define _CreateLoadedDataAddress 0
767  #endif // CC_CreateLoaded
768  #if CC_Duplicate
769  #include "Duplicate_fp.h"
770  typedef TPM_RC  (Duplicate_Entry)(
771      Duplicate_In                *in,
772      Duplicate_Out               *out
773  );
774  typedef const struct {
775      Duplicate_Entry         *entry;
776      UINT16                  inSize;
777      UINT16                  outSize;
778      UINT16                  offsetOfTypes;
779      UINT16                  paramOffsets[5];
780      BYTE                    types[9];
781  } Duplicate_COMMAND_DESCRIPTOR_t;
782  Duplicate_COMMAND_DESCRIPTOR_t _DuplicateData = {
783      /* entry         */     &TPM2_Duplicate,
784      /* inSize        */     (UINT16)(sizeof(Duplicate_In)),
785      /* outSize       */     (UINT16)(sizeof(Duplicate_Out)),
786      /* offsetOfTypes */     offsetof(Duplicate_COMMAND_DESCRIPTOR_t, types),
787      /* offsets       */     {(UINT16)(offsetof(Duplicate_In, newParentHandle)),
788                              (UINT16)(offsetof(Duplicate_In, encryptionKeyIn)),
789                              (UINT16)(offsetof(Duplicate_In, symmetricAlg)),
790                              (UINT16)(offsetof(Duplicate_Out, duplicate)),
791                              (UINT16)(offsetof(Duplicate_Out, outSymSeed))},
792      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
793                              TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
794                              TPM2B_DATA_P_UNMARSHAL,
795                              TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
796                              END_OF_LIST,
797                              TPM2B_DATA_P_MARSHAL,
798                              TPM2B_PRIVATE_P_MARSHAL,
799                              TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
800                              END_OF_LIST}
801  };
802  #define _DuplicateDataAddress (&_DuplicateData)
803  #else
804  #define _DuplicateDataAddress 0
805  #endif // CC_Duplicate
806  #if CC_Rewrap
807  #include "Rewrap_fp.h"
808  typedef TPM_RC  (Rewrap_Entry)(
809      Rewrap_In                   *in,
810      Rewrap_Out                  *out
811  );
812  typedef const struct {
813      Rewrap_Entry            *entry;
814      UINT16                  inSize;
815      UINT16                  outSize;
816      UINT16                  offsetOfTypes;
817      UINT16                  paramOffsets[5];
818      BYTE                    types[9];
819  } Rewrap_COMMAND_DESCRIPTOR_t;
820  Rewrap_COMMAND_DESCRIPTOR_t _RewrapData = {
821      /* entry         */     &TPM2_Rewrap,
822      /* inSize        */     (UINT16)(sizeof(Rewrap_In)),
823      /* outSize       */     (UINT16)(sizeof(Rewrap_Out)),
824      /* offsetOfTypes */     offsetof(Rewrap_COMMAND_DESCRIPTOR_t, types),
825      /* offsets       */     {(UINT16)(offsetof(Rewrap_In, newParent)),
826                              (UINT16)(offsetof(Rewrap_In, inDuplicate)),
827                              (UINT16)(offsetof(Rewrap_In, name)),
828                              (UINT16)(offsetof(Rewrap_In, inSymSeed)),
829                              (UINT16)(offsetof(Rewrap_Out, outSymSeed))},
```



```
830         /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
831                                     TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
832                                     TPM2B_PRIVATE_P_UNMARSHAL,
833                                     TPM2B_NAME_P_UNMARSHAL,
834                                     TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
835                                     END_OF_LIST,
836                                     TPM2B_PRIVATE_P_MARSHAL,
837                                     TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
838                                     END_OF_LIST}
839     };
840     #define _RewrapDataAddress (&_RewrapData)
841     #else
842     #define _RewrapDataAddress 0
843     #endif // CC_Rewrap
844     #if CC_Import
845     #include "Import_fp.h"
846     typedef TPM_RC  (Import_Entry)(
847         Import_In                   *in,
848         Import_Out                  *out
849     );
850     typedef const struct {
851         Import_Entry            *entry;
852         UINT16                  inSize;
853         UINT16                  outSize;
854         UINT16                  offsetOfTypes;
855         UINT16                  paramOffsets[5];
856         BYTE                    types[9];
857     } Import_COMMAND_DESCRIPTOR_t;
858     Import_COMMAND_DESCRIPTOR_t _ImportData = {
859         /* entry          */     &TPM2_Import,
860         /* inSize         */     (UINT16)(sizeof(Import_In)),
861         /* outSize        */     (UINT16)(sizeof(Import_Out)),
862         /* offsetOfTypes */      offsetof(Import_COMMAND_DESCRIPTOR_t, types),
863         /* offsets        */     {(UINT16)(offsetof(Import_In, encryptionKey)),
864                                     (UINT16)(offsetof(Import_In, objectPublic)),
865                                     (UINT16)(offsetof(Import_In, duplicate)),
866                                     (UINT16)(offsetof(Import_In, inSymSeed)),
867                                     (UINT16)(offsetof(Import_In, symmetricAlg))},
868         /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
869                                     TPM2B_DATA_P_UNMARSHAL,
870                                     TPM2B_PUBLIC_P_UNMARSHAL,
871                                     TPM2B_PRIVATE_P_UNMARSHAL,
872                                     TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
873                                     TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
874                                     END_OF_LIST,
875                                     TPM2B_PRIVATE_P_MARSHAL,
876                                     END_OF_LIST}
877     };
878     #define _ImportDataAddress (&_ImportData)
879     #else
880     #define _ImportDataAddress 0
881     #endif // CC_Import
882     #if CC_RSA_Encrypt
883     #include "RSA_Encrypt_fp.h"
884     typedef TPM_RC  (RSA_Encrypt_Entry)(
885         RSA_Encrypt_In              *in,
886         RSA_Encrypt_Out             *out
887     );
888     typedef const struct {
889         RSA_Encrypt_Entry       *entry;
890         UINT16                  inSize;
891         UINT16                  outSize;
892         UINT16                  offsetOfTypes;
893         UINT16                  paramOffsets[3];
894         BYTE                    types[7];
895     } RSA_Encrypt_COMMAND_DESCRIPTOR_t;
```

```
896  RSA_Encrypt_COMMAND_DESCRIPTOR_t _RSA_EncryptData = {
897      /* entry         */     &TPM2_RSA_Encrypt,
898      /* inSize        */     (UINT16)(sizeof(RSA_Encrypt_In)),
899      /* outSize       */     (UINT16)(sizeof(RSA_Encrypt_Out)),
900      /* offsetOfTypes */     offsetof(RSA_Encrypt_COMMAND_DESCRIPTOR_t, types),
901      /* offsets       */     {(UINT16)(offsetof(RSA_Encrypt_In, message)),
902                               (UINT16)(offsetof(RSA_Encrypt_In, inScheme)),
903                               (UINT16)(offsetof(RSA_Encrypt_In, label))},
904      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
905                               TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
906                               TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
907                               TPM2B_DATA_P_UNMARSHAL,
908                               END_OF_LIST,
909                               TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
910                               END_OF_LIST}
911  };
912  #define _RSA_EncryptDataAddress (&_RSA_EncryptData)
913  #else
914  #define _RSA_EncryptDataAddress 0
915  #endif // CC_RSA_Encrypt
916  #if CC_RSA_Decrypt
917  #include "RSA_Decrypt_fp.h"
918  typedef TPM_RC  (RSA_Decrypt_Entry)(
919      RSA_Decrypt_In              *in,
920      RSA_Decrypt_Out             *out
921  );
922  typedef const struct {
923      RSA_Decrypt_Entry       *entry;
924      UINT16                  inSize;
925      UINT16                  outSize;
926      UINT16                  offsetOfTypes;
927      UINT16                  paramOffsets[3];
928      BYTE                    types[7];
929  } RSA_Decrypt_COMMAND_DESCRIPTOR_t;
930  RSA_Decrypt_COMMAND_DESCRIPTOR_t _RSA_DecryptData = {
931      /* entry         */     &TPM2_RSA_Decrypt,
932      /* inSize        */     (UINT16)(sizeof(RSA_Decrypt_In)),
933      /* outSize       */     (UINT16)(sizeof(RSA_Decrypt_Out)),
934      /* offsetOfTypes */     offsetof(RSA_Decrypt_COMMAND_DESCRIPTOR_t, types),
935      /* offsets       */     {(UINT16)(offsetof(RSA_Decrypt_In, cipherText)),
936                               (UINT16)(offsetof(RSA_Decrypt_In, inScheme)),
937                               (UINT16)(offsetof(RSA_Decrypt_In, label))},
938      /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
939                               TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
940                               TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
941                               TPM2B_DATA_P_UNMARSHAL,
942                               END_OF_LIST,
943                               TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
944                               END_OF_LIST}
945  };
946  #define _RSA_DecryptDataAddress (&_RSA_DecryptData)
947  #else
948  #define _RSA_DecryptDataAddress 0
949  #endif // CC_RSA_Decrypt
950  #if CC_ECDH_KeyGen
951  #include "ECDH_KeyGen_fp.h"
952  typedef TPM_RC  (ECDH_KeyGen_Entry)(
953      ECDH_KeyGen_In              *in,
954      ECDH_KeyGen_Out             *out
955  );
956  typedef const struct {
957      ECDH_KeyGen_Entry       *entry;
958      UINT16                  inSize;
959      UINT16                  outSize;
960      UINT16                  offsetOfTypes;
961      UINT16                  paramOffsets[1];
```



```
962         BYTE                    types[5];
963     } ECDH_KeyGen_COMMAND_DESCRIPTOR_t;
964     ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
965         /* entry        */      &TPM2_ECDH_KeyGen,
966         /* inSize       */      (UINT16)(sizeof(ECDH_KeyGen_In)),
967         /* outSize      */      (UINT16)(sizeof(ECDH_KeyGen_Out)),
968         /* offsetOfTypes */     offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types),
969         /* offsets      */      {(UINT16)(offsetof(ECDH_KeyGen_Out, pubPoint))},
970         /* types        */      {TPMI_DH_OBJECT_H_UNMARSHAL,
971                                  END_OF_LIST,
972                                  TPM2B_ECC_POINT_P_MARSHAL,
973                                  TPM2B_ECC_POINT_P_MARSHAL,
974                                  END_OF_LIST}
975     };
976     #define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)
977     #else
978     #define _ECDH_KeyGenDataAddress 0
979     #endif // CC_ECDH_KeyGen
980     #if CC_ECDH_ZGen
981     #include "ECDH_ZGen_fp.h"
982     typedef TPM_RC  (ECDH_ZGen_Entry)(
983         ECDH_ZGen_In                *in,
984         ECDH_ZGen_Out               *out
985     );
986     typedef const struct {
987         ECDH_ZGen_Entry         *entry;
988         UINT16                  inSize;
989         UINT16                  outSize;
990         UINT16                  offsetOfTypes;
991         UINT16                  paramOffsets[1];
992         BYTE                    types[5];
993     } ECDH_ZGen_COMMAND_DESCRIPTOR_t;
994     ECDH_ZGen_COMMAND_DESCRIPTOR_t _ECDH_ZGenData = {
995         /* entry        */      &TPM2_ECDH_ZGen,
996         /* inSize       */      (UINT16)(sizeof(ECDH_ZGen_In)),
997         /* outSize      */      (UINT16)(sizeof(ECDH_ZGen_Out)),
998         /* offsetOfTypes */     offsetof(ECDH_ZGen_COMMAND_DESCRIPTOR_t, types),
999         /* offsets      */      {(UINT16)(offsetof(ECDH_ZGen_In, inPoint))},
1000        /* types        */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1001                                 TPM2B_ECC_POINT_P_UNMARSHAL,
1002                                 END_OF_LIST,
1003                                 TPM2B_ECC_POINT_P_MARSHAL,
1004                                 END_OF_LIST}
1005    };
1006    #define _ECDH_ZGenDataAddress (&_ECDH_ZGenData)
1007    #else
1008    #define _ECDH_ZGenDataAddress 0
1009    #endif // CC_ECDH_ZGen
1010    #if CC_ECC_Parameters
1011    #include "ECC_Parameters_fp.h"
1012    typedef TPM_RC  (ECC_Parameters_Entry)(
1013        ECC_Parameters_In           *in,
1014        ECC_Parameters_Out          *out
1015    );
1016    typedef const struct {
1017        ECC_Parameters_Entry    *entry;
1018        UINT16                  inSize;
1019        UINT16                  outSize;
1020        UINT16                  offsetOfTypes;
1021        BYTE                    types[4];
1022    } ECC_Parameters_COMMAND_DESCRIPTOR_t;
1023    ECC_Parameters_COMMAND_DESCRIPTOR_t _ECC_ParametersData = {
1024        /* entry        */      &TPM2_ECC_Parameters,
1025        /* inSize       */      (UINT16)(sizeof(ECC_Parameters_In)),
1026        /* outSize      */      (UINT16)(sizeof(ECC_Parameters_Out)),
1027        /* offsetOfTypes */     offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types),
```

```
1028        /* offsets       */     // No parameter offsets;
1029        /* types         */     {TPMI_ECC_CURVE_P_UNMARSHAL,
1030                                  END_OF_LIST,
1031                                  TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL,
1032                                  END_OF_LIST}
1033    };
1034    #define _ECC_ParametersDataAddress (&_ECC_ParametersData)
1035    #else
1036    #define _ECC_ParametersDataAddress 0
1037    #endif // CC_ECC_Parameters
1038    #if CC_ZGen_2Phase
1039    #include "ZGen_2Phase_fp.h"
1040    typedef TPM_RC  (ZGen_2Phase_Entry)(
1041        ZGen_2Phase_In              *in,
1042        ZGen_2Phase_Out             *out
1043    );
1044    typedef const struct {
1045        ZGen_2Phase_Entry       *entry;
1046        UINT16                  inSize;
1047        UINT16                  outSize;
1048        UINT16                  offsetOfTypes;
1049        UINT16                  paramOffsets[5];
1050        BYTE                    types[9];
1051    } ZGen_2Phase_COMMAND_DESCRIPTOR_t;
1052    ZGen_2Phase_COMMAND_DESCRIPTOR_t _ZGen_2PhaseData = {
1053        /* entry         */     &TPM2_ZGen_2Phase,
1054        /* inSize        */     (UINT16)(sizeof(ZGen_2Phase_In)),
1055        /* outSize       */     (UINT16)(sizeof(ZGen_2Phase_Out)),
1056        /* offsetOfTypes */     offsetof(ZGen_2Phase_COMMAND_DESCRIPTOR_t, types),
1057        /* offsets       */     {(UINT16)(offsetof(ZGen_2Phase_In, inQsB)),
1058                                  (UINT16)(offsetof(ZGen_2Phase_In, inQeB)),
1059                                  (UINT16)(offsetof(ZGen_2Phase_In, inScheme)),
1060                                  (UINT16)(offsetof(ZGen_2Phase_In, counter)),
1061                                  (UINT16)(offsetof(ZGen_2Phase_Out, outZ2))},
1062        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1063                                  TPM2B_ECC_POINT_P_UNMARSHAL,
1064                                  TPM2B_ECC_POINT_P_UNMARSHAL,
1065                                  TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL,
1066                                  UINT16_P_UNMARSHAL,
1067                                  END_OF_LIST,
1068                                  TPM2B_ECC_POINT_P_MARSHAL,
1069                                  TPM2B_ECC_POINT_P_MARSHAL,
1070                                  END_OF_LIST}
1071    };
1072    #define _ZGen_2PhaseDataAddress (&_ZGen_2PhaseData)
1073    #else
1074    #define _ZGen_2PhaseDataAddress 0
1075    #endif // CC_ZGen_2Phase
1076    #if CC_EncryptDecrypt
1077    #include "EncryptDecrypt_fp.h"
1078    typedef TPM_RC  (EncryptDecrypt_Entry)(
1079        EncryptDecrypt_In           *in,
1080        EncryptDecrypt_Out          *out
1081    );
1082    typedef const struct {
1083        EncryptDecrypt_Entry    *entry;
1084        UINT16                  inSize;
1085        UINT16                  outSize;
1086        UINT16                  offsetOfTypes;
1087        UINT16                  paramOffsets[5];
1088        BYTE                    types[9];
1089    } EncryptDecrypt_COMMAND_DESCRIPTOR_t;
1090    EncryptDecrypt_COMMAND_DESCRIPTOR_t _EncryptDecryptData = {
1091        /* entry         */     &TPM2_EncryptDecrypt,
1092        /* inSize        */     (UINT16)(sizeof(EncryptDecrypt_In)),
1093        /* outSize       */     (UINT16)(sizeof(EncryptDecrypt_Out)),
```



```
1094        /* offsetOfTypes */     offsetof(EncryptDecrypt_COMMAND_DESCRIPTOR_t, types),
1095        /* offsets       */     {(UINT16)(offsetof(EncryptDecrypt_In, decrypt)),
1096                                  (UINT16)(offsetof(EncryptDecrypt_In, mode)),
1097                                  (UINT16)(offsetof(EncryptDecrypt_In, ivIn)),
1098                                  (UINT16)(offsetof(EncryptDecrypt_In, inData)),
1099                                  (UINT16)(offsetof(EncryptDecrypt_Out, ivOut))},
1100        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1101                                  TPMI_YES_NO_P_UNMARSHAL,
1102                                  TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1103                                  TPM2B_IV_P_UNMARSHAL,
1104                                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
1105                                  END_OF_LIST,
1106                                  TPM2B_MAX_BUFFER_P_MARSHAL,
1107                                  TPM2B_IV_P_MARSHAL,
1108                                  END_OF_LIST}
1109    };
1110    #define _EncryptDecryptDataAddress (&_EncryptDecryptData)
1111    #else
1112    #define _EncryptDecryptDataAddress 0
1113    #endif // CC_EncryptDecrypt
1114    #if CC_EncryptDecrypt2
1115    #include "EncryptDecrypt2_fp.h"
1116    typedef TPM_RC  (EncryptDecrypt2_Entry)(
1117        EncryptDecrypt2_In          *in,
1118        EncryptDecrypt2_Out         *out
1119    );
1120    typedef const struct {
1121        EncryptDecrypt2_Entry   *entry;
1122        UINT16                  inSize;
1123        UINT16                  outSize;
1124        UINT16                  offsetOfTypes;
1125        UINT16                  paramOffsets[5];
1126        BYTE                    types[9];
1127    } EncryptDecrypt2_COMMAND_DESCRIPTOR_t;
1128    EncryptDecrypt2_COMMAND_DESCRIPTOR_t _EncryptDecrypt2Data = {
1129        /* entry         */     &TPM2_EncryptDecrypt2,
1130        /* inSize        */     (UINT16)(sizeof(EncryptDecrypt2_In)),
1131        /* outSize       */     (UINT16)(sizeof(EncryptDecrypt2_Out)),
1132        /* offsetOfTypes */     offsetof(EncryptDecrypt2_COMMAND_DESCRIPTOR_t, types),
1133        /* offsets       */     {(UINT16)(offsetof(EncryptDecrypt2_In, inData)),
1134                                  (UINT16)(offsetof(EncryptDecrypt2_In, decrypt)),
1135                                  (UINT16)(offsetof(EncryptDecrypt2_In, mode)),
1136                                  (UINT16)(offsetof(EncryptDecrypt2_In, ivIn)),
1137                                  (UINT16)(offsetof(EncryptDecrypt2_Out, ivOut))},
1138        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1139                                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
1140                                  TPMI_YES_NO_P_UNMARSHAL,
1141                                  TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1142                                  TPM2B_IV_P_UNMARSHAL,
1143                                  END_OF_LIST,
1144                                  TPM2B_MAX_BUFFER_P_MARSHAL,
1145                                  TPM2B_IV_P_MARSHAL,
1146                                  END_OF_LIST}
1147    };
1148    #define _EncryptDecrypt2DataAddress (&_EncryptDecrypt2Data)
1149    #else
1150    #define _EncryptDecrypt2DataAddress 0
1151    #endif // CC_EncryptDecrypt2
1152    #if CC_Hash
1153    #include "Hash_fp.h"
1154    typedef TPM_RC  (Hash_Entry)(
1155        Hash_In                     *in,
1156        Hash_Out                    *out
1157    );
1158    typedef const struct {
1159        Hash_Entry              *entry;
```

```
1160        UINT16                  inSize;
1161        UINT16                  outSize;
1162        UINT16                  offsetOfTypes;
1163        UINT16                  paramOffsets[3];
1164        BYTE                    types[7];
1165    } Hash_COMMAND_DESCRIPTOR_t;
1166    Hash_COMMAND_DESCRIPTOR_t _HashData = {
1167        /* entry         */     &TPM2_Hash,
1168        /* inSize        */     (UINT16)(sizeof(Hash_In)),
1169        /* outSize       */     (UINT16)(sizeof(Hash_Out)),
1170        /* offsetOfTypes */     offsetof(Hash_COMMAND_DESCRIPTOR_t, types),
1171        /* offsets       */     {(UINT16)(offsetof(Hash_In, hashAlg)),
1172                                 (UINT16)(offsetof(Hash_In, hierarchy)),
1173                                 (UINT16)(offsetof(Hash_Out, validation))},
1174        /* types         */     {TPM2B_MAX_BUFFER_P_UNMARSHAL,
1175                                 TPMI_ALG_HASH_P_UNMARSHAL,
1176                                 TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1177                                 END_OF_LIST,
1178                                 TPM2B_DIGEST_P_MARSHAL,
1179                                 TPMT_TK_HASHCHECK_P_MARSHAL,
1180                                 END_OF_LIST}
1181    };
1182    #define _HashDataAddress (&_HashData)
1183    #else
1184    #define _HashDataAddress 0
1185    #endif // CC_Hash
1186    #if CC_HMAC
1187    #include "HMAC_fp.h"
1188    typedef TPM_RC  (HMAC_Entry)(
1189        HMAC_In                     *in,
1190        HMAC_Out                    *out
1191    );
1192    typedef const struct {
1193        HMAC_Entry              *entry;
1194        UINT16                  inSize;
1195        UINT16                  outSize;
1196        UINT16                  offsetOfTypes;
1197        UINT16                  paramOffsets[2];
1198        BYTE                    types[6];
1199    } HMAC_COMMAND_DESCRIPTOR_t;
1200    HMAC_COMMAND_DESCRIPTOR_t _HMACData = {
1201        /* entry         */     &TPM2_HMAC,
1202        /* inSize        */     (UINT16)(sizeof(HMAC_In)),
1203        /* outSize       */     (UINT16)(sizeof(HMAC_Out)),
1204        /* offsetOfTypes */     offsetof(HMAC_COMMAND_DESCRIPTOR_t, types),
1205        /* offsets       */     {(UINT16)(offsetof(HMAC_In, buffer)),
1206                                 (UINT16)(offsetof(HMAC_In, hashAlg))},
1207        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1208                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1209                                 TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1210                                 END_OF_LIST,
1211                                 TPM2B_DIGEST_P_MARSHAL,
1212                                 END_OF_LIST}
1213    };
1214    #define _HMACDataAddress (&_HMACData)
1215    #else
1216    #define _HMACDataAddress 0
1217    #endif // CC_HMAC
1218    #if CC_MAC
1219    #include "MAC_fp.h"
1220    typedef TPM_RC  (MAC_Entry)(
1221        MAC_In                      *in,
1222        MAC_Out                     *out
1223    );
1224    typedef const struct {
1225        MAC_Entry               *entry;
```



```
1226        UINT16                  inSize;
1227        UINT16                  outSize;
1228        UINT16                  offsetOfTypes;
1229        UINT16                  paramOffsets[2];
1230        BYTE                    types[6];
1231    } MAC_COMMAND_DESCRIPTOR_t;
1232    MAC_COMMAND_DESCRIPTOR_t _MACData = {
1233        /* entry         */     &TPM2_MAC,
1234        /* inSize        */     (UINT16)(sizeof(MAC_In)),
1235        /* outSize       */     (UINT16)(sizeof(MAC_Out)),
1236        /* offsetOfTypes */     offsetof(MAC_COMMAND_DESCRIPTOR_t, types),
1237        /* offsets       */     {(UINT16)(offsetof(MAC_In, buffer)),
1238                                 (UINT16)(offsetof(MAC_In, inScheme))},
1239        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1240                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1241                                 TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1242                                 END_OF_LIST,
1243                                 TPM2B_DIGEST_P_MARSHAL,
1244                                 END_OF_LIST}
1245    };
1246    #define _MACDataAddress (&_MACData)
1247    #else
1248    #define _MACDataAddress 0
1249    #endif // CC_MAC
1250    #if CC_GetRandom
1251    #include "GetRandom_fp.h"
1252    typedef TPM_RC  (GetRandom_Entry)(
1253        GetRandom_In                *in,
1254        GetRandom_Out               *out
1255    );
1256    typedef const struct {
1257        GetRandom_Entry         *entry;
1258        UINT16                  inSize;
1259        UINT16                  outSize;
1260        UINT16                  offsetOfTypes;
1261        BYTE                    types[4];
1262    } GetRandom_COMMAND_DESCRIPTOR_t;
1263    GetRandom_COMMAND_DESCRIPTOR_t _GetRandomData = {
1264        /* entry         */     &TPM2_GetRandom,
1265        /* inSize        */     (UINT16)(sizeof(GetRandom_In)),
1266        /* outSize       */     (UINT16)(sizeof(GetRandom_Out)),
1267        /* offsetOfTypes */     offsetof(GetRandom_COMMAND_DESCRIPTOR_t, types),
1268        /* offsets       */     // No parameter offsets;
1269        /* types         */     {UINT16_P_UNMARSHAL,
1270                                 END_OF_LIST,
1271                                 TPM2B_DIGEST_P_MARSHAL,
1272                                 END_OF_LIST}
1273    };
1274    #define _GetRandomDataAddress (&_GetRandomData)
1275    #else
1276    #define _GetRandomDataAddress 0
1277    #endif // CC_GetRandom
1278    #if CC_StirRandom
1279    #include "StirRandom_fp.h"
1280    typedef TPM_RC  (StirRandom_Entry)(
1281        StirRandom_In               *in
1282    );
1283    typedef const struct {
1284        StirRandom_Entry        *entry;
1285        UINT16                  inSize;
1286        UINT16                  outSize;
1287        UINT16                  offsetOfTypes;
1288        BYTE                    types[3];
1289    } StirRandom_COMMAND_DESCRIPTOR_t;
1290    StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {
1291        /* entry         */     &TPM2_StirRandom,
```

```
1292        /* inSize        */     (UINT16)(sizeof(StirRandom_In)),
1293        /* outSize       */     0,
1294        /* offsetOfTypes */     offsetof(StirRandom_COMMAND_DESCRIPTOR_t, types),
1295        /* offsets       */     // No parameter offsets;
1296        /* types         */     {TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1297                                 END_OF_LIST,
1298                                 END_OF_LIST}
1299    };
1300    #define _StirRandomDataAddress (&_StirRandomData)
1301    #else
1302    #define _StirRandomDataAddress 0
1303    #endif // CC_StirRandom
1304    #if CC_HMAC_Start
1305    #include "HMAC_Start_fp.h"
1306    typedef TPM_RC  (HMAC_Start_Entry)(
1307        HMAC_Start_In               *in,
1308        HMAC_Start_Out              *out
1309    );
1310    typedef const struct {
1311        HMAC_Start_Entry        *entry;
1312        UINT16                  inSize;
1313        UINT16                  outSize;
1314        UINT16                  offsetOfTypes;
1315        UINT16                  paramOffsets[2];
1316        BYTE                    types[6];
1317    } HMAC_Start_COMMAND_DESCRIPTOR_t;
1318    HMAC_Start_COMMAND_DESCRIPTOR_t _HMAC_StartData = {
1319        /* entry         */     &TPM2_HMAC_Start,
1320        /* inSize        */     (UINT16)(sizeof(HMAC_Start_In)),
1321        /* outSize       */     (UINT16)(sizeof(HMAC_Start_Out)),
1322        /* offsetOfTypes */     offsetof(HMAC_Start_COMMAND_DESCRIPTOR_t, types),
1323        /* offsets       */     {(UINT16)(offsetof(HMAC_Start_In, auth)),
1324                                  (UINT16)(offsetof(HMAC_Start_In, hashAlg))},
1325        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1326                                 TPM2B_AUTH_P_UNMARSHAL,
1327                                 TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1328                                 END_OF_LIST,
1329                                 TPMI_DH_OBJECT_H_MARSHAL,
1330                                 END_OF_LIST}
1331    };
1332    #define _HMAC_StartDataAddress (&_HMAC_StartData)
1333    #else
1334    #define _HMAC_StartDataAddress 0
1335    #endif // CC_HMAC_Start
1336    #if CC_MAC_Start
1337    #include "MAC_Start_fp.h"
1338    typedef TPM_RC  (MAC_Start_Entry)(
1339        MAC_Start_In                *in,
1340        MAC_Start_Out               *out
1341    );
1342    typedef const struct {
1343        MAC_Start_Entry         *entry;
1344        UINT16                  inSize;
1345        UINT16                  outSize;
1346        UINT16                  offsetOfTypes;
1347        UINT16                  paramOffsets[2];
1348        BYTE                    types[6];
1349    } MAC_Start_COMMAND_DESCRIPTOR_t;
1350    MAC_Start_COMMAND_DESCRIPTOR_t _MAC_StartData = {
1351        /* entry         */     &TPM2_MAC_Start,
1352        /* inSize        */     (UINT16)(sizeof(MAC_Start_In)),
1353        /* outSize       */     (UINT16)(sizeof(MAC_Start_Out)),
1354        /* offsetOfTypes */     offsetof(MAC_Start_COMMAND_DESCRIPTOR_t, types),
1355        /* offsets       */     {(UINT16)(offsetof(MAC_Start_In, auth)),
1356                                  (UINT16)(offsetof(MAC_Start_In, inScheme))},
1357        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
```



```
1358                                TPM2B_AUTH_P_UNMARSHAL,
1359                                TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1360                                END_OF_LIST,
1361                                TPMI_DH_OBJECT_H_MARSHAL,
1362                                END_OF_LIST}
1363    };
1364    #define _MAC_StartDataAddress (&_MAC_StartData)
1365    #else
1366    #define _MAC_StartDataAddress 0
1367    #endif // CC_MAC_Start
1368    #if CC_HashSequenceStart
1369    #include "HashSequenceStart_fp.h"
1370    typedef TPM_RC  (HashSequenceStart_Entry)(
1371        HashSequenceStart_In            *in,
1372        HashSequenceStart_Out           *out
1373    );
1374    typedef const struct {
1375        HashSequenceStart_Entry     *entry;
1376        UINT16                      inSize;
1377        UINT16                      outSize;
1378        UINT16                      offsetOfTypes;
1379        UINT16                      paramOffsets[1];
1380        BYTE                        types[5];
1381    } HashSequenceStart_COMMAND_DESCRIPTOR_t;
1382    HashSequenceStart_COMMAND_DESCRIPTOR_t _HashSequenceStartData = {
1383        /* entry         */         &TPM2_HashSequenceStart,
1384        /* inSize        */         (UINT16)(sizeof(HashSequenceStart_In)),
1385        /* outSize       */         (UINT16)(sizeof(HashSequenceStart_Out)),
1386        /* offsetOfTypes */         offsetof(HashSequenceStart_COMMAND_DESCRIPTOR_t,
        types),
1387        /* offsets       */         {(UINT16)(offsetof(HashSequenceStart_In, hashAlg))},
1388        /* types         */         {TPM2B_AUTH_P_UNMARSHAL,
1389                                     TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1390                                     END_OF_LIST,
1391                                     TPMI_DH_OBJECT_H_MARSHAL,
1392                                     END_OF_LIST}
1393    };
1394    #define _HashSequenceStartDataAddress (&_HashSequenceStartData)
1395    #else
1396    #define _HashSequenceStartDataAddress 0
1397    #endif // CC_HashSequenceStart
1398    #if CC_SequenceUpdate
1399    #include "SequenceUpdate_fp.h"
1400    typedef TPM_RC  (SequenceUpdate_Entry)(
1401        SequenceUpdate_In           *in
1402    );
1403    typedef const struct {
1404        SequenceUpdate_Entry    *entry;
1405        UINT16                  inSize;
1406        UINT16                  outSize;
1407        UINT16                  offsetOfTypes;
1408        UINT16                  paramOffsets[1];
1409        BYTE                    types[4];
1410    } SequenceUpdate_COMMAND_DESCRIPTOR_t;
1411    SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
1412        /* entry         */     &TPM2_SequenceUpdate,
1413        /* inSize        */     (UINT16)(sizeof(SequenceUpdate_In)),
1414        /* outSize       */     0,
1415        /* offsetOfTypes */     offsetof(SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
1416        /* offsets       */     {(UINT16)(offsetof(SequenceUpdate_In, buffer))},
1417        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1418                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1419                                 END_OF_LIST,
1420                                 END_OF_LIST}
1421    };
1422    #define _SequenceUpdateDataAddress (&_SequenceUpdateData)
```

```
1423    #else
1424    #define _SequenceUpdateDataAddress 0
1425    #endif // CC_SequenceUpdate
1426    #if CC_SequenceComplete
1427    #include "SequenceComplete_fp.h"
1428    typedef TPM_RC  (SequenceComplete_Entry)(
1429        SequenceComplete_In         *in,
1430        SequenceComplete_Out        *out
1431    );
1432    typedef const struct {
1433        SequenceComplete_Entry  *entry;
1434        UINT16                  inSize;
1435        UINT16                  outSize;
1436        UINT16                  offsetOfTypes;
1437        UINT16                  paramOffsets[3];
1438        BYTE                    types[7];
1439    } SequenceComplete_COMMAND_DESCRIPTOR_t;
1440    SequenceComplete_COMMAND_DESCRIPTOR_t _SequenceCompleteData = {
1441        /* entry         */     &TPM2_SequenceComplete,
1442        /* inSize        */     (UINT16)(sizeof(SequenceComplete_In)),
1443        /* outSize       */     (UINT16)(sizeof(SequenceComplete_Out)),
1444        /* offsetOfTypes */     offsetof(SequenceComplete_COMMAND_DESCRIPTOR_t, types),
1445        /* offsets       */     {(UINT16)(offsetof(SequenceComplete_In, buffer)),
1446                                 (UINT16)(offsetof(SequenceComplete_In, hierarchy)),
1447                                 (UINT16)(offsetof(SequenceComplete_Out, validation))},
1448        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1449                                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1450                                 TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1451                                 END_OF_LIST,
1452                                 TPM2B_DIGEST_P_MARSHAL,
1453                                 TPMT_TK_HASHCHECK_P_MARSHAL,
1454                                 END_OF_LIST}
1455    };
1456    #define _SequenceCompleteDataAddress (&_SequenceCompleteData)
1457    #else
1458    #define _SequenceCompleteDataAddress 0
1459    #endif // CC_SequenceComplete
1460    #if CC_EventSequenceComplete
1461    #include "EventSequenceComplete_fp.h"
1462    typedef TPM_RC  (EventSequenceComplete_Entry)(
1463        EventSequenceComplete_In            *in,
1464        EventSequenceComplete_Out           *out
1465    );
1466    typedef const struct {
1467        EventSequenceComplete_Entry     *entry;
1468        UINT16                          inSize;
1469        UINT16                          outSize;
1470        UINT16                          offsetOfTypes;
1471        UINT16                          paramOffsets[2];
1472        BYTE                            types[6];
1473    } EventSequenceComplete_COMMAND_DESCRIPTOR_t;
1474    EventSequenceComplete_COMMAND_DESCRIPTOR_t _EventSequenceCompleteData = {
1475        /* entry         */             &TPM2_EventSequenceComplete,
1476        /* inSize        */             (UINT16)(sizeof(EventSequenceComplete_In)),
1477        /* outSize       */             (UINT16)(sizeof(EventSequenceComplete_Out)),
1478        /* offsetOfTypes */
        offsetof(EventSequenceComplete_COMMAND_DESCRIPTOR_t, types),
1479        /* offsets       */             {(UINT16)(offsetof(EventSequenceComplete_In,
        sequenceHandle)),
1480                                         (UINT16)(offsetof(EventSequenceComplete_In,
        buffer))},
1481        /* types         */             {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1482                                         TPMI_DH_OBJECT_H_UNMARSHAL,
1483                                         TPM2B_MAX_BUFFER_P_UNMARSHAL,
1484                                         END_OF_LIST,
1485                                         TPML_DIGEST_VALUES_P_MARSHAL,
```



```
                                        END_OF_LIST}
};
#define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)
#else
#define _EventSequenceCompleteDataAddress 0
#endif // CC_EventSequenceComplete
#if CC_Certify
#include "Certify_fp.h"
typedef TPM_RC  (Certify_Entry)(
    Certify_In                  *in,
    Certify_Out                 *out
);
typedef const struct {
    Certify_Entry           *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[4];
    BYTE                    types[8];
} Certify_COMMAND_DESCRIPTOR_t;
Certify_COMMAND_DESCRIPTOR_t _CertifyData = {
    /* entry         */     &TPM2_Certify,
    /* inSize        */     (UINT16)(sizeof(Certify_In)),
    /* outSize       */     (UINT16)(sizeof(Certify_Out)),
    /* offsetOfTypes */     offsetof(Certify_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(Certify_In, signHandle)),
                              (UINT16)(offsetof(Certify_In, qualifyingData)),
                              (UINT16)(offsetof(Certify_In, inScheme)),
                              (UINT16)(offsetof(Certify_Out, signature))},
    /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
                              TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
                              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 _CertifyDataAddress (&_CertifyData)
#else
#define _CertifyDataAddress 0
#endif // CC_Certify
#if CC_CertifyCreation
#include "CertifyCreation_fp.h"
typedef TPM_RC  (CertifyCreation_Entry)(
    CertifyCreation_In          *in,
    CertifyCreation_Out         *out
);
typedef const struct {
    CertifyCreation_Entry   *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[6];
    BYTE                    types[10];
} CertifyCreation_COMMAND_DESCRIPTOR_t;
CertifyCreation_COMMAND_DESCRIPTOR_t _CertifyCreationData = {
    /* entry         */     &TPM2_CertifyCreation,
    /* inSize        */     (UINT16)(sizeof(CertifyCreation_In)),
    /* outSize       */     (UINT16)(sizeof(CertifyCreation_Out)),
    /* offsetOfTypes */     offsetof(CertifyCreation_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */     {(UINT16)(offsetof(CertifyCreation_In, objectHandle)),
                              (UINT16)(offsetof(CertifyCreation_In, qualifyingData)),
                              (UINT16)(offsetof(CertifyCreation_In, creationHash)),
                              (UINT16)(offsetof(CertifyCreation_In, inScheme)),
                              (UINT16)(offsetof(CertifyCreation_In, creationTicket)),
```

```
1552                                (UINT16)(offsetof(CertifyCreation_Out, signature))},
1553        /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1554                                 TPMI_DH_OBJECT_H_UNMARSHAL,
1555                                 TPM2B_DATA_P_UNMARSHAL,
1556                                 TPM2B_DIGEST_P_UNMARSHAL,
1557                                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1558                                 TPMT_TK_CREATION_P_UNMARSHAL,
1559                                 END_OF_LIST,
1560                                 TPM2B_ATTEST_P_MARSHAL,
1561                                 TPMT_SIGNATURE_P_MARSHAL,
1562                                 END_OF_LIST}
1563    };
1564    #define _CertifyCreationDataAddress (&_CertifyCreationData)
1565    #else
1566    #define _CertifyCreationDataAddress 0
1567    #endif // CC_CertifyCreation
1568    #if CC_Quote
1569    #include "Quote_fp.h"
1570    typedef TPM_RC  (Quote_Entry)(
1571        Quote_In                    *in,
1572        Quote_Out                   *out
1573    );
1574    typedef const struct {
1575        Quote_Entry             *entry;
1576        UINT16                  inSize;
1577        UINT16                  outSize;
1578        UINT16                  offsetOfTypes;
1579        UINT16                  paramOffsets[4];
1580        BYTE                    types[8];
1581    } Quote_COMMAND_DESCRIPTOR_t;
1582    Quote_COMMAND_DESCRIPTOR_t _QuoteData = {
1583        /* entry          */     &TPM2_Quote,
1584        /* inSize         */     (UINT16)(sizeof(Quote_In)),
1585        /* outSize        */     (UINT16)(sizeof(Quote_Out)),
1586        /* offsetOfTypes */      offsetof(Quote_COMMAND_DESCRIPTOR_t, types),
1587        /* offsets        */     {(UINT16)(offsetof(Quote_In, qualifyingData)),
1588                                 (UINT16)(offsetof(Quote_In, inScheme)),
1589                                 (UINT16)(offsetof(Quote_In, PCRselect)),
1590                                 (UINT16)(offsetof(Quote_Out, signature))},
1591        /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1592                                 TPM2B_DATA_P_UNMARSHAL,
1593                                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1594                                 TPML_PCR_SELECTION_P_UNMARSHAL,
1595                                 END_OF_LIST,
1596                                 TPM2B_ATTEST_P_MARSHAL,
1597                                 TPMT_SIGNATURE_P_MARSHAL,
1598                                 END_OF_LIST}
1599    };
1600    #define _QuoteDataAddress (&_QuoteData)
1601    #else
1602    #define _QuoteDataAddress 0
1603    #endif // CC_Quote
1604    #if CC_GetSessionAuditDigest
1605    #include "GetSessionAuditDigest_fp.h"
1606    typedef TPM_RC  (GetSessionAuditDigest_Entry)(
1607        GetSessionAuditDigest_In            *in,
1608        GetSessionAuditDigest_Out           *out
1609    );
1610    typedef const struct {
1611        GetSessionAuditDigest_Entry     *entry;
1612        UINT16                          inSize;
1613        UINT16                          outSize;
1614        UINT16                          offsetOfTypes;
1615        UINT16                          paramOffsets[5];
1616        BYTE                            types[9];
1617    } GetSessionAuditDigest_COMMAND_DESCRIPTOR_t;
```



```
1618 GetSessionAuditDigest_COMMAND_DESCRIPTOR_t _GetSessionAuditDigestData = {
1619     /* entry         */             &TPM2_GetSessionAuditDigest,
1620     /* inSize        */             (UINT16)(sizeof(GetSessionAuditDigest_In)),
1621     /* outSize       */             (UINT16)(sizeof(GetSessionAuditDigest_Out)),
1622     /* offsetOfTypes */
     offsetof(GetSessionAuditDigest_COMMAND_DESCRIPTOR_t, types),
1623     /* offsets       */             {(UINT16)(offsetof(GetSessionAuditDigest_In,
     signHandle)),
1624                                      (UINT16)(offsetof(GetSessionAuditDigest_In,
     sessionHandle)),
1625                                      (UINT16)(offsetof(GetSessionAuditDigest_In,
     qualifyingData)),
1626                                      (UINT16)(offsetof(GetSessionAuditDigest_In,
     inScheme)),
1627                                      (UINT16)(offsetof(GetSessionAuditDigest_Out,
     signature))},
1628     /* types         */             {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1629                                      TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1630                                      TPMI_SH_HMAC_H_UNMARSHAL,
1631                                      TPM2B_DATA_P_UNMARSHAL,
1632                                      TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1633                                      END_OF_LIST,
1634                                      TPM2B_ATTEST_P_MARSHAL,
1635                                      TPMT_SIGNATURE_P_MARSHAL,
1636                                      END_OF_LIST}
1637 };
1638 #define _GetSessionAuditDigestDataAddress (&_GetSessionAuditDigestData)
1639 #else
1640 #define _GetSessionAuditDigestDataAddress 0
1641 #endif // CC_GetSessionAuditDigest
1642 #if CC_GetCommandAuditDigest
1643 #include "GetCommandAuditDigest_fp.h"
1644 typedef TPM_RC  (GetCommandAuditDigest_Entry)(
1645     GetCommandAuditDigest_In            *in,
1646     GetCommandAuditDigest_Out           *out
1647 );
1648 typedef const struct {
1649     GetCommandAuditDigest_Entry     *entry;
1650     UINT16                          inSize;
1651     UINT16                          outSize;
1652     UINT16                          offsetOfTypes;
1653     UINT16                          paramOffsets[4];
1654     BYTE                            types[8];
1655 } GetCommandAuditDigest_COMMAND_DESCRIPTOR_t;
1656 GetCommandAuditDigest_COMMAND_DESCRIPTOR_t _GetCommandAuditDigestData = {
1657     /* entry         */             &TPM2_GetCommandAuditDigest,
1658     /* inSize        */             (UINT16)(sizeof(GetCommandAuditDigest_In)),
1659     /* outSize       */             (UINT16)(sizeof(GetCommandAuditDigest_Out)),
1660     /* offsetOfTypes */
     offsetof(GetCommandAuditDigest_COMMAND_DESCRIPTOR_t, types),
1661     /* offsets       */             {(UINT16)(offsetof(GetCommandAuditDigest_In,
     signHandle)),
1662                                      (UINT16)(offsetof(GetCommandAuditDigest_In,
     qualifyingData)),
1663                                      (UINT16)(offsetof(GetCommandAuditDigest_In,
     inScheme)),
1664                                      (UINT16)(offsetof(GetCommandAuditDigest_Out,
     signature))},
1665     /* types         */             {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1666                                      TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1667                                      TPM2B_DATA_P_UNMARSHAL,
1668                                      TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1669                                      END_OF_LIST,
1670                                      TPM2B_ATTEST_P_MARSHAL,
1671                                      TPMT_SIGNATURE_P_MARSHAL,
1672                                      END_OF_LIST}
```

```
1673    };
1674    #define _GetCommandAuditDigestDataAddress (&_GetCommandAuditDigestData)
1675    #else
1676    #define _GetCommandAuditDigestDataAddress 0
1677    #endif // CC_GetCommandAuditDigest
1678    #if CC_GetTime
1679    #include "GetTime_fp.h"
1680    typedef TPM_RC  (GetTime_Entry)(
1681        GetTime_In                  *in,
1682        GetTime_Out                 *out
1683    );
1684    typedef const struct {
1685        GetTime_Entry           *entry;
1686        UINT16                  inSize;
1687        UINT16                  outSize;
1688        UINT16                  offsetOfTypes;
1689        UINT16                  paramOffsets[4];
1690        BYTE                    types[8];
1691    } GetTime_COMMAND_DESCRIPTOR_t;
1692    GetTime_COMMAND_DESCRIPTOR_t _GetTimeData = {
1693        /* entry         */     &TPM2_GetTime,
1694        /* inSize        */     (UINT16)(sizeof(GetTime_In)),
1695        /* outSize       */     (UINT16)(sizeof(GetTime_Out)),
1696        /* offsetOfTypes */     offsetof(GetTime_COMMAND_DESCRIPTOR_t, types),
1697        /* offsets       */     {(UINT16)(offsetof(GetTime_In, signHandle)),
1698                                 (UINT16)(offsetof(GetTime_In, qualifyingData)),
1699                                 (UINT16)(offsetof(GetTime_In, inScheme)),
1700                                 (UINT16)(offsetof(GetTime_Out, signature))},
1701        /* types         */     {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1702                                 TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1703                                 TPM2B_DATA_P_UNMARSHAL,
1704                                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1705                                 END_OF_LIST,
1706                                 TPM2B_ATTEST_P_MARSHAL,
1707                                 TPMT_SIGNATURE_P_MARSHAL,
1708                                 END_OF_LIST}
1709    };
1710    #define _GetTimeDataAddress (&_GetTimeData)
1711    #else
1712    #define _GetTimeDataAddress 0
1713    #endif // CC_GetTime
1714    #if CC_Commit
1715    #include "Commit_fp.h"
1716    typedef TPM_RC  (Commit_Entry)(
1717        Commit_In                   *in,
1718        Commit_Out                  *out
1719    );
1720    typedef const struct {
1721        Commit_Entry            *entry;
1722        UINT16                  inSize;
1723        UINT16                  outSize;
1724        UINT16                  offsetOfTypes;
1725        UINT16                  paramOffsets[6];
1726        BYTE                    types[10];
1727    } Commit_COMMAND_DESCRIPTOR_t;
1728    Commit_COMMAND_DESCRIPTOR_t _CommitData = {
1729        /* entry         */     &TPM2_Commit,
1730        /* inSize        */     (UINT16)(sizeof(Commit_In)),
1731        /* outSize       */     (UINT16)(sizeof(Commit_Out)),
1732        /* offsetOfTypes */     offsetof(Commit_COMMAND_DESCRIPTOR_t, types),
1733        /* offsets       */     {(UINT16)(offsetof(Commit_In, P1)),
1734                                 (UINT16)(offsetof(Commit_In, s2)),
1735                                 (UINT16)(offsetof(Commit_In, y2)),
1736                                 (UINT16)(offsetof(Commit_Out, L)),
1737                                 (UINT16)(offsetof(Commit_Out, E)),
1738                                 (UINT16)(offsetof(Commit_Out, counter))},
```



```
1739        /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1740                                    TPM2B_ECC_POINT_P_UNMARSHAL,
1741                                    TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1742                                    TPM2B_ECC_PARAMETER_P_UNMARSHAL,
1743                                    END_OF_LIST,
1744                                    TPM2B_ECC_POINT_P_MARSHAL,
1745                                    TPM2B_ECC_POINT_P_MARSHAL,
1746                                    TPM2B_ECC_POINT_P_MARSHAL,
1747                                    UINT16_P_MARSHAL,
1748                                    END_OF_LIST}
1749    };
1750    #define _CommitDataAddress (&_CommitData)
1751    #else
1752    #define _CommitDataAddress 0
1753    #endif // CC_Commit
1754    #if CC_EC_Ephemeral
1755    #include "EC_Ephemeral_fp.h"
1756    typedef TPM_RC  (EC_Ephemeral_Entry)(
1757        EC_Ephemeral_In             *in,
1758        EC_Ephemeral_Out            *out
1759    );
1760    typedef const struct {
1761        EC_Ephemeral_Entry      *entry;
1762        UINT16                  inSize;
1763        UINT16                  outSize;
1764        UINT16                  offsetOfTypes;
1765        UINT16                  paramOffsets[1];
1766        BYTE                    types[5];
1767    } EC_Ephemeral_COMMAND_DESCRIPTOR_t;
1768    EC_Ephemeral_COMMAND_DESCRIPTOR_t _EC_EphemeralData = {
1769        /* entry          */     &TPM2_EC_Ephemeral,
1770        /* inSize         */     (UINT16)(sizeof(EC_Ephemeral_In)),
1771        /* outSize        */     (UINT16)(sizeof(EC_Ephemeral_Out)),
1772        /* offsetOfTypes */      offsetof(EC_Ephemeral_COMMAND_DESCRIPTOR_t, types),
1773        /* offsets        */     {(UINT16)(offsetof(EC_Ephemeral_Out, counter))},
1774        /* types          */     {TPMI_ECC_CURVE_P_UNMARSHAL,
1775                                    END_OF_LIST,
1776                                    TPM2B_ECC_POINT_P_MARSHAL,
1777                                    UINT16_P_MARSHAL,
1778                                    END_OF_LIST}
1779    };
1780    #define _EC_EphemeralDataAddress (&_EC_EphemeralData)
1781    #else
1782    #define _EC_EphemeralDataAddress 0
1783    #endif // CC_EC_Ephemeral
1784    #if CC_VerifySignature
1785    #include "VerifySignature_fp.h"
1786    typedef TPM_RC  (VerifySignature_Entry)(
1787        VerifySignature_In          *in,
1788        VerifySignature_Out         *out
1789    );
1790    typedef const struct {
1791        VerifySignature_Entry   *entry;
1792        UINT16                  inSize;
1793        UINT16                  outSize;
1794        UINT16                  offsetOfTypes;
1795        UINT16                  paramOffsets[2];
1796        BYTE                    types[6];
1797    } VerifySignature_COMMAND_DESCRIPTOR_t;
1798    VerifySignature_COMMAND_DESCRIPTOR_t _VerifySignatureData = {
1799        /* entry          */     &TPM2_VerifySignature,
1800        /* inSize         */     (UINT16)(sizeof(VerifySignature_In)),
1801        /* outSize        */     (UINT16)(sizeof(VerifySignature_Out)),
1802        /* offsetOfTypes */      offsetof(VerifySignature_COMMAND_DESCRIPTOR_t, types),
1803        /* offsets        */     {(UINT16)(offsetof(VerifySignature_In, digest)),
1804                                    (UINT16)(offsetof(VerifySignature_In, signature))},
```

```
1805        /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1806                                   TPM2B_DIGEST_P_UNMARSHAL,
1807                                   TPMT_SIGNATURE_P_UNMARSHAL,
1808                                   END_OF_LIST,
1809                                   TPMT_TK_VERIFIED_P_MARSHAL,
1810                                   END_OF_LIST}
1811    };
1812    #define _VerifySignatureDataAddress (&_VerifySignatureData)
1813    #else
1814    #define _VerifySignatureDataAddress 0
1815    #endif // CC_VerifySignature
1816    #if CC_Sign
1817    #include "Sign_fp.h"
1818    typedef TPM_RC  (Sign_Entry)(
1819        Sign_In                     *in,
1820        Sign_Out                    *out
1821    );
1822    typedef const struct {
1823        Sign_Entry              *entry;
1824        UINT16                  inSize;
1825        UINT16                  outSize;
1826        UINT16                  offsetOfTypes;
1827        UINT16                  paramOffsets[3];
1828        BYTE                    types[7];
1829    } Sign_COMMAND_DESCRIPTOR_t;
1830    Sign_COMMAND_DESCRIPTOR_t _SignData = {
1831        /* entry          */     &TPM2_Sign,
1832        /* inSize         */     (UINT16)(sizeof(Sign_In)),
1833        /* outSize        */     (UINT16)(sizeof(Sign_Out)),
1834        /* offsetOfTypes */     offsetof(Sign_COMMAND_DESCRIPTOR_t, types),
1835        /* offsets        */     {(UINT16)(offsetof(Sign_In, digest)),
1836                                  (UINT16)(offsetof(Sign_In, inScheme)),
1837                                  (UINT16)(offsetof(Sign_In, validation))},
1838        /* types          */     {TPMI_DH_OBJECT_H_UNMARSHAL,
1839                                   TPM2B_DIGEST_P_UNMARSHAL,
1840                                   TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1841                                   TPMT_TK_HASHCHECK_P_UNMARSHAL,
1842                                   END_OF_LIST,
1843                                   TPMT_SIGNATURE_P_MARSHAL,
1844                                   END_OF_LIST}
1845    };
1846    #define _SignDataAddress (&_SignData)
1847    #else
1848    #define _SignDataAddress 0
1849    #endif // CC_Sign
1850    #if CC_SetCommandCodeAuditStatus
1851    #include "SetCommandCodeAuditStatus_fp.h"
1852    typedef TPM_RC  (SetCommandCodeAuditStatus_Entry)(
1853        SetCommandCodeAuditStatus_In            *in
1854    );
1855    typedef const struct {
1856        SetCommandCodeAuditStatus_Entry     *entry;
1857        UINT16                              inSize;
1858        UINT16                              outSize;
1859        UINT16                              offsetOfTypes;
1860        UINT16                              paramOffsets[3];
1861        BYTE                                types[6];
1862    } SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t;
1863    SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t _SetCommandCodeAuditStatusData = {
1864        /* entry          */                &TPM2_SetCommandCodeAuditStatus,
1865        /* inSize         */
        (UINT16)(sizeof(SetCommandCodeAuditStatus_In)),
1866        /* outSize        */                0,
1867        /* offsetOfTypes */
        offsetof(SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t, types),
```



```
1868        /* offsets       */
        {(UINT16)(offsetof(SetCommandCodeAuditStatus_In, auditAlg)),
1869
        (UINT16)(offsetof(SetCommandCodeAuditStatus_In, setList)),
1870
        (UINT16)(offsetof(SetCommandCodeAuditStatus_In, clearList))},
1871        /* types         */                 {TPMI_RH_PROVISION_H_UNMARSHAL,
1872                                              TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1873                                              TPML_CC_P_UNMARSHAL,
1874                                              TPML_CC_P_UNMARSHAL,
1875                                              END_OF_LIST,
1876                                              END_OF_LIST}
1877    };
1878    #define _SetCommandCodeAuditStatusDataAddress (&_SetCommandCodeAuditStatusData)
1879    #else
1880    #define _SetCommandCodeAuditStatusDataAddress 0
1881    #endif // CC_SetCommandCodeAuditStatus
1882    #if CC_PCR_Extend
1883    #include "PCR_Extend_fp.h"
1884    typedef TPM_RC  (PCR_Extend_Entry)(
1885        PCR_Extend_In               *in
1886    );
1887    typedef const struct {
1888        PCR_Extend_Entry        *entry;
1889        UINT16                  inSize;
1890        UINT16                  outSize;
1891        UINT16                  offsetOfTypes;
1892        UINT16                  paramOffsets[1];
1893        BYTE                    types[4];
1894    } PCR_Extend_COMMAND_DESCRIPTOR_t;
1895    PCR_Extend_COMMAND_DESCRIPTOR_t _PCR_ExtendData = {
1896        /* entry         */     &TPM2_PCR_Extend,
1897        /* inSize        */     (UINT16)(sizeof(PCR_Extend_In)),
1898        /* outSize       */     0,
1899        /* offsetOfTypes */     offsetof(PCR_Extend_COMMAND_DESCRIPTOR_t, types),
1900        /* offsets       */     {(UINT16)(offsetof(PCR_Extend_In, digests))},
1901        /* types         */     {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1902                                 TPML_DIGEST_VALUES_P_UNMARSHAL,
1903                                 END_OF_LIST,
1904                                 END_OF_LIST}
1905    };
1906    #define _PCR_ExtendDataAddress (&_PCR_ExtendData)
1907    #else
1908    #define _PCR_ExtendDataAddress 0
1909    #endif // CC_PCR_Extend
1910    #if CC_PCR_Event
1911    #include "PCR_Event_fp.h"
1912    typedef TPM_RC  (PCR_Event_Entry)(
1913        PCR_Event_In                *in,
1914        PCR_Event_Out               *out
1915    );
1916    typedef const struct {
1917        PCR_Event_Entry         *entry;
1918        UINT16                  inSize;
1919        UINT16                  outSize;
1920        UINT16                  offsetOfTypes;
1921        UINT16                  paramOffsets[1];
1922        BYTE                    types[5];
1923    } PCR_Event_COMMAND_DESCRIPTOR_t;
1924    PCR_Event_COMMAND_DESCRIPTOR_t _PCR_EventData = {
1925        /* entry         */     &TPM2_PCR_Event,
1926        /* inSize        */     (UINT16)(sizeof(PCR_Event_In)),
1927        /* outSize       */     (UINT16)(sizeof(PCR_Event_Out)),
1928        /* offsetOfTypes */     offsetof(PCR_Event_COMMAND_DESCRIPTOR_t, types),
1929        /* offsets       */     {(UINT16)(offsetof(PCR_Event_In, eventData))},
1930        /* types         */     {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
```

```
1931                                TPM2B_EVENT_P_UNMARSHAL,
1932                                END_OF_LIST,
1933                                TPML_DIGEST_VALUES_P_MARSHAL,
1934                                END_OF_LIST}
1935    };
1936    #define _PCR_EventDataAddress (&_PCR_EventData)
1937    #else
1938    #define _PCR_EventDataAddress 0
1939    #endif // CC_PCR_Event
1940    #if CC_PCR_Read
1941    #include "PCR_Read_fp.h"
1942    typedef TPM_RC  (PCR_Read_Entry)(
1943        PCR_Read_In                 *in,
1944        PCR_Read_Out                *out
1945    );
1946    typedef const struct {
1947        PCR_Read_Entry          *entry;
1948        UINT16                  inSize;
1949        UINT16                  outSize;
1950        UINT16                  offsetOfTypes;
1951        UINT16                  paramOffsets[2];
1952        BYTE                    types[6];
1953    } PCR_Read_COMMAND_DESCRIPTOR_t;
1954    PCR_Read_COMMAND_DESCRIPTOR_t _PCR_ReadData = {
1955        /* entry         */     &TPM2_PCR_Read,
1956        /* inSize        */     (UINT16)(sizeof(PCR_Read_In)),
1957        /* outSize       */     (UINT16)(sizeof(PCR_Read_Out)),
1958        /* offsetOfTypes */     offsetof(PCR_Read_COMMAND_DESCRIPTOR_t, types),
1959        /* offsets       */     {(UINT16)(offsetof(PCR_Read_Out, pcrSelectionOut)),
1960                                 (UINT16)(offsetof(PCR_Read_Out, pcrValues))},
1961        /* types         */     {TPML_PCR_SELECTION_P_UNMARSHAL,
1962                                 END_OF_LIST,
1963                                 UINT32_P_MARSHAL,
1964                                 TPML_PCR_SELECTION_P_MARSHAL,
1965                                 TPML_DIGEST_P_MARSHAL,
1966                                 END_OF_LIST}
1967    };
1968    #define _PCR_ReadDataAddress (&_PCR_ReadData)
1969    #else
1970    #define _PCR_ReadDataAddress 0
1971    #endif // CC_PCR_Read
1972    #if CC_PCR_Allocate
1973    #include "PCR_Allocate_fp.h"
1974    typedef TPM_RC  (PCR_Allocate_Entry)(
1975        PCR_Allocate_In             *in,
1976        PCR_Allocate_Out            *out
1977    );
1978    typedef const struct {
1979        PCR_Allocate_Entry      *entry;
1980        UINT16                  inSize;
1981        UINT16                  outSize;
1982        UINT16                  offsetOfTypes;
1983        UINT16                  paramOffsets[4];
1984        BYTE                    types[8];
1985    } PCR_Allocate_COMMAND_DESCRIPTOR_t;
1986    PCR_Allocate_COMMAND_DESCRIPTOR_t _PCR_AllocateData = {
1987        /* entry         */     &TPM2_PCR_Allocate,
1988        /* inSize        */     (UINT16)(sizeof(PCR_Allocate_In)),
1989        /* outSize       */     (UINT16)(sizeof(PCR_Allocate_Out)),
1990        /* offsetOfTypes */     offsetof(PCR_Allocate_COMMAND_DESCRIPTOR_t, types),
1991        /* offsets       */     {(UINT16)(offsetof(PCR_Allocate_In, pcrAllocation)),
1992                                 (UINT16)(offsetof(PCR_Allocate_Out, maxPCR)),
1993                                 (UINT16)(offsetof(PCR_Allocate_Out, sizeNeeded)),
1994                                 (UINT16)(offsetof(PCR_Allocate_Out, sizeAvailable))},
1995        /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
1996                                 TPML_PCR_SELECTION_P_UNMARSHAL,
```



```
1997                                END_OF_LIST,
1998                                TPMI_YES_NO_P_MARSHAL,
1999                                UINT32_P_MARSHAL,
2000                                UINT32_P_MARSHAL,
2001                                UINT32_P_MARSHAL,
2002                                END_OF_LIST}
2003    };
2004    #define _PCR_AllocateDataAddress (&_PCR_AllocateData)
2005    #else
2006    #define _PCR_AllocateDataAddress 0
2007    #endif // CC_PCR_Allocate
2008    #if CC_PCR_SetAuthPolicy
2009    #include "PCR_SetAuthPolicy_fp.h"
2010    typedef TPM_RC  (PCR_SetAuthPolicy_Entry)(
2011        PCR_SetAuthPolicy_In            *in
2012    );
2013    typedef const struct {
2014        PCR_SetAuthPolicy_Entry     *entry;
2015        UINT16                      inSize;
2016        UINT16                      outSize;
2017        UINT16                      offsetOfTypes;
2018        UINT16                      paramOffsets[3];
2019        BYTE                        types[6];
2020    } PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t;
2021    PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t _PCR_SetAuthPolicyData = {
2022        /* entry         */         &TPM2_PCR_SetAuthPolicy,
2023        /* inSize        */         (UINT16)(sizeof(PCR_SetAuthPolicy_In)),
2024        /* outSize       */         0,
2025        /* offsetOfTypes */         offsetof(PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t,
        types),
2026        /* offsets       */         {(UINT16)(offsetof(PCR_SetAuthPolicy_In, authPolicy)),
2027                                     (UINT16)(offsetof(PCR_SetAuthPolicy_In, hashAlg)),
2028                                     (UINT16)(offsetof(PCR_SetAuthPolicy_In, pcrNum))},
2029        /* types         */         {TPMI_RH_PLATFORM_H_UNMARSHAL,
2030                                     TPM2B_DIGEST_P_UNMARSHAL,
2031                                     TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2032                                     TPMI_DH_PCR_P_UNMARSHAL,
2033                                     END_OF_LIST,
2034                                     END_OF_LIST}
2035    };
2036    #define _PCR_SetAuthPolicyDataAddress (&_PCR_SetAuthPolicyData)
2037    #else
2038    #define _PCR_SetAuthPolicyDataAddress 0
2039    #endif // CC_PCR_SetAuthPolicy
2040    #if CC_PCR_SetAuthValue
2041    #include "PCR_SetAuthValue_fp.h"
2042    typedef TPM_RC  (PCR_SetAuthValue_Entry)(
2043        PCR_SetAuthValue_In         *in
2044    );
2045    typedef const struct {
2046        PCR_SetAuthValue_Entry  *entry;
2047        UINT16                  inSize;
2048        UINT16                  outSize;
2049        UINT16                  offsetOfTypes;
2050        UINT16                  paramOffsets[1];
2051        BYTE                    types[4];
2052    } PCR_SetAuthValue_COMMAND_DESCRIPTOR_t;
2053    PCR_SetAuthValue_COMMAND_DESCRIPTOR_t _PCR_SetAuthValueData = {
2054        /* entry         */     &TPM2_PCR_SetAuthValue,
2055        /* inSize        */     (UINT16)(sizeof(PCR_SetAuthValue_In)),
2056        /* outSize       */     0,
2057        /* offsetOfTypes */     offsetof(PCR_SetAuthValue_COMMAND_DESCRIPTOR_t, types),
2058        /* offsets       */     {(UINT16)(offsetof(PCR_SetAuthValue_In, auth))},
2059        /* types         */     {TPMI_DH_PCR_H_UNMARSHAL,
2060                                 TPM2B_DIGEST_P_UNMARSHAL,
2061                                 END_OF_LIST,
```



```
2062                                END_OF_LIST}
2063    };
2064    #define _PCR_SetAuthValueDataAddress (&_PCR_SetAuthValueData)
2065    #else
2066    #define _PCR_SetAuthValueDataAddress 0
2067    #endif // CC_PCR_SetAuthValue
2068    #if CC_PCR_Reset
2069    #include "PCR_Reset_fp.h"
2070    typedef TPM_RC  (PCR_Reset_Entry)(
2071        PCR_Reset_In                *in
2072    );
2073    typedef const struct {
2074        PCR_Reset_Entry         *entry;
2075        UINT16                  inSize;
2076        UINT16                  outSize;
2077        UINT16                  offsetOfTypes;
2078        BYTE                    types[3];
2079    } PCR_Reset_COMMAND_DESCRIPTOR_t;
2080    PCR_Reset_COMMAND_DESCRIPTOR_t _PCR_ResetData = {
2081        /* entry         */     &TPM2_PCR_Reset,
2082        /* inSize        */     (UINT16)(sizeof(PCR_Reset_In)),
2083        /* outSize       */     0,
2084        /* offsetOfTypes */     offsetof(PCR_Reset_COMMAND_DESCRIPTOR_t, types),
2085        /* offsets       */     // No parameter offsets;
2086        /* types         */     {TPMI_DH_PCR_H_UNMARSHAL,
2087                                 END_OF_LIST,
2088                                 END_OF_LIST}
2089    };
2090    #define _PCR_ResetDataAddress (&_PCR_ResetData)
2091    #else
2092    #define _PCR_ResetDataAddress 0
2093    #endif // CC_PCR_Reset
2094    #if CC_PolicySigned
2095    #include "PolicySigned_fp.h"
2096    typedef TPM_RC  (PolicySigned_Entry)(
2097        PolicySigned_In             *in,
2098        PolicySigned_Out            *out
2099    );
2100    typedef const struct {
2101        PolicySigned_Entry      *entry;
2102        UINT16                  inSize;
2103        UINT16                  outSize;
2104        UINT16                  offsetOfTypes;
2105        UINT16                  paramOffsets[7];
2106        BYTE                    types[11];
2107    } PolicySigned_COMMAND_DESCRIPTOR_t;
2108    PolicySigned_COMMAND_DESCRIPTOR_t _PolicySignedData = {
2109        /* entry         */     &TPM2_PolicySigned,
2110        /* inSize        */     (UINT16)(sizeof(PolicySigned_In)),
2111        /* outSize       */     (UINT16)(sizeof(PolicySigned_Out)),
2112        /* offsetOfTypes */     offsetof(PolicySigned_COMMAND_DESCRIPTOR_t, types),
2113        /* offsets       */     {(UINT16)(offsetof(PolicySigned_In, policySession)),
2114                                 (UINT16)(offsetof(PolicySigned_In, nonceTPM)),
2115                                 (UINT16)(offsetof(PolicySigned_In, cpHashA)),
2116                                 (UINT16)(offsetof(PolicySigned_In, policyRef)),
2117                                 (UINT16)(offsetof(PolicySigned_In, expiration)),
2118                                 (UINT16)(offsetof(PolicySigned_In, auth)),
2119                                 (UINT16)(offsetof(PolicySigned_Out, policyTicket))},
2120        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL,
2121                                 TPMI_SH_POLICY_H_UNMARSHAL,
2122                                 TPM2B_NONCE_P_UNMARSHAL,
2123                                 TPM2B_DIGEST_P_UNMARSHAL,
2124                                 TPM2B_NONCE_P_UNMARSHAL,
2125                                 INT32_P_UNMARSHAL,
2126                                 TPMT_SIGNATURE_P_UNMARSHAL,
2127                                 END_OF_LIST,
```



```
2128                                TPM2B_TIMEOUT_P_MARSHAL,
2129                                TPMT_TK_AUTH_P_MARSHAL,
2130                                END_OF_LIST}
2131    };
2132    #define _PolicySignedDataAddress (&_PolicySignedData)
2133    #else
2134    #define _PolicySignedDataAddress 0
2135    #endif // CC_PolicySigned
2136    #if CC_PolicySecret
2137    #include "PolicySecret_fp.h"
2138    typedef TPM_RC  (PolicySecret_Entry)(
2139        PolicySecret_In             *in,
2140        PolicySecret_Out            *out
2141    );
2142    typedef const struct {
2143        PolicySecret_Entry      *entry;
2144        UINT16                  inSize;
2145        UINT16                  outSize;
2146        UINT16                  offsetOfTypes;
2147        UINT16                  paramOffsets[6];
2148        BYTE                    types[10];
2149    } PolicySecret_COMMAND_DESCRIPTOR_t;
2150    PolicySecret_COMMAND_DESCRIPTOR_t _PolicySecretData = {
2151        /* entry         */     &TPM2_PolicySecret,
2152        /* inSize        */     (UINT16)(sizeof(PolicySecret_In)),
2153        /* outSize       */     (UINT16)(sizeof(PolicySecret_Out)),
2154        /* offsetOfTypes */     offsetof(PolicySecret_COMMAND_DESCRIPTOR_t, types),
2155        /* offsets       */     {(UINT16)(offsetof(PolicySecret_In, policySession)),
2156                                 (UINT16)(offsetof(PolicySecret_In, nonceTPM)),
2157                                 (UINT16)(offsetof(PolicySecret_In, cpHashA)),
2158                                 (UINT16)(offsetof(PolicySecret_In, policyRef)),
2159                                 (UINT16)(offsetof(PolicySecret_In, expiration)),
2160                                 (UINT16)(offsetof(PolicySecret_Out, policyTicket))},
2161        /* types         */     {TPMI_DH_ENTITY_H_UNMARSHAL,
2162                                 TPMI_SH_POLICY_H_UNMARSHAL,
2163                                 TPM2B_NONCE_P_UNMARSHAL,
2164                                 TPM2B_DIGEST_P_UNMARSHAL,
2165                                 TPM2B_NONCE_P_UNMARSHAL,
2166                                 INT32_P_UNMARSHAL,
2167                                 END_OF_LIST,
2168                                 TPM2B_TIMEOUT_P_MARSHAL,
2169                                 TPMT_TK_AUTH_P_MARSHAL,
2170                                 END_OF_LIST}
2171    };
2172    #define _PolicySecretDataAddress (&_PolicySecretData)
2173    #else
2174    #define _PolicySecretDataAddress 0
2175    #endif // CC_PolicySecret
2176    #if CC_PolicyTicket
2177    #include "PolicyTicket_fp.h"
2178    typedef TPM_RC  (PolicyTicket_Entry)(
2179        PolicyTicket_In             *in
2180    );
2181    typedef const struct {
2182        PolicyTicket_Entry      *entry;
2183        UINT16                  inSize;
2184        UINT16                  outSize;
2185        UINT16                  offsetOfTypes;
2186        UINT16                  paramOffsets[5];
2187        BYTE                    types[8];
2188    } PolicyTicket_COMMAND_DESCRIPTOR_t;
2189    PolicyTicket_COMMAND_DESCRIPTOR_t _PolicyTicketData = {
2190        /* entry         */     &TPM2_PolicyTicket,
2191        /* inSize        */     (UINT16)(sizeof(PolicyTicket_In)),
2192        /* outSize       */     0,
2193        /* offsetOfTypes */     offsetof(PolicyTicket_COMMAND_DESCRIPTOR_t, types),
```

```
2194        /* offsets       */     {(UINT16)(offsetof(PolicyTicket_In, timeout)),
2195                                  (UINT16)(offsetof(PolicyTicket_In, cpHashA)),
2196                                  (UINT16)(offsetof(PolicyTicket_In, policyRef)),
2197                                  (UINT16)(offsetof(PolicyTicket_In, authName)),
2198                                  (UINT16)(offsetof(PolicyTicket_In, ticket))},
2199        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2200                                  TPM2B_TIMEOUT_P_UNMARSHAL,
2201                                  TPM2B_DIGEST_P_UNMARSHAL,
2202                                  TPM2B_NONCE_P_UNMARSHAL,
2203                                  TPM2B_NAME_P_UNMARSHAL,
2204                                  TPMT_TK_AUTH_P_UNMARSHAL,
2205                                  END_OF_LIST,
2206                                  END_OF_LIST}
2207    };
2208    #define _PolicyTicketDataAddress (&_PolicyTicketData)
2209    #else
2210    #define _PolicyTicketDataAddress 0
2211    #endif // CC_PolicyTicket
2212    #if CC_PolicyOR
2213    #include "PolicyOR_fp.h"
2214    typedef TPM_RC  (PolicyOR_Entry)(
2215        PolicyOR_In                 *in
2216    );
2217    typedef const struct {
2218        PolicyOR_Entry          *entry;
2219        UINT16                  inSize;
2220        UINT16                  outSize;
2221        UINT16                  offsetOfTypes;
2222        UINT16                  paramOffsets[1];
2223        BYTE                    types[4];
2224    } PolicyOR_COMMAND_DESCRIPTOR_t;
2225    PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
2226        /* entry         */     &TPM2_PolicyOR,
2227        /* inSize        */     (UINT16)(sizeof(PolicyOR_In)),
2228        /* outSize       */     0,
2229        /* offsetOfTypes */     offsetof(PolicyOR_COMMAND_DESCRIPTOR_t, types),
2230        /* offsets       */     {(UINT16)(offsetof(PolicyOR_In, pHashList))},
2231        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2232                                  TPML_DIGEST_P_UNMARSHAL,
2233                                  END_OF_LIST,
2234                                  END_OF_LIST}
2235    };
2236    #define _PolicyORDataAddress (&_PolicyORData)
2237    #else
2238    #define _PolicyORDataAddress 0
2239    #endif // CC_PolicyOR
2240    #if CC_PolicyPCR
2241    #include "PolicyPCR_fp.h"
2242    typedef TPM_RC  (PolicyPCR_Entry)(
2243        PolicyPCR_In                *in
2244    );
2245    typedef const struct {
2246        PolicyPCR_Entry         *entry;
2247        UINT16                  inSize;
2248        UINT16                  outSize;
2249        UINT16                  offsetOfTypes;
2250        UINT16                  paramOffsets[2];
2251        BYTE                    types[5];
2252    } PolicyPCR_COMMAND_DESCRIPTOR_t;
2253    PolicyPCR_COMMAND_DESCRIPTOR_t _PolicyPCRData = {
2254        /* entry         */     &TPM2_PolicyPCR,
2255        /* inSize        */     (UINT16)(sizeof(PolicyPCR_In)),
2256        /* outSize       */     0,
2257        /* offsetOfTypes */     offsetof(PolicyPCR_COMMAND_DESCRIPTOR_t, types),
2258        /* offsets       */     {(UINT16)(offsetof(PolicyPCR_In, pcrDigest)),
2259                                  (UINT16)(offsetof(PolicyPCR_In, pcrs))},
```



```
2260        /* types          */     {TPMI_SH_POLICY_H_UNMARSHAL,
2261                                   TPM2B_DIGEST_P_UNMARSHAL,
2262                                   TPML_PCR_SELECTION_P_UNMARSHAL,
2263                                   END_OF_LIST,
2264                                   END_OF_LIST}
2265    };
2266    #define _PolicyPCRDataAddress (&_PolicyPCRData)
2267    #else
2268    #define _PolicyPCRDataAddress 0
2269    #endif // CC_PolicyPCR
2270    #if CC_PolicyLocality
2271    #include "PolicyLocality_fp.h"
2272    typedef TPM_RC  (PolicyLocality_Entry)(
2273        PolicyLocality_In           *in
2274    );
2275    typedef const struct {
2276        PolicyLocality_Entry    *entry;
2277        UINT16                  inSize;
2278        UINT16                  outSize;
2279        UINT16                  offsetOfTypes;
2280        UINT16                  paramOffsets[1];
2281        BYTE                    types[4];
2282    } PolicyLocality_COMMAND_DESCRIPTOR_t;
2283    PolicyLocality_COMMAND_DESCRIPTOR_t _PolicyLocalityData = {
2284        /* entry          */     &TPM2_PolicyLocality,
2285        /* inSize         */     (UINT16)(sizeof(PolicyLocality_In)),
2286        /* outSize        */     0,
2287        /* offsetOfTypes */      offsetof(PolicyLocality_COMMAND_DESCRIPTOR_t, types),
2288        /* offsets        */     {(UINT16)(offsetof(PolicyLocality_In, locality))},
2289        /* types          */     {TPMI_SH_POLICY_H_UNMARSHAL,
2290                                   TPMA_LOCALITY_P_UNMARSHAL,
2291                                   END_OF_LIST,
2292                                   END_OF_LIST}
2293    };
2294    #define _PolicyLocalityDataAddress (&_PolicyLocalityData)
2295    #else
2296    #define _PolicyLocalityDataAddress 0
2297    #endif // CC_PolicyLocality
2298    #if CC_PolicyNV
2299    #include "PolicyNV_fp.h"
2300    typedef TPM_RC  (PolicyNV_Entry)(
2301        PolicyNV_In                 *in
2302    );
2303    typedef const struct {
2304        PolicyNV_Entry          *entry;
2305        UINT16                  inSize;
2306        UINT16                  outSize;
2307        UINT16                  offsetOfTypes;
2308        UINT16                  paramOffsets[5];
2309        BYTE                    types[8];
2310    } PolicyNV_COMMAND_DESCRIPTOR_t;
2311    PolicyNV_COMMAND_DESCRIPTOR_t _PolicyNVData = {
2312        /* entry          */     &TPM2_PolicyNV,
2313        /* inSize         */     (UINT16)(sizeof(PolicyNV_In)),
2314        /* outSize        */     0,
2315        /* offsetOfTypes */      offsetof(PolicyNV_COMMAND_DESCRIPTOR_t, types),
2316        /* offsets        */     {(UINT16)(offsetof(PolicyNV_In, nvIndex)),
2317                                   (UINT16)(offsetof(PolicyNV_In, policySession)),
2318                                   (UINT16)(offsetof(PolicyNV_In, operandB)),
2319                                   (UINT16)(offsetof(PolicyNV_In, offset)),
2320                                   (UINT16)(offsetof(PolicyNV_In, operation))},
2321        /* types          */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2322                                   TPMI_RH_NV_INDEX_H_UNMARSHAL,
2323                                   TPMI_SH_POLICY_H_UNMARSHAL,
2324                                   TPM2B_OPERAND_P_UNMARSHAL,
2325                                   UINT16_P_UNMARSHAL,
```

```
2326                                TPM_EO_P_UNMARSHAL,
2327                                END_OF_LIST,
2328                                END_OF_LIST}
2329    };
2330    #define _PolicyNVDataAddress (&_PolicyNVData)
2331    #else
2332    #define _PolicyNVDataAddress 0
2333    #endif // CC_PolicyNV
2334    #if CC_PolicyCounterTimer
2335    #include "PolicyCounterTimer_fp.h"
2336    typedef TPM_RC  (PolicyCounterTimer_Entry)(
2337        PolicyCounterTimer_In           *in
2338    );
2339    typedef const struct {
2340        PolicyCounterTimer_Entry    *entry;
2341        UINT16                      inSize;
2342        UINT16                      outSize;
2343        UINT16                      offsetOfTypes;
2344        UINT16                      paramOffsets[3];
2345        BYTE                        types[6];
2346    } PolicyCounterTimer_COMMAND_DESCRIPTOR_t;
2347    PolicyCounterTimer_COMMAND_DESCRIPTOR_t _PolicyCounterTimerData = {
2348        /* entry         */         &TPM2_PolicyCounterTimer,
2349        /* inSize        */         (UINT16)(sizeof(PolicyCounterTimer_In)),
2350        /* outSize       */         0,
2351        /* offsetOfTypes */         offsetof(PolicyCounterTimer_COMMAND_DESCRIPTOR_t,
        types),
2352        /* offsets       */         {(UINT16)(offsetof(PolicyCounterTimer_In, operandB)),
2353                                     (UINT16)(offsetof(PolicyCounterTimer_In, offset)),
2354                                     (UINT16)(offsetof(PolicyCounterTimer_In,
        operation))},
2355        /* types         */         {TPMI_SH_POLICY_H_UNMARSHAL,
2356                                     TPM2B_OPERAND_P_UNMARSHAL,
2357                                     UINT16_P_UNMARSHAL,
2358                                     TPM_EO_P_UNMARSHAL,
2359                                     END_OF_LIST,
2360                                     END_OF_LIST}
2361    };
2362    #define _PolicyCounterTimerDataAddress (&_PolicyCounterTimerData)
2363    #else
2364    #define _PolicyCounterTimerDataAddress 0
2365    #endif // CC_PolicyCounterTimer
2366    #if CC_PolicyCommandCode
2367    #include "PolicyCommandCode_fp.h"
2368    typedef TPM_RC  (PolicyCommandCode_Entry)(
2369        PolicyCommandCode_In            *in
2370    );
2371    typedef const struct {
2372        PolicyCommandCode_Entry     *entry;
2373        UINT16                      inSize;
2374        UINT16                      outSize;
2375        UINT16                      offsetOfTypes;
2376        UINT16                      paramOffsets[1];
2377        BYTE                        types[4];
2378    } PolicyCommandCode_COMMAND_DESCRIPTOR_t;
2379    PolicyCommandCode_COMMAND_DESCRIPTOR_t _PolicyCommandCodeData = {
2380        /* entry         */         &TPM2_PolicyCommandCode,
2381        /* inSize        */         (UINT16)(sizeof(PolicyCommandCode_In)),
2382        /* outSize       */         0,
2383        /* offsetOfTypes */         offsetof(PolicyCommandCode_COMMAND_DESCRIPTOR_t,
        types),
2384        /* offsets       */         {(UINT16)(offsetof(PolicyCommandCode_In, code))},
2385        /* types         */         {TPMI_SH_POLICY_H_UNMARSHAL,
2386                                     TPM_CC_P_UNMARSHAL,
2387                                     END_OF_LIST,
2388                                     END_OF_LIST}
```



```
2389    };
2390    #define _PolicyCommandCodeDataAddress (&_PolicyCommandCodeData)
2391    #else
2392    #define _PolicyCommandCodeDataAddress 0
2393    #endif // CC_PolicyCommandCode
2394    #if CC_PolicyPhysicalPresence
2395    #include "PolicyPhysicalPresence_fp.h"
2396    typedef TPM_RC  (PolicyPhysicalPresence_Entry)(
2397        PolicyPhysicalPresence_In           *in
2398    );
2399    typedef const struct {
2400        PolicyPhysicalPresence_Entry    *entry;
2401        UINT16                          inSize;
2402        UINT16                          outSize;
2403        UINT16                          offsetOfTypes;
2404        BYTE                            types[3];
2405    } PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t;
2406    PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t _PolicyPhysicalPresenceData = {
2407        /* entry         */             &TPM2_PolicyPhysicalPresence,
2408        /* inSize        */             (UINT16)(sizeof(PolicyPhysicalPresence_In)),
2409        /* outSize       */             0,
2410        /* offsetOfTypes */
        offsetof(PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t, types),
2411        /* offsets       */             // No parameter offsets;
2412        /* types         */             {TPMI_SH_POLICY_H_UNMARSHAL,
2413                                         END_OF_LIST,
2414                                         END_OF_LIST}
2415    };
2416    #define _PolicyPhysicalPresenceDataAddress (&_PolicyPhysicalPresenceData)
2417    #else
2418    #define _PolicyPhysicalPresenceDataAddress 0
2419    #endif // CC_PolicyPhysicalPresence
2420    #if CC_PolicyCpHash
2421    #include "PolicyCpHash_fp.h"
2422    typedef TPM_RC  (PolicyCpHash_Entry)(
2423        PolicyCpHash_In             *in
2424    );
2425    typedef const struct {
2426        PolicyCpHash_Entry      *entry;
2427        UINT16                  inSize;
2428        UINT16                  outSize;
2429        UINT16                  offsetOfTypes;
2430        UINT16                  paramOffsets[1];
2431        BYTE                    types[4];
2432    } PolicyCpHash_COMMAND_DESCRIPTOR_t;
2433    PolicyCpHash_COMMAND_DESCRIPTOR_t _PolicyCpHashData = {
2434        /* entry         */     &TPM2_PolicyCpHash,
2435        /* inSize        */     (UINT16)(sizeof(PolicyCpHash_In)),
2436        /* outSize       */     0,
2437        /* offsetOfTypes */     offsetof(PolicyCpHash_COMMAND_DESCRIPTOR_t, types),
2438        /* offsets       */     {(UINT16)(offsetof(PolicyCpHash_In, cpHashA))},
2439        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2440                                 TPM2B_DIGEST_P_UNMARSHAL,
2441                                 END_OF_LIST,
2442                                 END_OF_LIST}
2443    };
2444    #define _PolicyCpHashDataAddress (&_PolicyCpHashData)
2445    #else
2446    #define _PolicyCpHashDataAddress 0
2447    #endif // CC_PolicyCpHash
2448    #if CC_PolicyNameHash
2449    #include "PolicyNameHash_fp.h"
2450    typedef TPM_RC  (PolicyNameHash_Entry)(
2451        PolicyNameHash_In           *in
2452    );
2453    typedef const struct {
```

```
2454        PolicyNameHash_Entry    *entry;
2455        UINT16                  inSize;
2456        UINT16                  outSize;
2457        UINT16                  offsetOfTypes;
2458        UINT16                  paramOffsets[1];
2459        BYTE                    types[4];
2460    } PolicyNameHash_COMMAND_DESCRIPTOR_t;
2461    PolicyNameHash_COMMAND_DESCRIPTOR_t _PolicyNameHashData = {
2462        /* entry         */     &TPM2_PolicyNameHash,
2463        /* inSize        */     (UINT16)(sizeof(PolicyNameHash_In)),
2464        /* outSize       */     0,
2465        /* offsetOfTypes */     offsetof(PolicyNameHash_COMMAND_DESCRIPTOR_t, types),
2466        /* offsets       */     {(UINT16)(offsetof(PolicyNameHash_In, nameHash))},
2467        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2468                                 TPM2B_DIGEST_P_UNMARSHAL,
2469                                 END_OF_LIST,
2470                                 END_OF_LIST}
2471    };
2472    #define _PolicyNameHashDataAddress (&_PolicyNameHashData)
2473    #else
2474    #define _PolicyNameHashDataAddress 0
2475    #endif // CC_PolicyNameHash
2476    #if CC_PolicyDuplicationSelect
2477    #include "PolicyDuplicationSelect_fp.h"
2478    typedef TPM_RC  (PolicyDuplicationSelect_Entry)(
2479        PolicyDuplicationSelect_In          *in
2480    );
2481    typedef const struct {
2482        PolicyDuplicationSelect_Entry   *entry;
2483        UINT16                          inSize;
2484        UINT16                          outSize;
2485        UINT16                          offsetOfTypes;
2486        UINT16                          paramOffsets[3];
2487        BYTE                            types[6];
2488    } PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t;
2489    PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t _PolicyDuplicationSelectData = {
2490        /* entry         */                 &TPM2_PolicyDuplicationSelect,
2491        /* inSize        */                 (UINT16)(sizeof(PolicyDuplicationSelect_In)),
2492        /* outSize       */                 0,
2493        /* offsetOfTypes */
        offsetof(PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t, types),
2494        /* offsets       */                 {(UINT16)(offsetof(PolicyDuplicationSelect_In,
        objectName)),
2495                                              (UINT16)(offsetof(PolicyDuplicationSelect_In,
        newParentName)),
2496                                              (UINT16)(offsetof(PolicyDuplicationSelect_In,
        includeObject))},
2497        /* types         */                 {TPMI_SH_POLICY_H_UNMARSHAL,
2498                                              TPM2B_NAME_P_UNMARSHAL,
2499                                              TPM2B_NAME_P_UNMARSHAL,
2500                                              TPMI_YES_NO_P_UNMARSHAL,
2501                                              END_OF_LIST,
2502                                              END_OF_LIST}
2503    };
2504    #define _PolicyDuplicationSelectDataAddress (&_PolicyDuplicationSelectData)
2505    #else
2506    #define _PolicyDuplicationSelectDataAddress 0
2507    #endif // CC_PolicyDuplicationSelect
2508    #if CC_PolicyAuthorize
2509    #include "PolicyAuthorize_fp.h"
2510    typedef TPM_RC  (PolicyAuthorize_Entry)(
2511        PolicyAuthorize_In          *in
2512    );
2513    typedef const struct {
2514        PolicyAuthorize_Entry   *entry;
2515        UINT16                  inSize;
```



```
2516        UINT16                  outSize;
2517        UINT16                  offsetOfTypes;
2518        UINT16                  paramOffsets[4];
2519        BYTE                    types[7];
2520    } PolicyAuthorize_COMMAND_DESCRIPTOR_t;
2521    PolicyAuthorize_COMMAND_DESCRIPTOR_t _PolicyAuthorizeData = {
2522        /* entry         */     &TPM2_PolicyAuthorize,
2523        /* inSize        */     (UINT16)(sizeof(PolicyAuthorize_In)),
2524        /* outSize       */     0,
2525        /* offsetOfTypes */     offsetof(PolicyAuthorize_COMMAND_DESCRIPTOR_t, types),
2526        /* offsets       */     {(UINT16)(offsetof(PolicyAuthorize_In, approvedPolicy)),
2527                                 (UINT16)(offsetof(PolicyAuthorize_In, policyRef)),
2528                                 (UINT16)(offsetof(PolicyAuthorize_In, keySign)),
2529                                 (UINT16)(offsetof(PolicyAuthorize_In, checkTicket))},
2530        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2531                                 TPM2B_DIGEST_P_UNMARSHAL,
2532                                 TPM2B_NONCE_P_UNMARSHAL,
2533                                 TPM2B_NAME_P_UNMARSHAL,
2534                                 TPMT_TK_VERIFIED_P_UNMARSHAL,
2535                                 END_OF_LIST,
2536                                 END_OF_LIST}
2537    };
2538    #define _PolicyAuthorizeDataAddress (&_PolicyAuthorizeData)
2539    #else
2540    #define _PolicyAuthorizeDataAddress 0
2541    #endif // CC_PolicyAuthorize
2542    #if CC_PolicyAuthValue
2543    #include "PolicyAuthValue_fp.h"
2544    typedef TPM_RC  (PolicyAuthValue_Entry)(
2545        PolicyAuthValue_In          *in
2546    );
2547    typedef const struct {
2548        PolicyAuthValue_Entry   *entry;
2549        UINT16                  inSize;
2550        UINT16                  outSize;
2551        UINT16                  offsetOfTypes;
2552        BYTE                    types[3];
2553    } PolicyAuthValue_COMMAND_DESCRIPTOR_t;
2554    PolicyAuthValue_COMMAND_DESCRIPTOR_t _PolicyAuthValueData = {
2555        /* entry         */     &TPM2_PolicyAuthValue,
2556        /* inSize        */     (UINT16)(sizeof(PolicyAuthValue_In)),
2557        /* outSize       */     0,
2558        /* offsetOfTypes */     offsetof(PolicyAuthValue_COMMAND_DESCRIPTOR_t, types),
2559        /* offsets       */     // No parameter offsets;
2560        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2561                                 END_OF_LIST,
2562                                 END_OF_LIST}
2563    };
2564    #define _PolicyAuthValueDataAddress (&_PolicyAuthValueData)
2565    #else
2566    #define _PolicyAuthValueDataAddress 0
2567    #endif // CC_PolicyAuthValue
2568    #if CC_PolicyPassword
2569    #include "PolicyPassword_fp.h"
2570    typedef TPM_RC  (PolicyPassword_Entry)(
2571        PolicyPassword_In           *in
2572    );
2573    typedef const struct {
2574        PolicyPassword_Entry    *entry;
2575        UINT16                  inSize;
2576        UINT16                  outSize;
2577        UINT16                  offsetOfTypes;
2578        BYTE                    types[3];
2579    } PolicyPassword_COMMAND_DESCRIPTOR_t;
2580    PolicyPassword_COMMAND_DESCRIPTOR_t _PolicyPasswordData = {
2581        /* entry         */     &TPM2_PolicyPassword,
```



```
2582        /* inSize        */     (UINT16)(sizeof(PolicyPassword_In)),
2583        /* outSize       */     0,
2584        /* offsetOfTypes */     offsetof(PolicyPassword_COMMAND_DESCRIPTOR_t, types),
2585        /* offsets       */     // No parameter offsets;
2586        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2587                                 END_OF_LIST,
2588                                 END_OF_LIST}
2589    };
2590    #define _PolicyPasswordDataAddress (&_PolicyPasswordData)
2591    #else
2592    #define _PolicyPasswordDataAddress 0
2593    #endif // CC_PolicyPassword
2594    #if CC_PolicyGetDigest
2595    #include "PolicyGetDigest_fp.h"
2596    typedef TPM_RC  (PolicyGetDigest_Entry)(
2597        PolicyGetDigest_In          *in,
2598        PolicyGetDigest_Out         *out
2599    );
2600    typedef const struct {
2601        PolicyGetDigest_Entry   *entry;
2602        UINT16                  inSize;
2603        UINT16                  outSize;
2604        UINT16                  offsetOfTypes;
2605        BYTE                    types[4];
2606    } PolicyGetDigest_COMMAND_DESCRIPTOR_t;
2607    PolicyGetDigest_COMMAND_DESCRIPTOR_t _PolicyGetDigestData = {
2608        /* entry         */     &TPM2_PolicyGetDigest,
2609        /* inSize        */     (UINT16)(sizeof(PolicyGetDigest_In)),
2610        /* outSize       */     (UINT16)(sizeof(PolicyGetDigest_Out)),
2611        /* offsetOfTypes */     offsetof(PolicyGetDigest_COMMAND_DESCRIPTOR_t, types),
2612        /* offsets       */     // No parameter offsets;
2613        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2614                                 END_OF_LIST,
2615                                 TPM2B_DIGEST_P_MARSHAL,
2616                                 END_OF_LIST}
2617    };
2618    #define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
2619    #else
2620    #define _PolicyGetDigestDataAddress 0
2621    #endif // CC_PolicyGetDigest
2622    #if CC_PolicyNvWritten
2623    #include "PolicyNvWritten_fp.h"
2624    typedef TPM_RC  (PolicyNvWritten_Entry)(
2625        PolicyNvWritten_In          *in
2626    );
2627    typedef const struct {
2628        PolicyNvWritten_Entry   *entry;
2629        UINT16                  inSize;
2630        UINT16                  outSize;
2631        UINT16                  offsetOfTypes;
2632        UINT16                  paramOffsets[1];
2633        BYTE                    types[4];
2634    } PolicyNvWritten_COMMAND_DESCRIPTOR_t;
2635    PolicyNvWritten_COMMAND_DESCRIPTOR_t _PolicyNvWrittenData = {
2636        /* entry         */     &TPM2_PolicyNvWritten,
2637        /* inSize        */     (UINT16)(sizeof(PolicyNvWritten_In)),
2638        /* outSize       */     0,
2639        /* offsetOfTypes */     offsetof(PolicyNvWritten_COMMAND_DESCRIPTOR_t, types),
2640        /* offsets       */     {(UINT16)(offsetof(PolicyNvWritten_In, writtenSet))},
2641        /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2642                                 TPMI_YES_NO_P_UNMARSHAL,
2643                                 END_OF_LIST,
2644                                 END_OF_LIST}
2645    };
2646    #define _PolicyNvWrittenDataAddress (&_PolicyNvWrittenData)
2647    #else
```



```
2648  #define _PolicyNvWrittenDataAddress 0
2649  #endif // CC_PolicyNvWritten
2650  #if CC_PolicyTemplate
2651  #include "PolicyTemplate_fp.h"
2652  typedef TPM_RC  (PolicyTemplate_Entry)(
2653      PolicyTemplate_In           *in
2654  );
2655  typedef const struct {
2656      PolicyTemplate_Entry    *entry;
2657      UINT16                  inSize;
2658      UINT16                  outSize;
2659      UINT16                  offsetOfTypes;
2660      UINT16                  paramOffsets[1];
2661      BYTE                    types[4];
2662  } PolicyTemplate_COMMAND_DESCRIPTOR_t;
2663  PolicyTemplate_COMMAND_DESCRIPTOR_t _PolicyTemplateData = {
2664      /* entry         */     &TPM2_PolicyTemplate,
2665      /* inSize        */     (UINT16)(sizeof(PolicyTemplate_In)),
2666      /* outSize       */     0,
2667      /* offsetOfTypes */     offsetof(PolicyTemplate_COMMAND_DESCRIPTOR_t, types),
2668      /* offsets       */     {(UINT16)(offsetof(PolicyTemplate_In, templateHash))},
2669      /* types         */     {TPMI_SH_POLICY_H_UNMARSHAL,
2670                               TPM2B_DIGEST_P_UNMARSHAL,
2671                               END_OF_LIST,
2672                               END_OF_LIST}
2673  };
2674  #define _PolicyTemplateDataAddress (&_PolicyTemplateData)
2675  #else
2676  #define _PolicyTemplateDataAddress 0
2677  #endif // CC_PolicyTemplate
2678  #if CC_PolicyAuthorizeNV
2679  #include "PolicyAuthorizeNV_fp.h"
2680  typedef TPM_RC  (PolicyAuthorizeNV_Entry)(
2681      PolicyAuthorizeNV_In            *in
2682  );
2683  typedef const struct {
2684      PolicyAuthorizeNV_Entry     *entry;
2685      UINT16                      inSize;
2686      UINT16                      outSize;
2687      UINT16                      offsetOfTypes;
2688      UINT16                      paramOffsets[2];
2689      BYTE                        types[5];
2690  } PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t;
2691  PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t _PolicyAuthorizeNVData = {
2692      /* entry         */         &TPM2_PolicyAuthorizeNV,
2693      /* inSize        */         (UINT16)(sizeof(PolicyAuthorizeNV_In)),
2694      /* outSize       */         0,
2695      /* offsetOfTypes */         offsetof(PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t,
      types),
2696      /* offsets       */         {(UINT16)(offsetof(PolicyAuthorizeNV_In, nvIndex)),
2697                                   (UINT16)(offsetof(PolicyAuthorizeNV_In,
      policySession))},
2698      /* types         */         {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2699                                   TPMI_RH_NV_INDEX_H_UNMARSHAL,
2700                                   TPMI_SH_POLICY_H_UNMARSHAL,
2701                                   END_OF_LIST,
2702                                   END_OF_LIST}
2703  };
2704  #define _PolicyAuthorizeNVDataAddress (&_PolicyAuthorizeNVData)
2705  #else
2706  #define _PolicyAuthorizeNVDataAddress 0
2707  #endif // CC_PolicyAuthorizeNV
2708  #if CC_CreatePrimary
2709  #include "CreatePrimary_fp.h"
2710  typedef TPM_RC  (CreatePrimary_Entry)(
2711      CreatePrimary_In            *in,
```

```
2712        CreatePrimary_Out           *out
2713    );
2714    typedef const struct {
2715        CreatePrimary_Entry     *entry;
2716        UINT16                  inSize;
2717        UINT16                  outSize;
2718        UINT16                  offsetOfTypes;
2719        UINT16                  paramOffsets[9];
2720        BYTE                    types[13];
2721    } CreatePrimary_COMMAND_DESCRIPTOR_t;
2722    CreatePrimary_COMMAND_DESCRIPTOR_t _CreatePrimaryData = {
2723        /* entry         */     &TPM2_CreatePrimary,
2724        /* inSize        */     (UINT16)(sizeof(CreatePrimary_In)),
2725        /* outSize       */     (UINT16)(sizeof(CreatePrimary_Out)),
2726        /* offsetOfTypes */     offsetof(CreatePrimary_COMMAND_DESCRIPTOR_t, types),
2727        /* offsets       */     {(UINT16)(offsetof(CreatePrimary_In, inSensitive)),
2728                                 (UINT16)(offsetof(CreatePrimary_In, inPublic)),
2729                                 (UINT16)(offsetof(CreatePrimary_In, outsideInfo)),
2730                                 (UINT16)(offsetof(CreatePrimary_In, creationPCR)),
2731                                 (UINT16)(offsetof(CreatePrimary_Out, outPublic)),
2732                                 (UINT16)(offsetof(CreatePrimary_Out, creationData)),
2733                                 (UINT16)(offsetof(CreatePrimary_Out, creationHash)),
2734                                 (UINT16)(offsetof(CreatePrimary_Out, creationTicket)),
2735                                 (UINT16)(offsetof(CreatePrimary_Out, name))},
2736        /* types         */     {TPMI_RH_HIERARCHY_H_UNMARSHAL + ADD_FLAG,
2737                                  TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
2738                                  TPM2B_PUBLIC_P_UNMARSHAL,
2739                                  TPM2B_DATA_P_UNMARSHAL,
2740                                  TPML_PCR_SELECTION_P_UNMARSHAL,
2741                                  END_OF_LIST,
2742                                  TPM_HANDLE_H_MARSHAL,
2743                                  TPM2B_PUBLIC_P_MARSHAL,
2744                                  TPM2B_CREATION_DATA_P_MARSHAL,
2745                                  TPM2B_DIGEST_P_MARSHAL,
2746                                  TPMT_TK_CREATION_P_MARSHAL,
2747                                  TPM2B_NAME_P_MARSHAL,
2748                                  END_OF_LIST}
2749    };
2750    #define _CreatePrimaryDataAddress (&_CreatePrimaryData)
2751    #else
2752    #define _CreatePrimaryDataAddress 0
2753    #endif // CC_CreatePrimary
2754    #if CC_HierarchyControl
2755    #include "HierarchyControl_fp.h"
2756    typedef TPM_RC  (HierarchyControl_Entry)(
2757        HierarchyControl_In         *in
2758    );
2759    typedef const struct {
2760        HierarchyControl_Entry  *entry;
2761        UINT16                  inSize;
2762        UINT16                  outSize;
2763        UINT16                  offsetOfTypes;
2764        UINT16                  paramOffsets[2];
2765        BYTE                    types[5];
2766    } HierarchyControl_COMMAND_DESCRIPTOR_t;
2767    HierarchyControl_COMMAND_DESCRIPTOR_t _HierarchyControlData = {
2768        /* entry         */     &TPM2_HierarchyControl,
2769        /* inSize        */     (UINT16)(sizeof(HierarchyControl_In)),
2770        /* outSize       */     0,
2771        /* offsetOfTypes */     offsetof(HierarchyControl_COMMAND_DESCRIPTOR_t, types),
2772        /* offsets       */     {(UINT16)(offsetof(HierarchyControl_In, enable)),
2773                                  (UINT16)(offsetof(HierarchyControl_In, state))},
2774        /* types         */     {TPMI_RH_HIERARCHY_H_UNMARSHAL,
2775                                  TPMI_RH_ENABLES_P_UNMARSHAL,
2776                                  TPMI_YES_NO_P_UNMARSHAL,
2777                                  END_OF_LIST,
```



```
2778                                END_OF_LIST}
2779    };
2780    #define _HierarchyControlDataAddress (&_HierarchyControlData)
2781    #else
2782    #define _HierarchyControlDataAddress 0
2783    #endif // CC_HierarchyControl
2784    #if CC_SetPrimaryPolicy
2785    #include "SetPrimaryPolicy_fp.h"
2786    typedef TPM_RC  (SetPrimaryPolicy_Entry)(
2787        SetPrimaryPolicy_In         *in
2788    );
2789    typedef const struct {
2790        SetPrimaryPolicy_Entry  *entry;
2791        UINT16                  inSize;
2792        UINT16                  outSize;
2793        UINT16                  offsetOfTypes;
2794        UINT16                  paramOffsets[2];
2795        BYTE                    types[5];
2796    } SetPrimaryPolicy_COMMAND_DESCRIPTOR_t;
2797    SetPrimaryPolicy_COMMAND_DESCRIPTOR_t _SetPrimaryPolicyData = {
2798        /* entry         */     &TPM2_SetPrimaryPolicy,
2799        /* inSize        */     (UINT16)(sizeof(SetPrimaryPolicy_In)),
2800        /* outSize       */     0,
2801        /* offsetOfTypes */     offsetof(SetPrimaryPolicy_COMMAND_DESCRIPTOR_t, types),
2802        /* offsets       */     {(UINT16)(offsetof(SetPrimaryPolicy_In, authPolicy)),
2803                                 (UINT16)(offsetof(SetPrimaryPolicy_In, hashAlg))},
2804        /* types         */     {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
2805                                 TPM2B_DIGEST_P_UNMARSHAL,
2806                                 TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2807                                 END_OF_LIST,
2808                                 END_OF_LIST}
2809    };
2810    #define _SetPrimaryPolicyDataAddress (&_SetPrimaryPolicyData)
2811    #else
2812    #define _SetPrimaryPolicyDataAddress 0
2813    #endif // CC_SetPrimaryPolicy
2814    #if CC_ChangePPS
2815    #include "ChangePPS_fp.h"
2816    typedef TPM_RC  (ChangePPS_Entry)(
2817        ChangePPS_In                *in
2818    );
2819    typedef const struct {
2820        ChangePPS_Entry         *entry;
2821        UINT16                  inSize;
2822        UINT16                  outSize;
2823        UINT16                  offsetOfTypes;
2824        BYTE                    types[3];
2825    } ChangePPS_COMMAND_DESCRIPTOR_t;
2826    ChangePPS_COMMAND_DESCRIPTOR_t _ChangePPSData = {
2827        /* entry         */     &TPM2_ChangePPS,
2828        /* inSize        */     (UINT16)(sizeof(ChangePPS_In)),
2829        /* outSize       */     0,
2830        /* offsetOfTypes */     offsetof(ChangePPS_COMMAND_DESCRIPTOR_t, types),
2831        /* offsets       */     // No parameter offsets;
2832        /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
2833                                 END_OF_LIST,
2834                                 END_OF_LIST}
2835    };
2836    #define _ChangePPSDataAddress (&_ChangePPSData)
2837    #else
2838    #define _ChangePPSDataAddress 0
2839    #endif // CC_ChangePPS
2840    #if CC_ChangeEPS
2841    #include "ChangeEPS_fp.h"
2842    typedef TPM_RC  (ChangeEPS_Entry)(
2843        ChangeEPS_In                *in
```

```
2844    );
2845    typedef const struct {
2846        ChangeEPS_Entry         *entry;
2847        UINT16                  inSize;
2848        UINT16                  outSize;
2849        UINT16                  offsetOfTypes;
2850        BYTE                    types[3];
2851    } ChangeEPS_COMMAND_DESCRIPTOR_t;
2852    ChangeEPS_COMMAND_DESCRIPTOR_t _ChangeEPSData = {
2853        /* entry         */     &TPM2_ChangeEPS,
2854        /* inSize        */     (UINT16)(sizeof(ChangeEPS_In)),
2855        /* outSize       */     0,
2856        /* offsetOfTypes */     offsetof(ChangeEPS_COMMAND_DESCRIPTOR_t, types),
2857        /* offsets       */     // No parameter offsets;
2858        /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
2859                                 END_OF_LIST,
2860                                 END_OF_LIST}
2861    };
2862    #define _ChangeEPSDataAddress (&_ChangeEPSData)
2863    #else
2864    #define _ChangeEPSDataAddress 0
2865    #endif // CC_ChangeEPS
2866    #if CC_Clear
2867    #include "Clear_fp.h"
2868    typedef TPM_RC  (Clear_Entry)(
2869        Clear_In                    *in
2870    );
2871    typedef const struct {
2872        Clear_Entry             *entry;
2873        UINT16                  inSize;
2874        UINT16                  outSize;
2875        UINT16                  offsetOfTypes;
2876        BYTE                    types[3];
2877    } Clear_COMMAND_DESCRIPTOR_t;
2878    Clear_COMMAND_DESCRIPTOR_t _ClearData = {
2879        /* entry         */     &TPM2_Clear,
2880        /* inSize        */     (UINT16)(sizeof(Clear_In)),
2881        /* outSize       */     0,
2882        /* offsetOfTypes */     offsetof(Clear_COMMAND_DESCRIPTOR_t, types),
2883        /* offsets       */     // No parameter offsets;
2884        /* types         */     {TPMI_RH_CLEAR_H_UNMARSHAL,
2885                                 END_OF_LIST,
2886                                 END_OF_LIST}
2887    };
2888    #define _ClearDataAddress (&_ClearData)
2889    #else
2890    #define _ClearDataAddress 0
2891    #endif // CC_Clear
2892    #if CC_ClearControl
2893    #include "ClearControl_fp.h"
2894    typedef TPM_RC  (ClearControl_Entry)(
2895        ClearControl_In             *in
2896    );
2897    typedef const struct {
2898        ClearControl_Entry      *entry;
2899        UINT16                  inSize;
2900        UINT16                  outSize;
2901        UINT16                  offsetOfTypes;
2902        UINT16                  paramOffsets[1];
2903        BYTE                    types[4];
2904    } ClearControl_COMMAND_DESCRIPTOR_t;
2905    ClearControl_COMMAND_DESCRIPTOR_t _ClearControlData = {
2906        /* entry         */     &TPM2_ClearControl,
2907        /* inSize        */     (UINT16)(sizeof(ClearControl_In)),
2908        /* outSize       */     0,
2909        /* offsetOfTypes */     offsetof(ClearControl_COMMAND_DESCRIPTOR_t, types),
```



```
2910        /* offsets       */     {(UINT16)(offsetof(ClearControl_In, disable))},
2911        /* types         */     {TPMI_RH_CLEAR_H_UNMARSHAL,
2912                                  TPMI_YES_NO_P_UNMARSHAL,
2913                                  END_OF_LIST,
2914                                  END_OF_LIST}
2915    };
2916    #define _ClearControlDataAddress (&_ClearControlData)
2917    #else
2918    #define _ClearControlDataAddress 0
2919    #endif // CC_ClearControl
2920    #if CC_HierarchyChangeAuth
2921    #include "HierarchyChangeAuth_fp.h"
2922    typedef TPM_RC  (HierarchyChangeAuth_Entry)(
2923        HierarchyChangeAuth_In          *in
2924    );
2925    typedef const struct {
2926        HierarchyChangeAuth_Entry   *entry;
2927        UINT16                      inSize;
2928        UINT16                      outSize;
2929        UINT16                      offsetOfTypes;
2930        UINT16                      paramOffsets[1];
2931        BYTE                        types[4];
2932    } HierarchyChangeAuth_COMMAND_DESCRIPTOR_t;
2933    HierarchyChangeAuth_COMMAND_DESCRIPTOR_t _HierarchyChangeAuthData = {
2934        /* entry         */         &TPM2_HierarchyChangeAuth,
2935        /* inSize        */         (UINT16)(sizeof(HierarchyChangeAuth_In)),
2936        /* outSize       */         0,
2937        /* offsetOfTypes */         offsetof(HierarchyChangeAuth_COMMAND_DESCRIPTOR_t,
        types),
2938        /* offsets       */         {(UINT16)(offsetof(HierarchyChangeAuth_In, newAuth))},
2939        /* types         */         {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
2940                                     TPM2B_AUTH_P_UNMARSHAL,
2941                                     END_OF_LIST,
2942                                     END_OF_LIST}
2943    };
2944    #define _HierarchyChangeAuthDataAddress (&_HierarchyChangeAuthData)
2945    #else
2946    #define _HierarchyChangeAuthDataAddress 0
2947    #endif // CC_HierarchyChangeAuth
2948    #if CC_DictionaryAttackLockReset
2949    #include "DictionaryAttackLockReset_fp.h"
2950    typedef TPM_RC  (DictionaryAttackLockReset_Entry)(
2951        DictionaryAttackLockReset_In            *in
2952    );
2953    typedef const struct {
2954        DictionaryAttackLockReset_Entry     *entry;
2955        UINT16                              inSize;
2956        UINT16                              outSize;
2957        UINT16                              offsetOfTypes;
2958        BYTE                                types[3];
2959    } DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t;
2960    DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t _DictionaryAttackLockResetData = {
2961        /* entry         */                 &TPM2_DictionaryAttackLockReset,
2962        /* inSize        */
        (UINT16)(sizeof(DictionaryAttackLockReset_In)),
2963        /* outSize       */                 0,
2964        /* offsetOfTypes */
        offsetof(DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t, types),
2965        /* offsets       */                 // No parameter offsets;
2966        /* types         */                 {TPMI_RH_LOCKOUT_H_UNMARSHAL,
2967                                              END_OF_LIST,
2968                                              END_OF_LIST}
2969    };
2970    #define _DictionaryAttackLockResetDataAddress (&_DictionaryAttackLockResetData)
2971    #else
2972    #define _DictionaryAttackLockResetDataAddress 0
```

```
2973  #endif // CC_DictionaryAttackLockReset
2974  #if CC_DictionaryAttackParameters
2975  #include "DictionaryAttackParameters_fp.h"
2976  typedef TPM_RC  (DictionaryAttackParameters_Entry)(
2977      DictionaryAttackParameters_In           *in
2978  );
2979  typedef const struct {
2980      DictionaryAttackParameters_Entry    *entry;
2981      UINT16                              inSize;
2982      UINT16                              outSize;
2983      UINT16                              offsetOfTypes;
2984      UINT16                              paramOffsets[3];
2985      BYTE                                types[6];
2986  } DictionaryAttackParameters_COMMAND_DESCRIPTOR_t;
2987  DictionaryAttackParameters_COMMAND_DESCRIPTOR_t _DictionaryAttackParametersData = {
2988      /* entry         */                 &TPM2_DictionaryAttackParameters,
2989      /* inSize        */
     (UINT16)(sizeof(DictionaryAttackParameters_In)),
2990      /* outSize       */                 0,
2991      /* offsetOfTypes */
     offsetof(DictionaryAttackParameters_COMMAND_DESCRIPTOR_t, types),
2992      /* offsets       */
     {(UINT16)(offsetof(DictionaryAttackParameters_In, newMaxTries)),
2993
     (UINT16)(offsetof(DictionaryAttackParameters_In, newRecoveryTime)),
2994
     (UINT16)(offsetof(DictionaryAttackParameters_In, lockoutRecovery))},
2995      /* types         */                 {TPMI_RH_LOCKOUT_H_UNMARSHAL,
2996                                           UINT32_P_UNMARSHAL,
2997                                           UINT32_P_UNMARSHAL,
2998                                           UINT32_P_UNMARSHAL,
2999                                           END_OF_LIST,
3000                                           END_OF_LIST}
3001  };
3002  #define _DictionaryAttackParametersDataAddress (&_DictionaryAttackParametersData)
3003  #else
3004  #define _DictionaryAttackParametersDataAddress 0
3005  #endif // CC_DictionaryAttackParameters
3006  #if CC_PP_Commands
3007  #include "PP_Commands_fp.h"
3008  typedef TPM_RC  (PP_Commands_Entry)(
3009      PP_Commands_In                  *in
3010  );
3011  typedef const struct {
3012      PP_Commands_Entry       *entry;
3013      UINT16                  inSize;
3014      UINT16                  outSize;
3015      UINT16                  offsetOfTypes;
3016      UINT16                  paramOffsets[2];
3017      BYTE                    types[5];
3018  } PP_Commands_COMMAND_DESCRIPTOR_t;
3019  PP_Commands_COMMAND_DESCRIPTOR_t _PP_CommandsData = {
3020      /* entry         */     &TPM2_PP_Commands,
3021      /* inSize        */     (UINT16)(sizeof(PP_Commands_In)),
3022      /* outSize       */     0,
3023      /* offsetOfTypes */     offsetof(PP_Commands_COMMAND_DESCRIPTOR_t, types),
3024      /* offsets       */     {(UINT16)(offsetof(PP_Commands_In, setList)),
3025                              (UINT16)(offsetof(PP_Commands_In, clearList))},
3026      /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
3027                               TPML_CC_P_UNMARSHAL,
3028                               TPML_CC_P_UNMARSHAL,
3029                               END_OF_LIST,
3030                               END_OF_LIST}
3031  };
3032  #define _PP_CommandsDataAddress (&_PP_CommandsData)
3033  #else
```



```
3034    #define _PP_CommandsDataAddress 0
3035    #endif // CC_PP_Commands
3036    #if CC_SetAlgorithmSet
3037    #include "SetAlgorithmSet_fp.h"
3038    typedef TPM_RC  (SetAlgorithmSet_Entry)(
3039        SetAlgorithmSet_In          *in
3040    );
3041    typedef const struct {
3042        SetAlgorithmSet_Entry   *entry;
3043        UINT16                  inSize;
3044        UINT16                  outSize;
3045        UINT16                  offsetOfTypes;
3046        UINT16                  paramOffsets[1];
3047        BYTE                    types[4];
3048    } SetAlgorithmSet_COMMAND_DESCRIPTOR_t;
3049    SetAlgorithmSet_COMMAND_DESCRIPTOR_t _SetAlgorithmSetData = {
3050        /* entry         */     &TPM2_SetAlgorithmSet,
3051        /* inSize        */     (UINT16)(sizeof(SetAlgorithmSet_In)),
3052        /* outSize       */     0,
3053        /* offsetOfTypes */     offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
3054        /* offsets       */     {(UINT16)(offsetof(SetAlgorithmSet_In, algorithmSet))},
3055        /* types         */     {TPMI_RH_PLATFORM_H_UNMARSHAL,
3056                                 UINT32_P_UNMARSHAL,
3057                                 END_OF_LIST,
3058                                 END_OF_LIST}
3059    };
3060    #define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
3061    #else
3062    #define _SetAlgorithmSetDataAddress 0
3063    #endif // CC_SetAlgorithmSet
3064    #if CC_FieldUpgradeStart
3065    #include "FieldUpgradeStart_fp.h"
3066    typedef TPM_RC  (FieldUpgradeStart_Entry)(
3067        FieldUpgradeStart_In            *in
3068    );
3069    typedef const struct {
3070        FieldUpgradeStart_Entry     *entry;
3071        UINT16                      inSize;
3072        UINT16                      outSize;
3073        UINT16                      offsetOfTypes;
3074        UINT16                      paramOffsets[3];
3075        BYTE                        types[6];
3076    } FieldUpgradeStart_COMMAND_DESCRIPTOR_t;
3077    FieldUpgradeStart_COMMAND_DESCRIPTOR_t _FieldUpgradeStartData = {
3078        /* entry         */         &TPM2_FieldUpgradeStart,
3079        /* inSize        */         (UINT16)(sizeof(FieldUpgradeStart_In)),
3080        /* outSize       */         0,
3081        /* offsetOfTypes */         offsetof(FieldUpgradeStart_COMMAND_DESCRIPTOR_t,
        types),
3082        /* offsets       */         {(UINT16)(offsetof(FieldUpgradeStart_In, keyHandle)),
3083                                     (UINT16)(offsetof(FieldUpgradeStart_In, fuDigest)),
3084                                     (UINT16)(offsetof(FieldUpgradeStart_In,
        manifestSignature))},
3085        /* types         */         {TPMI_RH_PLATFORM_H_UNMARSHAL,
3086                                     TPMI_DH_OBJECT_H_UNMARSHAL,
3087                                     TPM2B_DIGEST_P_UNMARSHAL,
3088                                     TPMT_SIGNATURE_P_UNMARSHAL,
3089                                     END_OF_LIST,
3090                                     END_OF_LIST}
3091    };
3092    #define _FieldUpgradeStartDataAddress (&_FieldUpgradeStartData)
3093    #else
3094    #define _FieldUpgradeStartDataAddress 0
3095    #endif // CC_FieldUpgradeStart
3096    #if CC_FieldUpgradeData
3097    #include "FieldUpgradeData_fp.h"
```

```
3098  typedef TPM_RC  (FieldUpgradeData_Entry)(
3099      FieldUpgradeData_In         *in,
3100      FieldUpgradeData_Out        *out
3101  );
3102  typedef const struct {
3103      FieldUpgradeData_Entry  *entry;
3104      UINT16                  inSize;
3105      UINT16                  outSize;
3106      UINT16                  offsetOfTypes;
3107      UINT16                  paramOffsets[1];
3108      BYTE                    types[5];
3109  } FieldUpgradeData_COMMAND_DESCRIPTOR_t;
3110  FieldUpgradeData_COMMAND_DESCRIPTOR_t _FieldUpgradeDataData = {
3111      /* entry         */     &TPM2_FieldUpgradeData,
3112      /* inSize        */     (UINT16)(sizeof(FieldUpgradeData_In)),
3113      /* outSize       */     (UINT16)(sizeof(FieldUpgradeData_Out)),
3114      /* offsetOfTypes */     offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t, types),
3115      /* offsets       */     {(UINT16)(offsetof(FieldUpgradeData_Out, firstDigest))},
3116      /* types         */     {TPM2B_MAX_BUFFER_P_UNMARSHAL,
3117                                END_OF_LIST,
3118                                TPMT_HA_P_MARSHAL,
3119                                TPMT_HA_P_MARSHAL,
3120                                END_OF_LIST}
3121  };
3122  #define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
3123  #else
3124  #define _FieldUpgradeDataDataAddress 0
3125  #endif // CC_FieldUpgradeData
3126  #if CC_FirmwareRead
3127  #include "FirmwareRead_fp.h"
3128  typedef TPM_RC  (FirmwareRead_Entry)(
3129      FirmwareRead_In             *in,
3130      FirmwareRead_Out            *out
3131  );
3132  typedef const struct {
3133      FirmwareRead_Entry      *entry;
3134      UINT16                  inSize;
3135      UINT16                  outSize;
3136      UINT16                  offsetOfTypes;
3137      BYTE                    types[4];
3138  } FirmwareRead_COMMAND_DESCRIPTOR_t;
3139  FirmwareRead_COMMAND_DESCRIPTOR_t _FirmwareReadData = {
3140      /* entry         */     &TPM2_FirmwareRead,
3141      /* inSize        */     (UINT16)(sizeof(FirmwareRead_In)),
3142      /* outSize       */     (UINT16)(sizeof(FirmwareRead_Out)),
3143      /* offsetOfTypes */     offsetof(FirmwareRead_COMMAND_DESCRIPTOR_t, types),
3144      /* offsets       */     // No parameter offsets;
3145      /* types         */     {UINT32_P_UNMARSHAL,
3146                                END_OF_LIST,
3147                                TPM2B_MAX_BUFFER_P_MARSHAL,
3148                                END_OF_LIST}
3149  };
3150  #define _FirmwareReadDataAddress (&_FirmwareReadData)
3151  #else
3152  #define _FirmwareReadDataAddress 0
3153  #endif // CC_FirmwareRead
3154  #if CC_ContextSave
3155  #include "ContextSave_fp.h"
3156  typedef TPM_RC  (ContextSave_Entry)(
3157      ContextSave_In              *in,
3158      ContextSave_Out             *out
3159  );
3160  typedef const struct {
3161      ContextSave_Entry       *entry;
3162      UINT16                  inSize;
3163      UINT16                  outSize;
```

```
3164        UINT16                  offsetOfTypes;
3165        BYTE                    types[4];
3166    } ContextSave_COMMAND_DESCRIPTOR_t;
3167    ContextSave_COMMAND_DESCRIPTOR_t _ContextSaveData = {
3168        /* entry        */      &TPM2_ContextSave,
3169        /* inSize       */      (UINT16)(sizeof(ContextSave_In)),
3170        /* outSize      */      (UINT16)(sizeof(ContextSave_Out)),
3171        /* offsetOfTypes */     offsetof(ContextSave_COMMAND_DESCRIPTOR_t, types),
3172        /* offsets      */      // No parameter offsets;
3173        /* types        */      {TPMI_DH_CONTEXT_H_UNMARSHAL,
3174                                 END_OF_LIST,
3175                                 TPMS_CONTEXT_P_MARSHAL,
3176                                 END_OF_LIST}
3177    };
3178    #define _ContextSaveDataAddress (&_ContextSaveData)
3179    #else
3180    #define _ContextSaveDataAddress 0
3181    #endif // CC_ContextSave
3182    #if CC_ContextLoad
3183    #include "ContextLoad_fp.h"
3184    typedef TPM_RC  (ContextLoad_Entry)(
3185        ContextLoad_In              *in,
3186        ContextLoad_Out             *out
3187    );
3188    typedef const struct {
3189        ContextLoad_Entry       *entry;
3190        UINT16                  inSize;
3191        UINT16                  outSize;
3192        UINT16                  offsetOfTypes;
3193        BYTE                    types[4];
3194    } ContextLoad_COMMAND_DESCRIPTOR_t;
3195    ContextLoad_COMMAND_DESCRIPTOR_t _ContextLoadData = {
3196        /* entry        */      &TPM2_ContextLoad,
3197        /* inSize       */      (UINT16)(sizeof(ContextLoad_In)),
3198        /* outSize      */      (UINT16)(sizeof(ContextLoad_Out)),
3199        /* offsetOfTypes */     offsetof(ContextLoad_COMMAND_DESCRIPTOR_t, types),
3200        /* offsets      */      // No parameter offsets;
3201        /* types        */      {TPMS_CONTEXT_P_UNMARSHAL,
3202                                 END_OF_LIST,
3203                                 TPMI_DH_CONTEXT_H_MARSHAL,
3204                                 END_OF_LIST}
3205    };
3206    #define _ContextLoadDataAddress (&_ContextLoadData)
3207    #else
3208    #define _ContextLoadDataAddress 0
3209    #endif // CC_ContextLoad
3210    #if CC_FlushContext
3211    #include "FlushContext_fp.h"
3212    typedef TPM_RC  (FlushContext_Entry)(
3213        FlushContext_In             *in
3214    );
3215    typedef const struct {
3216        FlushContext_Entry      *entry;
3217        UINT16                  inSize;
3218        UINT16                  outSize;
3219        UINT16                  offsetOfTypes;
3220        BYTE                    types[3];
3221    } FlushContext_COMMAND_DESCRIPTOR_t;
3222    FlushContext_COMMAND_DESCRIPTOR_t _FlushContextData = {
3223        /* entry        */      &TPM2_FlushContext,
3224        /* inSize       */      (UINT16)(sizeof(FlushContext_In)),
3225        /* outSize      */      0,
3226        /* offsetOfTypes */     offsetof(FlushContext_COMMAND_DESCRIPTOR_t, types),
3227        /* offsets      */      // No parameter offsets;
3228        /* types        */      {TPMI_DH_CONTEXT_P_UNMARSHAL,
3229                                 END_OF_LIST,
```

```
3230                                END_OF_LIST}
3231    };
3232    #define _FlushContextDataAddress (&_FlushContextData)
3233    #else
3234    #define _FlushContextDataAddress 0
3235    #endif // CC_FlushContext
3236    #if CC_EvictControl
3237    #include "EvictControl_fp.h"
3238    typedef TPM_RC  (EvictControl_Entry)(
3239        EvictControl_In             *in
3240    );
3241    typedef const struct {
3242        EvictControl_Entry      *entry;
3243        UINT16                  inSize;
3244        UINT16                  outSize;
3245        UINT16                  offsetOfTypes;
3246        UINT16                  paramOffsets[2];
3247        BYTE                    types[5];
3248    } EvictControl_COMMAND_DESCRIPTOR_t;
3249    EvictControl_COMMAND_DESCRIPTOR_t _EvictControlData = {
3250        /* entry         */     &TPM2_EvictControl,
3251        /* inSize        */     (UINT16)(sizeof(EvictControl_In)),
3252        /* outSize       */     0,
3253        /* offsetOfTypes */     offsetof(EvictControl_COMMAND_DESCRIPTOR_t, types),
3254        /* offsets       */     {(UINT16)(offsetof(EvictControl_In, objectHandle)),
3255                                 (UINT16)(offsetof(EvictControl_In, persistentHandle))},
3256        /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3257                                 TPMI_DH_OBJECT_H_UNMARSHAL,
3258                                 TPMI_DH_PERSISTENT_P_UNMARSHAL,
3259                                 END_OF_LIST,
3260                                 END_OF_LIST}
3261    };
3262    #define _EvictControlDataAddress (&_EvictControlData)
3263    #else
3264    #define _EvictControlDataAddress 0
3265    #endif // CC_EvictControl
3266    #if CC_ReadClock
3267    #include "ReadClock_fp.h"
3268    typedef TPM_RC  (ReadClock_Entry)(
3269        ReadClock_Out               *out
3270    );
3271    typedef const struct {
3272        ReadClock_Entry         *entry;
3273        UINT16                  inSize;
3274        UINT16                  outSize;
3275        UINT16                  offsetOfTypes;
3276        BYTE                    types[3];
3277    } ReadClock_COMMAND_DESCRIPTOR_t;
3278    ReadClock_COMMAND_DESCRIPTOR_t _ReadClockData = {
3279        /* entry         */     &TPM2_ReadClock,
3280        /* inSize        */     0,
3281        /* outSize       */     (UINT16)(sizeof(ReadClock_Out)),
3282        /* offsetOfTypes */     offsetof(ReadClock_COMMAND_DESCRIPTOR_t, types),
3283        /* offsets       */     // No parameter offsets;
3284        /* types         */     {END_OF_LIST,
3285                                 TPMS_TIME_INFO_P_MARSHAL,
3286                                 END_OF_LIST}
3287    };
3288    #define _ReadClockDataAddress (&_ReadClockData)
3289    #else
3290    #define _ReadClockDataAddress 0
3291    #endif // CC_ReadClock
3292    #if CC_ClockSet
3293    #include "ClockSet_fp.h"
3294    typedef TPM_RC  (ClockSet_Entry)(
3295        ClockSet_In                 *in
```



```
3296    );
3297    typedef const struct {
3298        ClockSet_Entry          *entry;
3299        UINT16                  inSize;
3300        UINT16                  outSize;
3301        UINT16                  offsetOfTypes;
3302        UINT16                  paramOffsets[1];
3303        BYTE                    types[4];
3304    } ClockSet_COMMAND_DESCRIPTOR_t;
3305    ClockSet_COMMAND_DESCRIPTOR_t _ClockSetData = {
3306        /* entry         */     &TPM2_ClockSet,
3307        /* inSize        */     (UINT16)(sizeof(ClockSet_In)),
3308        /* outSize       */     0,
3309        /* offsetOfTypes */     offsetof(ClockSet_COMMAND_DESCRIPTOR_t, types),
3310        /* offsets       */     {(UINT16)(offsetof(ClockSet_In, newTime))},
3311        /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3312                                 UINT64_P_UNMARSHAL,
3313                                 END_OF_LIST,
3314                                 END_OF_LIST}
3315    };
3316    #define _ClockSetDataAddress (&_ClockSetData)
3317    #else
3318    #define _ClockSetDataAddress 0
3319    #endif // CC_ClockSet
3320    #if CC_ClockRateAdjust
3321    #include "ClockRateAdjust_fp.h"
3322    typedef TPM_RC  (ClockRateAdjust_Entry)(
3323        ClockRateAdjust_In          *in
3324    );
3325    typedef const struct {
3326        ClockRateAdjust_Entry   *entry;
3327        UINT16                  inSize;
3328        UINT16                  outSize;
3329        UINT16                  offsetOfTypes;
3330        UINT16                  paramOffsets[1];
3331        BYTE                    types[4];
3332    } ClockRateAdjust_COMMAND_DESCRIPTOR_t;
3333    ClockRateAdjust_COMMAND_DESCRIPTOR_t _ClockRateAdjustData = {
3334        /* entry         */     &TPM2_ClockRateAdjust,
3335        /* inSize        */     (UINT16)(sizeof(ClockRateAdjust_In)),
3336        /* outSize       */     0,
3337        /* offsetOfTypes */     offsetof(ClockRateAdjust_COMMAND_DESCRIPTOR_t, types),
3338        /* offsets       */     {(UINT16)(offsetof(ClockRateAdjust_In, rateAdjust))},
3339        /* types         */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3340                                 TPM_CLOCK_ADJUST_P_UNMARSHAL,
3341                                 END_OF_LIST,
3342                                 END_OF_LIST}
3343    };
3344    #define _ClockRateAdjustDataAddress (&_ClockRateAdjustData)
3345    #else
3346    #define _ClockRateAdjustDataAddress 0
3347    #endif // CC_ClockRateAdjust
3348    #if CC_GetCapability
3349    #include "GetCapability_fp.h"
3350    typedef TPM_RC  (GetCapability_Entry)(
3351        GetCapability_In            *in,
3352        GetCapability_Out           *out
3353    );
3354    typedef const struct {
3355        GetCapability_Entry     *entry;
3356        UINT16                  inSize;
3357        UINT16                  outSize;
3358        UINT16                  offsetOfTypes;
3359        UINT16                  paramOffsets[3];
3360        BYTE                    types[7];
3361    } GetCapability_COMMAND_DESCRIPTOR_t;
```

```
3362  GetCapability_COMMAND_DESCRIPTOR_t _GetCapabilityData = {
3363      /* entry         */     &TPM2_GetCapability,
3364      /* inSize        */     (UINT16)(sizeof(GetCapability_In)),
3365      /* outSize       */     (UINT16)(sizeof(GetCapability_Out)),
3366      /* offsetOfTypes */     offsetof(GetCapability_COMMAND_DESCRIPTOR_t, types),
3367      /* offsets       */     {(UINT16)(offsetof(GetCapability_In, property)),
3368                                (UINT16)(offsetof(GetCapability_In, propertyCount)),
3369                                (UINT16)(offsetof(GetCapability_Out, capabilityData))},
3370      /* types         */     {TPM_CAP_P_UNMARSHAL,
3371                                UINT32_P_UNMARSHAL,
3372                                UINT32_P_UNMARSHAL,
3373                                END_OF_LIST,
3374                                TPMI_YES_NO_P_MARSHAL,
3375                                TPMS_CAPABILITY_DATA_P_MARSHAL,
3376                                END_OF_LIST}
3377  };
3378  #define _GetCapabilityDataAddress (&_GetCapabilityData)
3379  #else
3380  #define _GetCapabilityDataAddress 0
3381  #endif // CC_GetCapability
3382  #if CC_TestParms
3383  #include "TestParms_fp.h"
3384  typedef TPM_RC  (TestParms_Entry)(
3385      TestParms_In                *in
3386  );
3387  typedef const struct {
3388      TestParms_Entry         *entry;
3389      UINT16                  inSize;
3390      UINT16                  outSize;
3391      UINT16                  offsetOfTypes;
3392      BYTE                    types[3];
3393  } TestParms_COMMAND_DESCRIPTOR_t;
3394  TestParms_COMMAND_DESCRIPTOR_t _TestParmsData = {
3395      /* entry         */     &TPM2_TestParms,
3396      /* inSize        */     (UINT16)(sizeof(TestParms_In)),
3397      /* outSize       */     0,
3398      /* offsetOfTypes */     offsetof(TestParms_COMMAND_DESCRIPTOR_t, types),
3399      /* offsets       */     // No parameter offsets;
3400      /* types         */     {TPMT_PUBLIC_PARMS_P_UNMARSHAL,
3401                                END_OF_LIST,
3402                                END_OF_LIST}
3403  };
3404  #define _TestParmsDataAddress (&_TestParmsData)
3405  #else
3406  #define _TestParmsDataAddress 0
3407  #endif // CC_TestParms
3408  #if CC_NV_DefineSpace
3409  #include "NV_DefineSpace_fp.h"
3410  typedef TPM_RC  (NV_DefineSpace_Entry)(
3411      NV_DefineSpace_In           *in
3412  );
3413  typedef const struct {
3414      NV_DefineSpace_Entry    *entry;
3415      UINT16                  inSize;
3416      UINT16                  outSize;
3417      UINT16                  offsetOfTypes;
3418      UINT16                  paramOffsets[2];
3419      BYTE                    types[5];
3420  } NV_DefineSpace_COMMAND_DESCRIPTOR_t;
3421  NV_DefineSpace_COMMAND_DESCRIPTOR_t _NV_DefineSpaceData = {
3422      /* entry         */     &TPM2_NV_DefineSpace,
3423      /* inSize        */     (UINT16)(sizeof(NV_DefineSpace_In)),
3424      /* outSize       */     0,
3425      /* offsetOfTypes */     offsetof(NV_DefineSpace_COMMAND_DESCRIPTOR_t, types),
3426      /* offsets       */     {(UINT16)(offsetof(NV_DefineSpace_In, auth)),
3427                                (UINT16)(offsetof(NV_DefineSpace_In, publicInfo))},
```



```
3428        /* types          */      {TPMI_RH_PROVISION_H_UNMARSHAL,
3429                                    TPM2B_AUTH_P_UNMARSHAL,
3430                                    TPM2B_NV_PUBLIC_P_UNMARSHAL,
3431                                    END_OF_LIST,
3432                                    END_OF_LIST}
3433    };
3434    #define _NV_DefineSpaceDataAddress (&_NV_DefineSpaceData)
3435    #else
3436    #define _NV_DefineSpaceDataAddress 0
3437    #endif // CC_NV_DefineSpace
3438    #if CC_NV_UndefineSpace
3439    #include "NV_UndefineSpace_fp.h"
3440    typedef TPM_RC  (NV_UndefineSpace_Entry)(
3441        NV_UndefineSpace_In         *in
3442    );
3443    typedef const struct {
3444        NV_UndefineSpace_Entry  *entry;
3445        UINT16                  inSize;
3446        UINT16                  outSize;
3447        UINT16                  offsetOfTypes;
3448        UINT16                  paramOffsets[1];
3449        BYTE                    types[4];
3450    } NV_UndefineSpace_COMMAND_DESCRIPTOR_t;
3451    NV_UndefineSpace_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceData = {
3452        /* entry          */     &TPM2_NV_UndefineSpace,
3453        /* inSize         */     (UINT16)(sizeof(NV_UndefineSpace_In)),
3454        /* outSize        */     0,
3455        /* offsetOfTypes */      offsetof(NV_UndefineSpace_COMMAND_DESCRIPTOR_t, types),
3456        /* offsets        */     {(UINT16)(offsetof(NV_UndefineSpace_In, nvIndex))},
3457        /* types          */     {TPMI_RH_PROVISION_H_UNMARSHAL,
3458                                    TPMI_RH_NV_INDEX_H_UNMARSHAL,
3459                                    END_OF_LIST,
3460                                    END_OF_LIST}
3461    };
3462    #define _NV_UndefineSpaceDataAddress (&_NV_UndefineSpaceData)
3463    #else
3464    #define _NV_UndefineSpaceDataAddress 0
3465    #endif // CC_NV_UndefineSpace
3466    #if CC_NV_UndefineSpaceSpecial
3467    #include "NV_UndefineSpaceSpecial_fp.h"
3468    typedef TPM_RC  (NV_UndefineSpaceSpecial_Entry)(
3469        NV_UndefineSpaceSpecial_In          *in
3470    );
3471    typedef const struct {
3472        NV_UndefineSpaceSpecial_Entry   *entry;
3473        UINT16                          inSize;
3474        UINT16                          outSize;
3475        UINT16                          offsetOfTypes;
3476        UINT16                          paramOffsets[1];
3477        BYTE                            types[4];
3478    } NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t;
3479    NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceSpecialData = {
3480        /* entry          */             &TPM2_NV_UndefineSpaceSpecial,
3481        /* inSize         */             (UINT16)(sizeof(NV_UndefineSpaceSpecial_In)),
3482        /* outSize        */             0,
3483        /* offsetOfTypes */
        offsetof(NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t, types),
3484        /* offsets        */             {(UINT16)(offsetof(NV_UndefineSpaceSpecial_In,
        platform))},
3485        /* types          */             {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3486                                           TPMI_RH_PLATFORM_H_UNMARSHAL,
3487                                           END_OF_LIST,
3488                                           END_OF_LIST}
3489    };
3490    #define _NV_UndefineSpaceSpecialDataAddress (&_NV_UndefineSpaceSpecialData)
3491    #else
```



```
3492  #define _NV_UndefineSpaceSpecialDataAddress 0
3493  #endif // CC_NV_UndefineSpaceSpecial
3494  #if CC_NV_ReadPublic
3495  #include "NV_ReadPublic_fp.h"
3496  typedef TPM_RC  (NV_ReadPublic_Entry)(
3497      NV_ReadPublic_In            *in,
3498      NV_ReadPublic_Out           *out
3499  );
3500  typedef const struct {
3501      NV_ReadPublic_Entry     *entry;
3502      UINT16                  inSize;
3503      UINT16                  outSize;
3504      UINT16                  offsetOfTypes;
3505      UINT16                  paramOffsets[1];
3506      BYTE                    types[5];
3507  } NV_ReadPublic_COMMAND_DESCRIPTOR_t;
3508  NV_ReadPublic_COMMAND_DESCRIPTOR_t _NV_ReadPublicData = {
3509      /* entry         */     &TPM2_NV_ReadPublic,
3510      /* inSize        */     (UINT16)(sizeof(NV_ReadPublic_In)),
3511      /* outSize       */     (UINT16)(sizeof(NV_ReadPublic_Out)),
3512      /* offsetOfTypes */     offsetof(NV_ReadPublic_COMMAND_DESCRIPTOR_t, types),
3513      /* offsets       */     {(UINT16)(offsetof(NV_ReadPublic_Out, nvName))},
3514      /* types         */     {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3515                               END_OF_LIST,
3516                               TPM2B_NV_PUBLIC_P_MARSHAL,
3517                               TPM2B_NAME_P_MARSHAL,
3518                               END_OF_LIST}
3519  };
3520  #define _NV_ReadPublicDataAddress (&_NV_ReadPublicData)
3521  #else
3522  #define _NV_ReadPublicDataAddress 0
3523  #endif // CC_NV_ReadPublic
3524  #if CC_NV_Write
3525  #include "NV_Write_fp.h"
3526  typedef TPM_RC  (NV_Write_Entry)(
3527      NV_Write_In                 *in
3528  );
3529  typedef const struct {
3530      NV_Write_Entry          *entry;
3531      UINT16                  inSize;
3532      UINT16                  outSize;
3533      UINT16                  offsetOfTypes;
3534      UINT16                  paramOffsets[3];
3535      BYTE                    types[6];
3536  } NV_Write_COMMAND_DESCRIPTOR_t;
3537  NV_Write_COMMAND_DESCRIPTOR_t _NV_WriteData = {
3538      /* entry         */     &TPM2_NV_Write,
3539      /* inSize        */     (UINT16)(sizeof(NV_Write_In)),
3540      /* outSize       */     0,
3541      /* offsetOfTypes */     offsetof(NV_Write_COMMAND_DESCRIPTOR_t, types),
3542      /* offsets       */     {(UINT16)(offsetof(NV_Write_In, nvIndex)),
3543                               (UINT16)(offsetof(NV_Write_In, data)),
3544                               (UINT16)(offsetof(NV_Write_In, offset))},
3545      /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3546                               TPMI_RH_NV_INDEX_H_UNMARSHAL,
3547                               TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3548                               UINT16_P_UNMARSHAL,
3549                               END_OF_LIST,
3550                               END_OF_LIST}
3551  };
3552  #define _NV_WriteDataAddress (&_NV_WriteData)
3553  #else
3554  #define _NV_WriteDataAddress 0
3555  #endif // CC_NV_Write
3556  #if CC_NV_Increment
3557  #include "NV_Increment_fp.h"
```



```
3558  typedef TPM_RC  (NV_Increment_Entry)(
3559      NV_Increment_In             *in
3560  );
3561  typedef const struct {
3562      NV_Increment_Entry      *entry;
3563      UINT16                  inSize;
3564      UINT16                  outSize;
3565      UINT16                  offsetOfTypes;
3566      UINT16                  paramOffsets[1];
3567      BYTE                    types[4];
3568  } NV_Increment_COMMAND_DESCRIPTOR_t;
3569  NV_Increment_COMMAND_DESCRIPTOR_t _NV_IncrementData = {
3570      /* entry         */     &TPM2_NV_Increment,
3571      /* inSize        */     (UINT16)(sizeof(NV_Increment_In)),
3572      /* outSize       */     0,
3573      /* offsetOfTypes */     offsetof(NV_Increment_COMMAND_DESCRIPTOR_t, types),
3574      /* offsets       */     {(UINT16)(offsetof(NV_Increment_In, nvIndex))},
3575      /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3576                                TPMI_RH_NV_INDEX_H_UNMARSHAL,
3577                                END_OF_LIST,
3578                                END_OF_LIST}
3579  };
3580  #define _NV_IncrementDataAddress (&_NV_IncrementData)
3581  #else
3582  #define _NV_IncrementDataAddress 0
3583  #endif // CC_NV_Increment
3584  #if CC_NV_Extend
3585  #include "NV_Extend_fp.h"
3586  typedef TPM_RC  (NV_Extend_Entry)(
3587      NV_Extend_In                *in
3588  );
3589  typedef const struct {
3590      NV_Extend_Entry         *entry;
3591      UINT16                  inSize;
3592      UINT16                  outSize;
3593      UINT16                  offsetOfTypes;
3594      UINT16                  paramOffsets[2];
3595      BYTE                    types[5];
3596  } NV_Extend_COMMAND_DESCRIPTOR_t;
3597  NV_Extend_COMMAND_DESCRIPTOR_t _NV_ExtendData = {
3598      /* entry         */     &TPM2_NV_Extend,
3599      /* inSize        */     (UINT16)(sizeof(NV_Extend_In)),
3600      /* outSize       */     0,
3601      /* offsetOfTypes */     offsetof(NV_Extend_COMMAND_DESCRIPTOR_t, types),
3602      /* offsets       */     {(UINT16)(offsetof(NV_Extend_In, nvIndex)),
3603                                (UINT16)(offsetof(NV_Extend_In, data))},
3604      /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3605                                TPMI_RH_NV_INDEX_H_UNMARSHAL,
3606                                TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3607                                END_OF_LIST,
3608                                END_OF_LIST}
3609  };
3610  #define _NV_ExtendDataAddress (&_NV_ExtendData)
3611  #else
3612  #define _NV_ExtendDataAddress 0
3613  #endif // CC_NV_Extend
3614  #if CC_NV_SetBits
3615  #include "NV_SetBits_fp.h"
3616  typedef TPM_RC  (NV_SetBits_Entry)(
3617      NV_SetBits_In               *in
3618  );
3619  typedef const struct {
3620      NV_SetBits_Entry        *entry;
3621      UINT16                  inSize;
3622      UINT16                  outSize;
3623      UINT16                  offsetOfTypes;
```

```
3624        UINT16                  paramOffsets[2];
3625        BYTE                    types[5];
3626    } NV_SetBits_COMMAND_DESCRIPTOR_t;
3627    NV_SetBits_COMMAND_DESCRIPTOR_t _NV_SetBitsData = {
3628        /* entry         */     &TPM2_NV_SetBits,
3629        /* inSize        */     (UINT16)(sizeof(NV_SetBits_In)),
3630        /* outSize       */     0,
3631        /* offsetOfTypes */     offsetof(NV_SetBits_COMMAND_DESCRIPTOR_t, types),
3632        /* offsets       */     {(UINT16)(offsetof(NV_SetBits_In, nvIndex)),
3633                                 (UINT16)(offsetof(NV_SetBits_In, bits))},
3634        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3635                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3636                                 UINT64_P_UNMARSHAL,
3637                                 END_OF_LIST,
3638                                 END_OF_LIST}
3639    };
3640    #define _NV_SetBitsDataAddress (&_NV_SetBitsData)
3641    #else
3642    #define _NV_SetBitsDataAddress 0
3643    #endif // CC_NV_SetBits
3644    #if CC_NV_WriteLock
3645    #include "NV_WriteLock_fp.h"
3646    typedef TPM_RC  (NV_WriteLock_Entry)(
3647        NV_WriteLock_In             *in
3648    );
3649    typedef const struct {
3650        NV_WriteLock_Entry      *entry;
3651        UINT16                  inSize;
3652        UINT16                  outSize;
3653        UINT16                  offsetOfTypes;
3654        UINT16                  paramOffsets[1];
3655        BYTE                    types[4];
3656    } NV_WriteLock_COMMAND_DESCRIPTOR_t;
3657    NV_WriteLock_COMMAND_DESCRIPTOR_t _NV_WriteLockData = {
3658        /* entry         */     &TPM2_NV_WriteLock,
3659        /* inSize        */     (UINT16)(sizeof(NV_WriteLock_In)),
3660        /* outSize       */     0,
3661        /* offsetOfTypes */     offsetof(NV_WriteLock_COMMAND_DESCRIPTOR_t, types),
3662        /* offsets       */     {(UINT16)(offsetof(NV_WriteLock_In, nvIndex))},
3663        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3664                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3665                                 END_OF_LIST,
3666                                 END_OF_LIST}
3667    };
3668    #define _NV_WriteLockDataAddress (&_NV_WriteLockData)
3669    #else
3670    #define _NV_WriteLockDataAddress 0
3671    #endif // CC_NV_WriteLock
3672    #if CC_NV_GlobalWriteLock
3673    #include "NV_GlobalWriteLock_fp.h"
3674    typedef TPM_RC  (NV_GlobalWriteLock_Entry)(
3675        NV_GlobalWriteLock_In           *in
3676    );
3677    typedef const struct {
3678        NV_GlobalWriteLock_Entry    *entry;
3679        UINT16                      inSize;
3680        UINT16                      outSize;
3681        UINT16                      offsetOfTypes;
3682        BYTE                        types[3];
3683    } NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t;
3684    NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t _NV_GlobalWriteLockData = {
3685        /* entry         */         &TPM2_NV_GlobalWriteLock,
3686        /* inSize        */         (UINT16)(sizeof(NV_GlobalWriteLock_In)),
3687        /* outSize       */         0,
3688        /* offsetOfTypes */         offsetof(NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t,
        types),
```



```
3689        /* offsets       */          // No parameter offsets;
3690        /* types         */          {TPMI_RH_PROVISION_H_UNMARSHAL,
3691                                       END_OF_LIST,
3692                                       END_OF_LIST}
3693    };
3694    #define _NV_GlobalWriteLockDataAddress (&_NV_GlobalWriteLockData)
3695    #else
3696    #define _NV_GlobalWriteLockDataAddress 0
3697    #endif // CC_NV_GlobalWriteLock
3698    #if CC_NV_Read
3699    #include "NV_Read_fp.h"
3700    typedef TPM_RC  (NV_Read_Entry)(
3701        NV_Read_In                  *in,
3702        NV_Read_Out                 *out
3703    );
3704    typedef const struct {
3705        NV_Read_Entry           *entry;
3706        UINT16                  inSize;
3707        UINT16                  outSize;
3708        UINT16                  offsetOfTypes;
3709        UINT16                  paramOffsets[3];
3710        BYTE                    types[7];
3711    } NV_Read_COMMAND_DESCRIPTOR_t;
3712    NV_Read_COMMAND_DESCRIPTOR_t _NV_ReadData = {
3713        /* entry         */     &TPM2_NV_Read,
3714        /* inSize        */     (UINT16)(sizeof(NV_Read_In)),
3715        /* outSize       */     (UINT16)(sizeof(NV_Read_Out)),
3716        /* offsetOfTypes */     offsetof(NV_Read_COMMAND_DESCRIPTOR_t, types),
3717        /* offsets       */     {(UINT16)(offsetof(NV_Read_In, nvIndex)),
3718                                 (UINT16)(offsetof(NV_Read_In, size)),
3719                                 (UINT16)(offsetof(NV_Read_In, offset))},
3720        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3721                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3722                                 UINT16_P_UNMARSHAL,
3723                                 UINT16_P_UNMARSHAL,
3724                                 END_OF_LIST,
3725                                 TPM2B_MAX_NV_BUFFER_P_MARSHAL,
3726                                 END_OF_LIST}
3727    };
3728    #define _NV_ReadDataAddress (&_NV_ReadData)
3729    #else
3730    #define _NV_ReadDataAddress 0
3731    #endif // CC_NV_Read
3732    #if CC_NV_ReadLock
3733    #include "NV_ReadLock_fp.h"
3734    typedef TPM_RC  (NV_ReadLock_Entry)(
3735        NV_ReadLock_In              *in
3736    );
3737    typedef const struct {
3738        NV_ReadLock_Entry       *entry;
3739        UINT16                  inSize;
3740        UINT16                  outSize;
3741        UINT16                  offsetOfTypes;
3742        UINT16                  paramOffsets[1];
3743        BYTE                    types[4];
3744    } NV_ReadLock_COMMAND_DESCRIPTOR_t;
3745    NV_ReadLock_COMMAND_DESCRIPTOR_t _NV_ReadLockData = {
3746        /* entry         */     &TPM2_NV_ReadLock,
3747        /* inSize        */     (UINT16)(sizeof(NV_ReadLock_In)),
3748        /* outSize       */     0,
3749        /* offsetOfTypes */     offsetof(NV_ReadLock_COMMAND_DESCRIPTOR_t, types),
3750        /* offsets       */     {(UINT16)(offsetof(NV_ReadLock_In, nvIndex))},
3751        /* types         */     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3752                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3753                                 END_OF_LIST,
3754                                 END_OF_LIST}
```

```
3755    };
3756    #define _NV_ReadLockDataAddress (&_NV_ReadLockData)
3757    #else
3758    #define _NV_ReadLockDataAddress 0
3759    #endif // CC_NV_ReadLock
3760    #if CC_NV_ChangeAuth
3761    #include "NV_ChangeAuth_fp.h"
3762    typedef TPM_RC  (NV_ChangeAuth_Entry)(
3763        NV_ChangeAuth_In            *in
3764    );
3765    typedef const struct {
3766        NV_ChangeAuth_Entry     *entry;
3767        UINT16                  inSize;
3768        UINT16                  outSize;
3769        UINT16                  offsetOfTypes;
3770        UINT16                  paramOffsets[1];
3771        BYTE                    types[4];
3772    } NV_ChangeAuth_COMMAND_DESCRIPTOR_t;
3773    NV_ChangeAuth_COMMAND_DESCRIPTOR_t _NV_ChangeAuthData = {
3774        /* entry         */     &TPM2_NV_ChangeAuth,
3775        /* inSize        */     (UINT16)(sizeof(NV_ChangeAuth_In)),
3776        /* outSize       */     0,
3777        /* offsetOfTypes */     offsetof(NV_ChangeAuth_COMMAND_DESCRIPTOR_t, types),
3778        /* offsets       */     {(UINT16)(offsetof(NV_ChangeAuth_In, newAuth))},
3779        /* types         */     {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3780                                 TPM2B_AUTH_P_UNMARSHAL,
3781                                 END_OF_LIST,
3782                                 END_OF_LIST}
3783    };
3784    #define _NV_ChangeAuthDataAddress (&_NV_ChangeAuthData)
3785    #else
3786    #define _NV_ChangeAuthDataAddress 0
3787    #endif // CC_NV_ChangeAuth
3788    #if CC_NV_Certify
3789    #include "NV_Certify_fp.h"
3790    typedef TPM_RC  (NV_Certify_Entry)(
3791        NV_Certify_In               *in,
3792        NV_Certify_Out              *out
3793    );
3794    typedef const struct {
3795        NV_Certify_Entry        *entry;
3796        UINT16                  inSize;
3797        UINT16                  outSize;
3798        UINT16                  offsetOfTypes;
3799        UINT16                  paramOffsets[7];
3800        BYTE                    types[11];
3801    } NV_Certify_COMMAND_DESCRIPTOR_t;
3802    NV_Certify_COMMAND_DESCRIPTOR_t _NV_CertifyData = {
3803        /* entry         */     &TPM2_NV_Certify,
3804        /* inSize        */     (UINT16)(sizeof(NV_Certify_In)),
3805        /* outSize       */     (UINT16)(sizeof(NV_Certify_Out)),
3806        /* offsetOfTypes */     offsetof(NV_Certify_COMMAND_DESCRIPTOR_t, types),
3807        /* offsets       */     {(UINT16)(offsetof(NV_Certify_In, authHandle)),
3808                                 (UINT16)(offsetof(NV_Certify_In, nvIndex)),
3809                                 (UINT16)(offsetof(NV_Certify_In, qualifyingData)),
3810                                 (UINT16)(offsetof(NV_Certify_In, inScheme)),
3811                                 (UINT16)(offsetof(NV_Certify_In, size)),
3812                                 (UINT16)(offsetof(NV_Certify_In, offset)),
3813                                 (UINT16)(offsetof(NV_Certify_Out, signature))},
3814        /* types         */     {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
3815                                 TPMI_RH_NV_AUTH_H_UNMARSHAL,
3816                                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3817                                 TPM2B_DATA_P_UNMARSHAL,
3818                                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
3819                                 UINT16_P_UNMARSHAL,
3820                                 UINT16_P_UNMARSHAL,
```



```
3821                                END_OF_LIST,
3822                                TPM2B_ATTEST_P_MARSHAL,
3823                                TPMT_SIGNATURE_P_MARSHAL,
3824                                END_OF_LIST}
3825    };
3826    #define _NV_CertifyDataAddress (&_NV_CertifyData)
3827    #else
3828    #define _NV_CertifyDataAddress 0
3829    #endif // CC_NV_Certify
3830    #if CC_AC_GetCapability
3831    #include "AC_GetCapability_fp.h"
3832    typedef TPM_RC  (AC_GetCapability_Entry)(
3833        AC_GetCapability_In         *in,
3834        AC_GetCapability_Out        *out
3835    );
3836    typedef const struct {
3837        AC_GetCapability_Entry  *entry;
3838        UINT16                  inSize;
3839        UINT16                  outSize;
3840        UINT16                  offsetOfTypes;
3841        UINT16                  paramOffsets[3];
3842        BYTE                    types[7];
3843    } AC_GetCapability_COMMAND_DESCRIPTOR_t;
3844    AC_GetCapability_COMMAND_DESCRIPTOR_t _AC_GetCapabilityData = {
3845        /* entry         */     &TPM2_AC_GetCapability,
3846        /* inSize        */     (UINT16)(sizeof(AC_GetCapability_In)),
3847        /* outSize       */     (UINT16)(sizeof(AC_GetCapability_Out)),
3848        /* offsetOfTypes */     offsetof(AC_GetCapability_COMMAND_DESCRIPTOR_t, types),
3849        /* offsets       */     {(UINT16)(offsetof(AC_GetCapability_In, capability)),
3850                                 (UINT16)(offsetof(AC_GetCapability_In, count)),
3851                                 (UINT16)(offsetof(AC_GetCapability_Out,
        capabilitiesData))},
3852        /* types         */     {TPMI_RH_AC_H_UNMARSHAL,
3853                                 TPM_AT_P_UNMARSHAL,
3854                                 UINT32_P_UNMARSHAL,
3855                                 END_OF_LIST,
3856                                 TPMI_YES_NO_P_MARSHAL,
3857                                 TPML_AC_CAPABILITIES_P_MARSHAL,
3858                                 END_OF_LIST}
3859    };
3860    #define _AC_GetCapabilityDataAddress (&_AC_GetCapabilityData)
3861    #else
3862    #define _AC_GetCapabilityDataAddress 0
3863    #endif // CC_AC_GetCapability
3864    #if CC_AC_Send
3865    #include "AC_Send_fp.h"
3866    typedef TPM_RC  (AC_Send_Entry)(
3867        AC_Send_In                  *in,
3868        AC_Send_Out                 *out
3869    );
3870    typedef const struct {
3871        AC_Send_Entry           *entry;
3872        UINT16                  inSize;
3873        UINT16                  outSize;
3874        UINT16                  offsetOfTypes;
3875        UINT16                  paramOffsets[3];
3876        BYTE                    types[7];
3877    } AC_Send_COMMAND_DESCRIPTOR_t;
3878    AC_Send_COMMAND_DESCRIPTOR_t _AC_SendData = {
3879        /* entry         */     &TPM2_AC_Send,
3880        /* inSize        */     (UINT16)(sizeof(AC_Send_In)),
3881        /* outSize       */     (UINT16)(sizeof(AC_Send_Out)),
3882        /* offsetOfTypes */     offsetof(AC_Send_COMMAND_DESCRIPTOR_t, types),
3883        /* offsets       */     {(UINT16)(offsetof(AC_Send_In, authHandle)),
3884                                 (UINT16)(offsetof(AC_Send_In, ac)),
3885                                 (UINT16)(offsetof(AC_Send_In, acDataIn))},
```

```
3886        /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL,
3887                                    TPMI_RH_NV_AUTH_H_UNMARSHAL,
3888                                    TPMI_RH_AC_H_UNMARSHAL,
3889                                    TPM2B_MAX_BUFFER_P_UNMARSHAL,
3890                                    END_OF_LIST,
3891                                    TPMS_AC_OUTPUT_P_MARSHAL,
3892                                    END_OF_LIST}
3893    };
3894    #define _AC_SendDataAddress (&_AC_SendData)
3895    #else
3896    #define _AC_SendDataAddress 0
3897    #endif // CC_AC_Send
3898    #if CC_Policy_AC_SendSelect
3899    #include "Policy_AC_SendSelect_fp.h"
3900    typedef TPM_RC  (Policy_AC_SendSelect_Entry)(
3901        Policy_AC_SendSelect_In         *in
3902    );
3903    typedef const struct {
3904        Policy_AC_SendSelect_Entry  *entry;
3905        UINT16                      inSize;
3906        UINT16                      outSize;
3907        UINT16                      offsetOfTypes;
3908        UINT16                      paramOffsets[4];
3909        BYTE                        types[7];
3910    } Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t;
3911    Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t _Policy_AC_SendSelectData = {
3912        /* entry          */         &TPM2_Policy_AC_SendSelect,
3913        /* inSize         */         (UINT16)(sizeof(Policy_AC_SendSelect_In)),
3914        /* outSize        */         0,
3915        /* offsetOfTypes */          offsetof(Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t,
        types),
3916        /* offsets        */         {(UINT16)(offsetof(Policy_AC_SendSelect_In,
        objectName)),
3917                                     (UINT16)(offsetof(Policy_AC_SendSelect_In,
        authHandleName)),
3918                                     (UINT16)(offsetof(Policy_AC_SendSelect_In, acName)),
3919                                     (UINT16)(offsetof(Policy_AC_SendSelect_In,
        includeObject))},
3920        /* types          */         {TPMI_SH_POLICY_H_UNMARSHAL,
3921                                     TPM2B_NAME_P_UNMARSHAL,
3922                                     TPM2B_NAME_P_UNMARSHAL,
3923                                     TPM2B_NAME_P_UNMARSHAL,
3924                                     TPMI_YES_NO_P_UNMARSHAL,
3925                                     END_OF_LIST,
3926                                     END_OF_LIST}
3927    };
3928    #define _Policy_AC_SendSelectDataAddress (&_Policy_AC_SendSelectData)
3929    #else
3930    #define _Policy_AC_SendSelectDataAddress 0
3931    #endif // CC_Policy_AC_SendSelect
3932    #if CC_Vendor_TCG_Test
3933    #include "Vendor_TCG_Test_fp.h"
3934    typedef TPM_RC  (Vendor_TCG_Test_Entry)(
3935        Vendor_TCG_Test_In          *in,
3936        Vendor_TCG_Test_Out         *out
3937    );
3938    typedef const struct {
3939        Vendor_TCG_Test_Entry   *entry;
3940        UINT16                  inSize;
3941        UINT16                  outSize;
3942        UINT16                  offsetOfTypes;
3943        BYTE                    types[4];
3944    } Vendor_TCG_Test_COMMAND_DESCRIPTOR_t;
3945    Vendor_TCG_Test_COMMAND_DESCRIPTOR_t _Vendor_TCG_TestData = {
3946        /* entry          */     &TPM2_Vendor_TCG_Test,
3947        /* inSize         */     (UINT16)(sizeof(Vendor_TCG_Test_In)),
```



```
3948        /* outSize       */     (UINT16)(sizeof(Vendor_TCG_Test_Out)),
3949        /* offsetOfTypes */     offsetof(Vendor_TCG_Test_COMMAND_DESCRIPTOR_t, types),
3950        /* offsets       */     // No parameter offsets;
3951        /* types         */     {TPM2B_DATA_P_UNMARSHAL,
3952                                 END_OF_LIST,
3953                                 TPM2B_DATA_P_MARSHAL,
3954                                 END_OF_LIST}
3955    };
3956    #define _Vendor_TCG_TestDataAddress (&_Vendor_TCG_TestData)
3957    #else
3958    #define _Vendor_TCG_TestDataAddress 0
3959    #endif // CC_Vendor_TCG_Test
3960    COMMAND_DESCRIPTOR_t *s_CommandDataArray[] = {
3961    #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
3962            (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceSpecialDataAddress,
3963    #endif // CC_NV_UndefineSpaceSpecial
3964    #if (PAD_LIST || CC_EvictControl)
3965            (COMMAND_DESCRIPTOR_t *)_EvictControlDataAddress,
3966    #endif // CC_EvictControl
3967    #if (PAD_LIST || CC_HierarchyControl)
3968            (COMMAND_DESCRIPTOR_t *)_HierarchyControlDataAddress,
3969    #endif // CC_HierarchyControl
3970    #if (PAD_LIST || CC_NV_UndefineSpace)
3971            (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceDataAddress,
3972    #endif // CC_NV_UndefineSpace
3973    #if (PAD_LIST)
3974            (COMMAND_DESCRIPTOR_t *)0,
3975    #endif //
3976    #if (PAD_LIST || CC_ChangeEPS)
3977            (COMMAND_DESCRIPTOR_t *)_ChangeEPSDataAddress,
3978    #endif // CC_ChangeEPS
3979    #if (PAD_LIST || CC_ChangePPS)
3980            (COMMAND_DESCRIPTOR_t *)_ChangePPSDataAddress,
3981    #endif // CC_ChangePPS
3982    #if (PAD_LIST || CC_Clear)
3983            (COMMAND_DESCRIPTOR_t *)_ClearDataAddress,
3984    #endif // CC_Clear
3985    #if (PAD_LIST || CC_ClearControl)
3986            (COMMAND_DESCRIPTOR_t *)_ClearControlDataAddress,
3987    #endif // CC_ClearControl
3988    #if (PAD_LIST || CC_ClockSet)
3989            (COMMAND_DESCRIPTOR_t *)_ClockSetDataAddress,
3990    #endif // CC_ClockSet
3991    #if (PAD_LIST || CC_HierarchyChangeAuth)
3992            (COMMAND_DESCRIPTOR_t *)_HierarchyChangeAuthDataAddress,
3993    #endif // CC_HierarchyChangeAuth
3994    #if (PAD_LIST || CC_NV_DefineSpace)
3995            (COMMAND_DESCRIPTOR_t *)_NV_DefineSpaceDataAddress,
3996    #endif // CC_NV_DefineSpace
3997    #if (PAD_LIST || CC_PCR_Allocate)
3998            (COMMAND_DESCRIPTOR_t *)_PCR_AllocateDataAddress,
3999    #endif // CC_PCR_Allocate
4000    #if (PAD_LIST || CC_PCR_SetAuthPolicy)
4001            (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthPolicyDataAddress,
4002    #endif // CC_PCR_SetAuthPolicy
4003    #if (PAD_LIST || CC_PP_Commands)
4004            (COMMAND_DESCRIPTOR_t *)_PP_CommandsDataAddress,
4005    #endif // CC_PP_Commands
4006    #if (PAD_LIST || CC_SetPrimaryPolicy)
4007            (COMMAND_DESCRIPTOR_t *)_SetPrimaryPolicyDataAddress,
4008    #endif // CC_SetPrimaryPolicy
4009    #if (PAD_LIST || CC_FieldUpgradeStart)
4010            (COMMAND_DESCRIPTOR_t *)_FieldUpgradeStartDataAddress,
4011    #endif // CC_FieldUpgradeStart
4012    #if (PAD_LIST || CC_ClockRateAdjust)
4013            (COMMAND_DESCRIPTOR_t *)_ClockRateAdjustDataAddress,
```

```
4014    #endif // CC_ClockRateAdjust
4015    #if (PAD_LIST || CC_CreatePrimary)
4016            (COMMAND_DESCRIPTOR_t *)_CreatePrimaryDataAddress,
4017    #endif // CC_CreatePrimary
4018    #if (PAD_LIST || CC_NV_GlobalWriteLock)
4019            (COMMAND_DESCRIPTOR_t *)_NV_GlobalWriteLockDataAddress,
4020    #endif // CC_NV_GlobalWriteLock
4021    #if (PAD_LIST || CC_GetCommandAuditDigest)
4022            (COMMAND_DESCRIPTOR_t *)_GetCommandAuditDigestDataAddress,
4023    #endif // CC_GetCommandAuditDigest
4024    #if (PAD_LIST || CC_NV_Increment)
4025            (COMMAND_DESCRIPTOR_t *)_NV_IncrementDataAddress,
4026    #endif // CC_NV_Increment
4027    #if (PAD_LIST || CC_NV_SetBits)
4028            (COMMAND_DESCRIPTOR_t *)_NV_SetBitsDataAddress,
4029    #endif // CC_NV_SetBits
4030    #if (PAD_LIST || CC_NV_Extend)
4031            (COMMAND_DESCRIPTOR_t *)_NV_ExtendDataAddress,
4032    #endif // CC_NV_Extend
4033    #if (PAD_LIST || CC_NV_Write)
4034            (COMMAND_DESCRIPTOR_t *)_NV_WriteDataAddress,
4035    #endif // CC_NV_Write
4036    #if (PAD_LIST || CC_NV_WriteLock)
4037            (COMMAND_DESCRIPTOR_t *)_NV_WriteLockDataAddress,
4038    #endif // CC_NV_WriteLock
4039    #if (PAD_LIST || CC_DictionaryAttackLockReset)
4040            (COMMAND_DESCRIPTOR_t *)_DictionaryAttackLockResetDataAddress,
4041    #endif // CC_DictionaryAttackLockReset
4042    #if (PAD_LIST || CC_DictionaryAttackParameters)
4043            (COMMAND_DESCRIPTOR_t *)_DictionaryAttackParametersDataAddress,
4044    #endif // CC_DictionaryAttackParameters
4045    #if (PAD_LIST || CC_NV_ChangeAuth)
4046            (COMMAND_DESCRIPTOR_t *)_NV_ChangeAuthDataAddress,
4047    #endif // CC_NV_ChangeAuth
4048    #if (PAD_LIST || CC_PCR_Event)
4049            (COMMAND_DESCRIPTOR_t *)_PCR_EventDataAddress,
4050    #endif // CC_PCR_Event
4051    #if (PAD_LIST || CC_PCR_Reset)
4052            (COMMAND_DESCRIPTOR_t *)_PCR_ResetDataAddress,
4053    #endif // CC_PCR_Reset
4054    #if (PAD_LIST || CC_SequenceComplete)
4055            (COMMAND_DESCRIPTOR_t *)_SequenceCompleteDataAddress,
4056    #endif // CC_SequenceComplete
4057    #if (PAD_LIST || CC_SetAlgorithmSet)
4058            (COMMAND_DESCRIPTOR_t *)_SetAlgorithmSetDataAddress,
4059    #endif // CC_SetAlgorithmSet
4060    #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
4061            (COMMAND_DESCRIPTOR_t *)_SetCommandCodeAuditStatusDataAddress,
4062    #endif // CC_SetCommandCodeAuditStatus
4063    #if (PAD_LIST || CC_FieldUpgradeData)
4064            (COMMAND_DESCRIPTOR_t *)_FieldUpgradeDataDataAddress,
4065    #endif // CC_FieldUpgradeData
4066    #if (PAD_LIST || CC_IncrementalSelfTest)
4067            (COMMAND_DESCRIPTOR_t *)_IncrementalSelfTestDataAddress,
4068    #endif // CC_IncrementalSelfTest
4069    #if (PAD_LIST || CC_SelfTest)
4070            (COMMAND_DESCRIPTOR_t *)_SelfTestDataAddress,
4071    #endif // CC_SelfTest
4072    #if (PAD_LIST || CC_Startup)
4073            (COMMAND_DESCRIPTOR_t *)_StartupDataAddress,
4074    #endif // CC_Startup
4075    #if (PAD_LIST || CC_Shutdown)
4076            (COMMAND_DESCRIPTOR_t *)_ShutdownDataAddress,
4077    #endif // CC_Shutdown
4078    #if (PAD_LIST || CC_StirRandom)
4079            (COMMAND_DESCRIPTOR_t *)_StirRandomDataAddress,
```



```
4080    #endif // CC_StirRandom
4081    #if (PAD_LIST || CC_ActivateCredential)
4082            (COMMAND_DESCRIPTOR_t *)_ActivateCredentialDataAddress,
4083    #endif // CC_ActivateCredential
4084    #if (PAD_LIST || CC_Certify)
4085            (COMMAND_DESCRIPTOR_t *)_CertifyDataAddress,
4086    #endif // CC_Certify
4087    #if (PAD_LIST || CC_PolicyNV)
4088            (COMMAND_DESCRIPTOR_t *)_PolicyNVDataAddress,
4089    #endif // CC_PolicyNV
4090    #if (PAD_LIST || CC_CertifyCreation)
4091            (COMMAND_DESCRIPTOR_t *)_CertifyCreationDataAddress,
4092    #endif // CC_CertifyCreation
4093    #if (PAD_LIST || CC_Duplicate)
4094            (COMMAND_DESCRIPTOR_t *)_DuplicateDataAddress,
4095    #endif // CC_Duplicate
4096    #if (PAD_LIST || CC_GetTime)
4097            (COMMAND_DESCRIPTOR_t *)_GetTimeDataAddress,
4098    #endif // CC_GetTime
4099    #if (PAD_LIST || CC_GetSessionAuditDigest)
4100            (COMMAND_DESCRIPTOR_t *)_GetSessionAuditDigestDataAddress,
4101    #endif // CC_GetSessionAuditDigest
4102    #if (PAD_LIST || CC_NV_Read)
4103            (COMMAND_DESCRIPTOR_t *)_NV_ReadDataAddress,
4104    #endif // CC_NV_Read
4105    #if (PAD_LIST || CC_NV_ReadLock)
4106            (COMMAND_DESCRIPTOR_t *)_NV_ReadLockDataAddress,
4107    #endif // CC_NV_ReadLock
4108    #if (PAD_LIST || CC_ObjectChangeAuth)
4109            (COMMAND_DESCRIPTOR_t *)_ObjectChangeAuthDataAddress,
4110    #endif // CC_ObjectChangeAuth
4111    #if (PAD_LIST || CC_PolicySecret)
4112            (COMMAND_DESCRIPTOR_t *)_PolicySecretDataAddress,
4113    #endif // CC_PolicySecret
4114    #if (PAD_LIST || CC_Rewrap)
4115            (COMMAND_DESCRIPTOR_t *)_RewrapDataAddress,
4116    #endif // CC_Rewrap
4117    #if (PAD_LIST || CC_Create)
4118            (COMMAND_DESCRIPTOR_t *)_CreateDataAddress,
4119    #endif // CC_Create
4120    #if (PAD_LIST || CC_ECDH_ZGen)
4121            (COMMAND_DESCRIPTOR_t *)_ECDH_ZGenDataAddress,
4122    #endif // CC_ECDH_ZGen
4123    #if (PAD_LIST || (CC_HMAC || CC_MAC))
4124    #    if CC_HMAC
4125            (COMMAND_DESCRIPTOR_t *)_HMACDataAddress,
4126    #    endif
4127    #    if CC_MAC
4128            (COMMAND_DESCRIPTOR_t *)_MACDataAddress,
4129    #    endif
4130    #    if (CC_HMAC || CC_MAC) > 1
4131    #        error "More than one aliased command defined"
4132    #    endif
4133    #endif // CC_HMAC CC_MAC
4134    #if (PAD_LIST || CC_Import)
4135            (COMMAND_DESCRIPTOR_t *)_ImportDataAddress,
4136    #endif // CC_Import
4137    #if (PAD_LIST || CC_Load)
4138            (COMMAND_DESCRIPTOR_t *)_LoadDataAddress,
4139    #endif // CC_Load
4140    #if (PAD_LIST || CC_Quote)
4141            (COMMAND_DESCRIPTOR_t *)_QuoteDataAddress,
4142    #endif // CC_Quote
4143    #if (PAD_LIST || CC_RSA_Decrypt)
4144            (COMMAND_DESCRIPTOR_t *)_RSA_DecryptDataAddress,
4145    #endif // CC_RSA_Decrypt
```

```
4146    #if (PAD_LIST)
4147            (COMMAND_DESCRIPTOR_t *)0,
4148    #endif //
4149    #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
4150    #    if CC_HMAC_Start
4151            (COMMAND_DESCRIPTOR_t *)_HMAC_StartDataAddress,
4152    #    endif
4153    #    if CC_MAC_Start
4154            (COMMAND_DESCRIPTOR_t *)_MAC_StartDataAddress,
4155    #    endif
4156    #    if (CC_HMAC_Start || CC_MAC_Start) > 1
4157    #        error "More than one aliased command defined"
4158    #    endif
4159    #endif // CC_HMAC_Start CC_MAC_Start
4160    #if (PAD_LIST || CC_SequenceUpdate)
4161            (COMMAND_DESCRIPTOR_t *)_SequenceUpdateDataAddress,
4162    #endif // CC_SequenceUpdate
4163    #if (PAD_LIST || CC_Sign)
4164            (COMMAND_DESCRIPTOR_t *)_SignDataAddress,
4165    #endif // CC_Sign
4166    #if (PAD_LIST || CC_Unseal)
4167            (COMMAND_DESCRIPTOR_t *)_UnsealDataAddress,
4168    #endif // CC_Unseal
4169    #if (PAD_LIST)
4170            (COMMAND_DESCRIPTOR_t *)0,
4171    #endif //
4172    #if (PAD_LIST || CC_PolicySigned)
4173            (COMMAND_DESCRIPTOR_t *)_PolicySignedDataAddress,
4174    #endif // CC_PolicySigned
4175    #if (PAD_LIST || CC_ContextLoad)
4176            (COMMAND_DESCRIPTOR_t *)_ContextLoadDataAddress,
4177    #endif // CC_ContextLoad
4178    #if (PAD_LIST || CC_ContextSave)
4179            (COMMAND_DESCRIPTOR_t *)_ContextSaveDataAddress,
4180    #endif // CC_ContextSave
4181    #if (PAD_LIST || CC_ECDH_KeyGen)
4182            (COMMAND_DESCRIPTOR_t *)_ECDH_KeyGenDataAddress,
4183    #endif // CC_ECDH_KeyGen
4184    #if (PAD_LIST || CC_EncryptDecrypt)
4185            (COMMAND_DESCRIPTOR_t *)_EncryptDecryptDataAddress,
4186    #endif // CC_EncryptDecrypt
4187    #if (PAD_LIST || CC_FlushContext)
4188            (COMMAND_DESCRIPTOR_t *)_FlushContextDataAddress,
4189    #endif // CC_FlushContext
4190    #if (PAD_LIST)
4191            (COMMAND_DESCRIPTOR_t *)0,
4192    #endif //
4193    #if (PAD_LIST || CC_LoadExternal)
4194            (COMMAND_DESCRIPTOR_t *)_LoadExternalDataAddress,
4195    #endif // CC_LoadExternal
4196    #if (PAD_LIST || CC_MakeCredential)
4197            (COMMAND_DESCRIPTOR_t *)_MakeCredentialDataAddress,
4198    #endif // CC_MakeCredential
4199    #if (PAD_LIST || CC_NV_ReadPublic)
4200            (COMMAND_DESCRIPTOR_t *)_NV_ReadPublicDataAddress,
4201    #endif // CC_NV_ReadPublic
4202    #if (PAD_LIST || CC_PolicyAuthorize)
4203            (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeDataAddress,
4204    #endif // CC_PolicyAuthorize
4205    #if (PAD_LIST || CC_PolicyAuthValue)
4206            (COMMAND_DESCRIPTOR_t *)_PolicyAuthValueDataAddress,
4207    #endif // CC_PolicyAuthValue
4208    #if (PAD_LIST || CC_PolicyCommandCode)
4209            (COMMAND_DESCRIPTOR_t *)_PolicyCommandCodeDataAddress,
4210    #endif // CC_PolicyCommandCode
4211    #if (PAD_LIST || CC_PolicyCounterTimer)
```



```
4212            (COMMAND_DESCRIPTOR_t *)_PolicyCounterTimerDataAddress,
4213    #endif // CC_PolicyCounterTimer
4214    #if (PAD_LIST || CC_PolicyCpHash)
4215            (COMMAND_DESCRIPTOR_t *)_PolicyCpHashDataAddress,
4216    #endif // CC_PolicyCpHash
4217    #if (PAD_LIST || CC_PolicyLocality)
4218            (COMMAND_DESCRIPTOR_t *)_PolicyLocalityDataAddress,
4219    #endif // CC_PolicyLocality
4220    #if (PAD_LIST || CC_PolicyNameHash)
4221            (COMMAND_DESCRIPTOR_t *)_PolicyNameHashDataAddress,
4222    #endif // CC_PolicyNameHash
4223    #if (PAD_LIST || CC_PolicyOR)
4224            (COMMAND_DESCRIPTOR_t *)_PolicyORDataAddress,
4225    #endif // CC_PolicyOR
4226    #if (PAD_LIST || CC_PolicyTicket)
4227            (COMMAND_DESCRIPTOR_t *)_PolicyTicketDataAddress,
4228    #endif // CC_PolicyTicket
4229    #if (PAD_LIST || CC_ReadPublic)
4230            (COMMAND_DESCRIPTOR_t *)_ReadPublicDataAddress,
4231    #endif // CC_ReadPublic
4232    #if (PAD_LIST || CC_RSA_Encrypt)
4233            (COMMAND_DESCRIPTOR_t *)_RSA_EncryptDataAddress,
4234    #endif // CC_RSA_Encrypt
4235    #if (PAD_LIST)
4236            (COMMAND_DESCRIPTOR_t *)0,
4237    #endif //
4238    #if (PAD_LIST || CC_StartAuthSession)
4239            (COMMAND_DESCRIPTOR_t *)_StartAuthSessionDataAddress,
4240    #endif // CC_StartAuthSession
4241    #if (PAD_LIST || CC_VerifySignature)
4242            (COMMAND_DESCRIPTOR_t *)_VerifySignatureDataAddress,
4243    #endif // CC_VerifySignature
4244    #if (PAD_LIST || CC_ECC_Parameters)
4245            (COMMAND_DESCRIPTOR_t *)_ECC_ParametersDataAddress,
4246    #endif // CC_ECC_Parameters
4247    #if (PAD_LIST || CC_FirmwareRead)
4248            (COMMAND_DESCRIPTOR_t *)_FirmwareReadDataAddress,
4249    #endif // CC_FirmwareRead
4250    #if (PAD_LIST || CC_GetCapability)
4251            (COMMAND_DESCRIPTOR_t *)_GetCapabilityDataAddress,
4252    #endif // CC_GetCapability
4253    #if (PAD_LIST || CC_GetRandom)
4254            (COMMAND_DESCRIPTOR_t *)_GetRandomDataAddress,
4255    #endif // CC_GetRandom
4256    #if (PAD_LIST || CC_GetTestResult)
4257            (COMMAND_DESCRIPTOR_t *)_GetTestResultDataAddress,
4258    #endif // CC_GetTestResult
4259    #if (PAD_LIST || CC_Hash)
4260            (COMMAND_DESCRIPTOR_t *)_HashDataAddress,
4261    #endif // CC_Hash
4262    #if (PAD_LIST || CC_PCR_Read)
4263            (COMMAND_DESCRIPTOR_t *)_PCR_ReadDataAddress,
4264    #endif // CC_PCR_Read
4265    #if (PAD_LIST || CC_PolicyPCR)
4266            (COMMAND_DESCRIPTOR_t *)_PolicyPCRDataAddress,
4267    #endif // CC_PolicyPCR
4268    #if (PAD_LIST || CC_PolicyRestart)
4269            (COMMAND_DESCRIPTOR_t *)_PolicyRestartDataAddress,
4270    #endif // CC_PolicyRestart
4271    #if (PAD_LIST || CC_ReadClock)
4272            (COMMAND_DESCRIPTOR_t *)_ReadClockDataAddress,
4273    #endif // CC_ReadClock
4274    #if (PAD_LIST || CC_PCR_Extend)
4275            (COMMAND_DESCRIPTOR_t *)_PCR_ExtendDataAddress,
4276    #endif // CC_PCR_Extend
4277    #if (PAD_LIST || CC_PCR_SetAuthValue)
```

```
4278            (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthValueDataAddress,
4279    #endif // CC_PCR_SetAuthValue
4280    #if (PAD_LIST || CC_NV_Certify)
4281            (COMMAND_DESCRIPTOR_t *)_NV_CertifyDataAddress,
4282    #endif // CC_NV_Certify
4283    #if (PAD_LIST || CC_EventSequenceComplete)
4284            (COMMAND_DESCRIPTOR_t *)_EventSequenceCompleteDataAddress,
4285    #endif // CC_EventSequenceComplete
4286    #if (PAD_LIST || CC_HashSequenceStart)
4287            (COMMAND_DESCRIPTOR_t *)_HashSequenceStartDataAddress,
4288    #endif // CC_HashSequenceStart
4289    #if (PAD_LIST || CC_PolicyPhysicalPresence)
4290            (COMMAND_DESCRIPTOR_t *)_PolicyPhysicalPresenceDataAddress,
4291    #endif // CC_PolicyPhysicalPresence
4292    #if (PAD_LIST || CC_PolicyDuplicationSelect)
4293            (COMMAND_DESCRIPTOR_t *)_PolicyDuplicationSelectDataAddress,
4294    #endif // CC_PolicyDuplicationSelect
4295    #if (PAD_LIST || CC_PolicyGetDigest)
4296            (COMMAND_DESCRIPTOR_t *)_PolicyGetDigestDataAddress,
4297    #endif // CC_PolicyGetDigest
4298    #if (PAD_LIST || CC_TestParms)
4299            (COMMAND_DESCRIPTOR_t *)_TestParmsDataAddress,
4300    #endif // CC_TestParms
4301    #if (PAD_LIST || CC_Commit)
4302            (COMMAND_DESCRIPTOR_t *)_CommitDataAddress,
4303    #endif // CC_Commit
4304    #if (PAD_LIST || CC_PolicyPassword)
4305            (COMMAND_DESCRIPTOR_t *)_PolicyPasswordDataAddress,
4306    #endif // CC_PolicyPassword
4307    #if (PAD_LIST || CC_ZGen_2Phase)
4308            (COMMAND_DESCRIPTOR_t *)_ZGen_2PhaseDataAddress,
4309    #endif // CC_ZGen_2Phase
4310    #if (PAD_LIST || CC_EC_Ephemeral)
4311            (COMMAND_DESCRIPTOR_t *)_EC_EphemeralDataAddress,
4312    #endif // CC_EC_Ephemeral
4313    #if (PAD_LIST || CC_PolicyNvWritten)
4314            (COMMAND_DESCRIPTOR_t *)_PolicyNvWrittenDataAddress,
4315    #endif // CC_PolicyNvWritten
4316    #if (PAD_LIST || CC_PolicyTemplate)
4317            (COMMAND_DESCRIPTOR_t *)_PolicyTemplateDataAddress,
4318    #endif // CC_PolicyTemplate
4319    #if (PAD_LIST || CC_CreateLoaded)
4320            (COMMAND_DESCRIPTOR_t *)_CreateLoadedDataAddress,
4321    #endif // CC_CreateLoaded
4322    #if (PAD_LIST || CC_PolicyAuthorizeNV)
4323            (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeNVDataAddress,
4324    #endif // CC_PolicyAuthorizeNV
4325    #if (PAD_LIST || CC_EncryptDecrypt2)
4326            (COMMAND_DESCRIPTOR_t *)_EncryptDecrypt2DataAddress,
4327    #endif // CC_EncryptDecrypt2
4328    #if (PAD_LIST || CC_AC_GetCapability)
4329            (COMMAND_DESCRIPTOR_t *)_AC_GetCapabilityDataAddress,
4330    #endif // CC_AC_GetCapability
4331    #if (PAD_LIST || CC_AC_Send)
4332            (COMMAND_DESCRIPTOR_t *)_AC_SendDataAddress,
4333    #endif // CC_AC_Send
4334    #if (PAD_LIST || CC_Policy_AC_SendSelect)
4335            (COMMAND_DESCRIPTOR_t *)_Policy_AC_SendSelectDataAddress,
4336    #endif // CC_Policy_AC_SendSelect
4337    #if (PAD_LIST || CC_Vendor_TCG_Test)
4338            (COMMAND_DESCRIPTOR_t *)_Vendor_TCG_TestDataAddress,
4339    #endif // CC_Vendor_TCG_Test
4340            0
4341    };
4342    #endif  // _COMMAND_TABLE_DISPATCH_
```


5.7 Commands.h

```
1   #ifndef _COMMANDS_H_
2   #define _COMMANDS_H_
```

Start-up

```
3   #ifdef TPM_CC_Startup
4   #include "Startup_fp.h"
5   #endif
6   #ifdef TPM_CC_Shutdown
7   #include "Shutdown_fp.h"
8   #endif
```

Testing

```
9   #ifdef TPM_CC_SelfTest
10  #include "SelfTest_fp.h"
11  #endif
12  #ifdef TPM_CC_IncrementalSelfTest
13  #include "IncrementalSelfTest_fp.h"
14  #endif
15  #ifdef TPM_CC_GetTestResult
16  #include "GetTestResult_fp.h"
17  #endif
```

Session Commands

```
18  #ifdef TPM_CC_StartAuthSession
19  #include "StartAuthSession_fp.h"
20  #endif
21  #ifdef TPM_CC_PolicyRestart
22  #include "PolicyRestart_fp.h"
23  #endif
```

Object Commands

```
24  #ifdef TPM_CC_Create
25  #include "Create_fp.h"
26  #endif
27  #ifdef TPM_CC_Load
28  #include "Load_fp.h"
29  #endif
30  #ifdef TPM_CC_LoadExternal
31  #include "LoadExternal_fp.h"
32  #endif
33  #ifdef TPM_CC_ReadPublic
34  #include "ReadPublic_fp.h"
35  #endif
36  #ifdef TPM_CC_ActivateCredential
37  #include "ActivateCredential_fp.h"
38  #endif
39  #ifdef TPM_CC_MakeCredential
40  #include "MakeCredential_fp.h"
41  #endif
42  #ifdef TPM_CC_Unseal
43  #include "Unseal_fp.h"
44  #endif
45  #ifdef TPM_CC_ObjectChangeAuth
46  #include "ObjectChangeAuth_fp.h"
47  #endif
48  #ifdef TPM_CC_CreateLoaded
49  #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
90  #ifdef TPM_CC_MAC
91  #include "MAC_fp.h"
92  #endif
```

Random Number Generator

```
93  #ifdef TPM_CC_GetRandom
94  #include "GetRandom_fp.h"
95  #endif
96  #ifdef TPM_CC_StirRandom
97  #include "StirRandom_fp.h"
98  #endif
```

Hash/HMAC/Event Sequences


```
99   #ifdef TPM_CC_HMAC_Start
100  #include "HMAC_Start_fp.h"
101  #endif
102  #ifdef TPM_CC_MAC_Start
103  #include "MAC_Start_fp.h"
104  #endif
105  #ifdef TPM_CC_HashSequenceStart
106  #include "HashSequenceStart_fp.h"
107  #endif
108  #ifdef TPM_CC_SequenceUpdate
109  #include "SequenceUpdate_fp.h"
110  #endif
111  #ifdef TPM_CC_SequenceComplete
112  #include "SequenceComplete_fp.h"
113  #endif
114  #ifdef TPM_CC_EventSequenceComplete
115  #include "EventSequenceComplete_fp.h"
116  #endif
```

Attestation Commands

```
117  #ifdef TPM_CC_Certify
118  #include "Certify_fp.h"
119  #endif
120  #ifdef TPM_CC_CertifyCreation
121  #include "CertifyCreation_fp.h"
122  #endif
123  #ifdef TPM_CC_Quote
124  #include "Quote_fp.h"
125  #endif
126  #ifdef TPM_CC_GetSessionAuditDigest
127  #include "GetSessionAuditDigest_fp.h"
128  #endif
129  #ifdef TPM_CC_GetCommandAuditDigest
130  #include "GetCommandAuditDigest_fp.h"
131  #endif
132  #ifdef TPM_CC_GetTime
133  #include "GetTime_fp.h"
134  #endif
```

Ephemeral EC Keys

```
135  #ifdef TPM_CC_Commit
136  #include "Commit_fp.h"
137  #endif
138  #ifdef TPM_CC_EC_Ephemeral
139  #include "EC_Ephemeral_fp.h"
140  #endif
```

Signing and Signature Verification

```
141  #ifdef TPM_CC_VerifySignature
142  #include "VerifySignature_fp.h"
143  #endif
144  #ifdef TPM_CC_Sign
145  #include "Sign_fp.h"
146  #endif
```

Command Audit

```
147  #ifdef TPM_CC_SetCommandCodeAuditStatus
148  #include "SetCommandCodeAuditStatus_fp.h"
149  #endif
```

Integrity Collection (PCR)

```
150 #ifdef TPM_CC_PCR_Extend
151 #include "PCR_Extend_fp.h"
152 #endif
153 #ifdef TPM_CC_PCR_Event
154 #include "PCR_Event_fp.h"
155 #endif
156 #ifdef TPM_CC_PCR_Read
157 #include "PCR_Read_fp.h"
158 #endif
159 #ifdef TPM_CC_PCR_Allocate
160 #include "PCR_Allocate_fp.h"
161 #endif
162 #ifdef TPM_CC_PCR_SetAuthPolicy
163 #include "PCR_SetAuthPolicy_fp.h"
164 #endif
165 #ifdef TPM_CC_PCR_SetAuthValue
166 #include "PCR_SetAuthValue_fp.h"
167 #endif
168 #ifdef TPM_CC_PCR_Reset
169 #include "PCR_Reset_fp.h"
170 #endif
```

Enhanced Authorization (EA) Commands

```
171 #ifdef TPM_CC_PolicySigned
172 #include "PolicySigned_fp.h"
173 #endif
174 #ifdef TPM_CC_PolicySecret
175 #include "PolicySecret_fp.h"
176 #endif
177 #ifdef TPM_CC_PolicyTicket
178 #include "PolicyTicket_fp.h"
179 #endif
180 #ifdef TPM_CC_PolicyOR
181 #include "PolicyOR_fp.h"
182 #endif
183 #ifdef TPM_CC_PolicyPCR
184 #include "PolicyPCR_fp.h"
185 #endif
186 #ifdef TPM_CC_PolicyLocality
187 #include "PolicyLocality_fp.h"
188 #endif
189 #ifdef TPM_CC_PolicyNV
190 #include "PolicyNV_fp.h"
191 #endif
192 #ifdef TPM_CC_PolicyCounterTimer
193 #include "PolicyCounterTimer_fp.h"
194 #endif
195 #ifdef TPM_CC_PolicyCommandCode
196 #include "PolicyCommandCode_fp.h"
197 #endif
198 #ifdef TPM_CC_PolicyPhysicalPresence
199 #include "PolicyPhysicalPresence_fp.h"
200 #endif
201 #ifdef TPM_CC_PolicyCpHash
202 #include "PolicyCpHash_fp.h"
203 #endif
204 #ifdef TPM_CC_PolicyNameHash
205 #include "PolicyNameHash_fp.h"
206 #endif
207 #ifdef TPM_CC_PolicyDuplicationSelect
208 #include "PolicyDuplicationSelect_fp.h"
209 #endif
```



```
210 #ifdef TPM_CC_PolicyAuthorize
211 #include "PolicyAuthorize_fp.h"
212 #endif
213 #ifdef TPM_CC_PolicyAuthValue
214 #include "PolicyAuthValue_fp.h"
215 #endif
216 #ifdef TPM_CC_PolicyPassword
217 #include "PolicyPassword_fp.h"
218 #endif
219 #ifdef TPM_CC_PolicyGetDigest
220 #include "PolicyGetDigest_fp.h"
221 #endif
222 #ifdef TPM_CC_PolicyNvWritten
223 #include "PolicyNvWritten_fp.h"
224 #endif
225 #ifdef TPM_CC_PolicyTemplate
226 #include "PolicyTemplate_fp.h"
227 #endif
228 #ifdef TPM_CC_PolicyAuthorizeNV
229 #include "PolicyAuthorizeNV_fp.h"
230 #endif
```

Hierarchy Commands

```
231 #ifdef TPM_CC_CreatePrimary
232 #include "CreatePrimary_fp.h"
233 #endif
234 #ifdef TPM_CC_HierarchyControl
235 #include "HierarchyControl_fp.h"
236 #endif
237 #ifdef TPM_CC_SetPrimaryPolicy
238 #include "SetPrimaryPolicy_fp.h"
239 #endif
240 #ifdef TPM_CC_ChangePPS
241 #include "ChangePPS_fp.h"
242 #endif
243 #ifdef TPM_CC_ChangeEPS
244 #include "ChangeEPS_fp.h"
245 #endif
246 #ifdef TPM_CC_Clear
247 #include "Clear_fp.h"
248 #endif
249 #ifdef TPM_CC_ClearControl
250 #include "ClearControl_fp.h"
251 #endif
252 #ifdef TPM_CC_HierarchyChangeAuth
253 #include "HierarchyChangeAuth_fp.h"
254 #endif
```

Dictionary Attack Functions

```
255 #ifdef TPM_CC_DictionaryAttackLockReset
256 #include "DictionaryAttackLockReset_fp.h"
257 #endif
258 #ifdef TPM_CC_DictionaryAttackParameters
259 #include "DictionaryAttackParameters_fp.h"
260 #endif
```

Miscellaneous Management Functions

```
261 #ifdef TPM_CC_PP_Commands
262 #include "PP_Commands_fp.h"
263 #endif
264 #ifdef TPM_CC_SetAlgorithmSet
```

```
265 #include "SetAlgorithmSet_fp.h"
266 #endif
```

Field Upgrade

```
267 #ifdef TPM_CC_FieldUpgradeStart
268 #include "FieldUpgradeStart_fp.h"
269 #endif
270 #ifdef TPM_CC_FieldUpgradeData
271 #include "FieldUpgradeData_fp.h"
272 #endif
273 #ifdef TPM_CC_FirmwareRead
274 #include "FirmwareRead_fp.h"
275 #endif
```

Context Management

```
276 #ifdef TPM_CC_ContextSave
277 #include "ContextSave_fp.h"
278 #endif
279 #ifdef TPM_CC_ContextLoad
280 #include "ContextLoad_fp.h"
281 #endif
282 #ifdef TPM_CC_FlushContext
283 #include "FlushContext_fp.h"
284 #endif
285 #ifdef TPM_CC_EvictControl
286 #include "EvictControl_fp.h"
287 #endif
```

Clocks and Timers

```
288 #ifdef TPM_CC_ReadClock
289 #include "ReadClock_fp.h"
290 #endif
291 #ifdef TPM_CC_ClockSet
292 #include "ClockSet_fp.h"
293 #endif
294 #ifdef TPM_CC_ClockRateAdjust
295 #include "ClockRateAdjust_fp.h"
296 #endif
```

Capability Commands

```
297 #ifdef TPM_CC_GetCapability
298 #include "GetCapability_fp.h"
299 #endif
300 #ifdef TPM_CC_TestParms
301 #include "TestParms_fp.h"
302 #endif
```

Non-volatile Storage

```
303 #ifdef TPM_CC_NV_DefineSpace
304 #include "NV_DefineSpace_fp.h"
305 #endif
306 #ifdef TPM_CC_NV_UndefineSpace
307 #include "NV_UndefineSpace_fp.h"
308 #endif
309 #ifdef TPM_CC_NV_UndefineSpaceSpecial
310 #include "NV_UndefineSpaceSpecial_fp.h"
311 #endif
312 #ifdef TPM_CC_NV_ReadPublic
```



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

Attached Components

```
345 #ifdef TPM_CC_AC_GetCapability
346 #include "AC_GetCapability_fp.h"
347 #endif
348 #ifdef TPM_CC_AC_Send
349 #include "AC_Send_fp.h"
350 #endif
351 #ifdef TPM_CC_Policy_AC_SendSelect
352 #include "Policy_AC_SendSelect_fp.h"
353 #endif
```

Vendor Specific

```
354 #ifdef TPM_CC_Vendor_TCG_Test
355 #include "Vendor_TCG_Test_fp.h"
356 #endif
357 #endif
```

5.8 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.

```
1   #ifndef _COMPILER_DEPENDENCIES_H_
2   #define _COMPILER_DEPENDENCIES_H_
3   #ifdef GCC
4   #   undef _MSC_VER
5   #   undef WIN32
6   #endif
7   #ifdef _MSC_VER
```

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

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

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

```
11  //#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.

```
12  #   define LIB_EXPORT __declspec(dllexport) // VS compatible version
13  #   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.

```
14  #   define NORETURN   __declspec(noreturn)
15  #   if _MSC_VER >= 1400     // SAL processing when needed
16  #       include <sal.h>
17  #   endif
18  #   ifdef _WIN64
19  #       define _INTPTR 2
20  #    else
21  #       define _INTPTR 1
22  #    endif
23  #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

```
24  #   define _REDUCE_WARNING_LEVEL_(n)                    \
25  __pragma(warning(push, n))
```

Restore the compiler warning level

```
26  #   define _NORMAL_WARNING_LEVEL_                       \
27  __pragma(warning(pop))
28  #   include <stdint.h>
29  #endif
30  #ifndef _MSC_VER
31  #ifndef WINAPI
32  #   define WINAPI
33  #endif
34  #   define __pragma(x)
35  #   define REVERSE_ENDIAN_16(_Number) __builtin_bswap16(_Number)
```



```
36  #   define REVERSE_ENDIAN_32(_Number) __builtin_bswap32(_Number)
37  #   define REVERSE_ENDIAN_64(_Number) __builtin_bswap64(_Number)
38  #endif
39  #if defined(__GNUC__)
40  #   define NORETURN                    __attribute__((noreturn))
41  #   include <stdint.h>
42  #endif
```

Things that are not defined should be defined as NULL

```
43  #ifndef NORETURN
44  #   define NORETURN
45  #endif
46  #ifndef LIB_EXPORT
47  #   define LIB_EXPORT
48  #endif
49  #ifndef LIB_IMPORT
50  #   define LIB_IMPORT
51  #endif
52  #ifndef _REDUCE_WARNING_LEVEL_
53  #   define _REDUCE_WARNING_LEVEL_(n)
54  #endif
55  #ifndef _NORMAL_WARNING_LEVEL_
56  #   define _NORMAL_WARNING_LEVEL_
57  #endif
58  #ifndef NOT_REFERENCED
59  #   define  NOT_REFERENCED(x)  (x = x)
60  #endif
61  #ifdef _POSIX_
62  typedef int SOCKET;
63  #endif
64  #endif // _COMPILER_DEPENDENCIES_H_
```

5.9 Global.h

5.9.1 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.

```
1   #if !defined _TPM_H_
2   #error "Should only be instanced in TPM.h"
3   #endif
```

5.9.2 Includes

```
4   #ifndef         GLOBAL_H
5   #define         GLOBAL_H
6   _REDUCE_WARNING_LEVEL_(2)
7   #include <string.h>
8   #include <stddef.h>
9   _NORMAL_WARNING_LEVEL_
10
11  #if SIMULATION
12  #undef CONTEXT_SLOT
13  #  define CONTEXT_SLOT    UINT8
14  #endif
15  #include "Capabilities.h"
16  #include "TpmTypes.h"
17  #include "CommandAttributes.h"
18  #include "CryptTest.h"
19  #include "BnValues.h"
20  #include "CryptHash.h"
21  #include "CryptSym.h"
22  #include "CryptRand.h"
23  #include "CryptEcc.h"
24  #include "CryptRsa.h"
25  #include "CryptTest.h"
26  #include "TpmError.h"
27  #include "NV.h"
28
29  //** Defines and Types
30
31  //*** Crypto Self-Test Values
32  extern ALGORITHM_VECTOR     g_implementedAlgorithms;
33  extern ALGORITHM_VECTOR     g_toTest;
34
35  //*** Size Types
36  // These types are used to differentiate the two different size values used.
37  //
38  // NUMBYTES is used when a size is a number of bytes (usually a TPM2B)
39  typedef UINT16  NUMBYTES;
40
41  //*** Other Types
42  // An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)
43  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

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

Definition for a PROOF value

```
TPM2B_TYPE(PROOF, PROOF_SIZE);
```

Definition for a Primary Seed value

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

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.

```
#if CLOCK_STOPS
typedef UINT64          CLOCK_NONCE;
#else
typedef UINT32          CLOCK_NONCE;
#endif
```

5.9.3 Loaded Object Structures

5.9.3.1 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.

5.9.3.2 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.

```
typedef struct
{
    unsigned            publicOnly : 1;     //0) SET if only the public portion of
                                            //   an object is loaded
    unsigned            epsHierarchy : 1;   //1) SET if the object belongs to EPS
                                            //   Hierarchy
    unsigned            ppsHierarchy : 1;   //2) SET if the object belongs to PPS
                                            //   Hierarchy
    unsigned            spsHierarchy : 1;   //3) SET f the object belongs to SPS
                                            //   Hierarchy
    unsigned            evict : 1;          //4) SET if the object is a platform or
                                            //   owner evict object.  Platform-
                                            //   evict object belongs to PPS
                                            //   hierarchy, owner-evict object
                                            //   belongs to SPS or EPS hierarchy.
                                            //   This bit is also used to mark a
                                            //   completed sequence object so it
```

```
                                                //   will be flush when the
                                                //   SequenceComplete command succeeds.
    unsigned            primary : 1;            //5) SET for a primary object
    unsigned            temporary : 1;          //6) SET for a temporary object
    unsigned            stClear : 1;            //7) SET for an stClear object
    unsigned            hmacSeq : 1;            //8) SET for an HMAC or MAC sequence
                                                //   object
    unsigned            hashSeq : 1;            //9) SET for a hash sequence object
    unsigned            eventSeq : 1;           //10) SET for an event sequence object
    unsigned            ticketSafe : 1;         //11) SET if a ticket is safe to create
                                                //    for hash sequence object
    unsigned            firstBlock : 1;         //12) SET if the first block of hash
                                                //    data has been received.  It
                                                //    works with ticketSafe bit
    unsigned            isParent : 1;           //13) SET if the key has the proper
                                                //    attributes to be a parent key
    unsigned            privateExp : 1;         //14) SET when the private exponent
                                                //    of an RSA key has been validated.
    unsigned            occupied : 1;           //15) SET when the slot is occupied.
    unsigned            derivation : 1;         //16) SET when the key is a derivation
                                                //        parent
    unsigned            external : 1;           //17) SET when the object is loaded with
                                                //    TPM2_LoadExternal();
} OBJECT_ATTRIBUTES;
```

5.9.3.3 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.

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

#if     ALG_RSA
    privateExponent_t   privateExponent;    // Additional field for the private
#endif
    TPM2B_NAME          qualifiedName;      // object qualified name
    TPMI_DH_OBJECT      evictHandle;        // if the object is an evict object,
                                            // the original handle is kept here.
                                            // The 'working' handle will be the
                                            // handle of an object slot.

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

5.9.3.4 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

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

    // The data below is unique to a sequence object
    TPM2B_AUTH          auth;               // authorization for use of sequence
    union
    {
        HASH_STATE      hashState[HASH_COUNT];
        HMAC_STATE      hmacState;
    }                   state;
} HASH_OBJECT;
typedef BYTE  HASH_OBJECT_BUFFER[sizeof(HASH_OBJECT)];
```

5.9.3.5 ANY_OBJECT

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

```
typedef union ANY_OBJECT
{
    OBJECT              entity;
    HASH_OBJECT         hash;
} ANY_OBJECT;
typedef BYTE    ANY_OBJECT_BUFFER[sizeof(ANY_OBJECT)];
```

5.9.4 AUTH_DUP Types

These values are used in the authorization processing.

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

5.9.5 Active Session Context

5.9.5.1 Description

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

5.9.5.2 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.

```
typedef struct SESSION_ATTRIBUTES
{
    unsigned    isPolicy : 1;           //1) SET if the session may only be used
                                        //   for policy
    unsigned    isAudit : 1;            //2) SET if the session is used for audit
    unsigned    isBound : 1;            //3) SET if the session is bound to with an
```

```
                                        //   entity. This attribute will be CLEAR
                                        //   if either isPolicy or isAudit is SET.
    unsigned    isCpHashDefined : 1;    //3) SET if the cpHash has been defined
                                        //   This attribute is not SET unless
                                        //   'isPolicy' is SET.
    unsigned    isAuthValueNeeded : 1;  //5) SET if the authValue is required for
                                        //   computing the session HMAC. This
                                        //   attribute is not SET unless 'isPolicy'
                                        //   is SET.
    unsigned    isPasswordNeeded : 1;   //6) SET if a password authValue is required
                                        //   for authorization This attribute is not
                                        //   SET unless 'isPolicy' is SET.
    unsigned    isPPRequired : 1;       //7) SET if physical presence is required to
                                        //   be asserted when the authorization is
                                        //   checked. This attribute is not SET
                                        //   unless 'isPolicy' is SET.
    unsigned    isTrialPolicy : 1;      //8) SET if the policy session is created
                                        //   for trial of the policy's policyHash
                                        //   generation. This attribute is not SET
                                        //   unless 'isPolicy' is SET.
    unsigned    isDaBound : 1;          //9) SET if the bind entity had noDA CLEAR.
                                        //   If this is SET, then an authorization
                                        //   failure using this session will count
                                        //   against lockout even if the object
                                        //   being authorized is exempt from DA.
    unsigned    isLockoutBound : 1;     //10) SET if the session is bound to
                                        //    lockoutAuth.
    unsigned    includeAuth : 1;        //11) This attribute is SET when the
                                        //    authValue of an object is to be
                                        //    included in the computation of the
                                        //    HMAC key for the command and response
                                        //    computations. (was 'requestWasBound')
    unsigned    checkNvWritten : 1;     //12) SET if the TPMA_NV_WRITTEN attribute
                                        //    needs to be checked when the policy is
                                        //    used for authorization for NV access.
                                        //    If this is SET for any other type, the
                                        //    policy will fail.
    unsigned    nvWrittenState : 1;     //13) SET if TPMA_NV_WRITTEN is required to
                                        //    be SET. Used when 'checkNvWritten' is
                                        //    SET
    unsigned    isTemplateSet : 1;      //14) SET if the templateHash needs to be
                                        //    checked for Create, CreatePrimary, or
                                        //    CreateLoaded.
} SESSION_ATTRIBUTES;
```

5.9.5.3 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.

```
typedef struct SESSION
{
    SESSION_ATTRIBUTES  attributes;         // session attributes
    UINT32              pcrCounter;         // PCR counter value when PCR is
                                            // included (policy session)
                                            // If no PCR is included, this
                                            // value is 0.
    UINT64              startTime;          // The value in g_time when the session
                                            // was started (policy session)
    UINT64              timeout;            // The timeout relative to g_time
                                            // There is no timeout if this value
                                            // is 0.
    CLOCK_NONCE         epoch;              // The g_clockEpoch value when the
```



```
                                                // session was started. If g_clockEpoch
                                                // does not match this value when the
                                                // timeout is used, then
                                                // then the command will fail.
    TPM_CC              commandCode;            // command code (policy session)
    TPM_ALG_ID          authHashAlg;            // session hash algorithm
    TPMA_LOCALITY       commandLocality;        // command locality (policy session)
    TPMT_SYM_DEF        symmetric;              // session symmetric algorithm (if any)
    TPM2B_AUTH          sessionKey;             // session secret value used for
                                                // this session
    TPM2B_NONCE         nonceTPM;               // last TPM-generated nonce for
                                                // generating HMAC and encryption keys
   union
    {
        TPM2B_NAME      boundEntity;            // value used to track the entity to
                                                // which the session is bound

        TPM2B_DIGEST    cpHash;                 // the required cpHash value for the
                                                // command being authorized
        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)];
```

5.9.6 PCR

5.9.6.1 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
{
#if     ALG_SHA1
    BYTE                sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
#endif
#if     ALG_SHA256
    BYTE                sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
#endif
#if     ALG_SHA384
    BYTE                sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
#endif
#if     ALG_SHA512
    BYTE                sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
#endif
#if     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;
```

5.9.6.2 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];
} PCR_POLICY;
#endif
```

5.9.6.3 PCR_AUTHVALUE

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

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

5.9.7 STARTUP_TYPE

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

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

5.9.8 NV

5.9.8.1 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.

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


5.9.8.2 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.

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

5.9.8.3 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.

```
#if BIG_ENDIAN_TPM
typedef struct
{
    UINT32      pinCount;
    UINT32      pinLimit;
} PIN_DATA;
#else
typedef struct
{
    UINT32      pinLimit;
    UINT32      pinCount;
} PIN_DATA;
#endif
typedef union
{
    UINT64     intVal;
    PIN_DATA   pin;
} NV_PIN;
```

5.9.9 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.

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

5.9.10 RAM Global Values

5.9.10.1 Description

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

5.9.10.2 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.

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

5.9.10.3 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.

```
extern TPM_HANDLE        g_exclusiveAuditSession;
```

5.9.10.4 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 the accumulated time since the last time that the TPM's timer was last powered up. Clock is the accumulated time since the last time that the TPM was cleared. g_time is in *mS*.

```
extern  UINT64           g_time;
```

5.9.10.5 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.

```
#if CLOCK_STOPS
extern CLOCK_NONCE       g_timeEpoch;
#else
#define g_timeEpoch       gp.timeEpoch
#endif
```

5.9.10.6 g_phEnable

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

```
extern BOOL              g_phEnable;
```

5.9.10.7 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).

```
extern BOOL              g_pcrReConfig;
```

5.9.10.8 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().


```
extern TPMI_DH_OBJECT   g_DRTMHandle;
```

5.9.10.9 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.

```
extern  BOOL            g_DrtmPreStartup;
```

5.9.10.10 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.

```
extern  BOOL            g_StartupLocality3;
```

5.9.10.11 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.

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

5.9.10.12 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.

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

5.9.10.13 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).

```
#define PRE_STARTUP_FLAG     0x8000
#define STARTUP_LOCALITY_3   0x4000
#if USE_DA_USED
```

5.9.10.14 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

```
extern  BOOL                g_daUsed;
```

```
330 #endif
```

5.9.10.15 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.

```
331 typedef BYTE        UPDATE_TYPE;
332 #define UT_NONE     (UPDATE_TYPE)0
333 #define UT_NV       (UPDATE_TYPE)1
334 #define UT_ORDERLY  (UPDATE_TYPE)(UT_NV + 2)
335 extern UPDATE_TYPE          g_updateNV;
```

5.9.10.16 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.

```
336 extern BOOL             g_powerWasLost;
```

5.9.10.17 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.

```
337 extern BOOL             g_clearOrderly;
```

5.9.10.18 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.

```
338 extern TPM_SU           g_prevOrderlyState;
```

5.9.10.19 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.

```
339 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

```
extern TPM_RC           g_NvStatus;
```

5.9.10.20 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.

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

5.9.11 Persistent Global Values

5.9.11.1 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.

5.9.11.2 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().

```
typedef struct
{
//*****************************************************************************
//          Hierarchy
//*****************************************************************************
// The values in this section are related to the hierarchies.

    BOOL                disableClear;       // TRUE if TPM2_Clear() using
                                            // lockoutAuth is disabled

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

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

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

```

```
372         // Hierarchy proofs
373         TPM2B_PROOF          phProof;
374         TPM2B_PROOF          shProof;
375         TPM2B_PROOF          ehProof;
376         // Note there is a nullProof in the state_reset memory.
377
378     //*****************************************************************************
379     //          Reset Events
380     //*****************************************************************************
381     // A count that increments at each TPM reset and never get reset during the life
382     // time of TPM.  The value of this counter is initialized to 1 during TPM
383     // manufacture process. It is used to invalidate all saved contexts after a TPM
384     // Reset.
385         UINT64              totalResetCount;
386
387     // This counter increments on each TPM Reset. The counter is reset by
388     // TPM2_Clear().
389         UINT32              resetCount;
390
391     //*****************************************************************************
392     //          PCR
393     //*****************************************************************************
394     // This structure hold the policies for those PCR that have an update policy.
395     // This implementation only supports a single group of PCR controlled by
396     // policy. If more are required, then this structure would be changed to
397     // an array.
398     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
399         PCR_POLICY          pcrPolicies;
400     #endif
401
402     // This structure indicates the allocation of PCR. The structure contains a
403     // list of PCR allocations for each implemented algorithm. If no PCR are
404     // allocated for an algorithm, a list entry still exists but the bit map
405     // will contain no SET bits.
406         TPML_PCR_SELECTION  pcrAllocated;
407
408     //*****************************************************************************
409     //          Physical Presence
410     //*****************************************************************************
411     // The PP_LIST type contains a bit map of the commands that require physical
412     // to be asserted when the authorization is evaluated. Physical presence will be
413     // checked if the corresponding bit in the array is SET and if the authorization
414     // handle is TPM_RH_PLATFORM.
415     //
416     // These bits may be changed with TPM2_PP_Commands().
417         BYTE                ppList[(COMMAND_COUNT + 7) / 8];
418
419     //*****************************************************************************
420     //          Dictionary attack values
421     //*****************************************************************************
422     // These values are used for dictionary attack tracking and control.
423         UINT32              failedTries;        // the current count of unexpired
424                                                 // authorization failures
425
426         UINT32              maxTries;           // number of unexpired authorization
427                                                 // failures before the TPM is in
428                                                 // lockout
429
430         UINT32              recoveryTime;       // time between authorization failures
431                                                 // before failedTries is decremented
432
433         UINT32              lockoutRecovery;    // time that must expire between
434                                                 // authorization failures associated
435                                                 // with lockoutAuth
436
437         BOOL                lockOutAuthEnabled; // TRUE if use of lockoutAuth is
```



```
                                                // allowed

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

//*****************************************************************************
//           Command audit values.
//*****************************************************************************
    BYTE                auditCommands[((COMMAND_COUNT + 1) + 7) / 8];
    TPMI_ALG_HASH       auditHashAlg;
    UINT64              auditCounter;

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

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

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

// g_firmwareV1 contains the less significant 32-bits of the vendor version number.
// In the reference implementation, if this value is printed as a hex
// value, it will have the format of 00 HH MM SS
    UINT32              firmwareV2;
//*****************************************************************************
//           Timer Epoch
//*****************************************************************************
// timeEpoch contains a nonce that has a vendor=specific size (should not be
// less than 8 bytes. This nonce changes when the clock epoch changes. The clock
// epoch changes when there is a discontinuity in the timing of the TPM.
#if !CLOCK_STOPS
    CLOCK_NONCE         timeEpoch;
#endif

} PERSISTENT_DATA;
extern PERSISTENT_DATA  gp;
```

5.9.11.3 ORDERLY_DATA

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

```
typedef struct orderly_data
{
//*****************************************************************************
//           TIME
//*****************************************************************************
```

```
497
498  // Clock has two parts. One is the state save part and one is the NV part. The
499  // state save version is updated on each command. When the clock rolls over, the
500  // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
501  // orderly way, then the sClock value is used to initialize the clock. If the
502  // TPM shutdown was not orderly, then the persistent value is used and the safe
503  // attribute is clear.
504
505      UINT64              clock;              // The orderly version of clock
506      TPMI_YES_NO         clockSafe;          // Indicates if the clock value is
507                                              // safe.
508
509      // In many implementations, the quality of the entropy available is not that
510      // high. To compensate, the current value of the drbgState can be saved and
511      // restored on each power cycle. This prevents the internal state from reverting
512      // to the initial state on each power cycle and starting with a limited amount
513      // of entropy. By keeping the old state and adding entropy, the entropy will
514      // accumulate.
515      DRBG_STATE          drbgState;
516
517  // These values allow the accumulation of self-healing time across orderly shutdown
518  // of the TPM.
519  #if ACCUMULATE_SELF_HEAL_TIMER
520      UINT64              selfHealTimer;  // current value of s_selfHealTimer
521      UINT64              lockoutTimer;   // current value of s_lockoutTimer
522      UINT64              time;           // current value of g_time at shutdown
523  #endif // ACCUMULATE_SELF_HEAL_TIMER
524
525  } ORDERLY_DATA;
526  #if ACCUMULATE_SELF_HEAL_TIMER
527  #define     s_selfHealTimer     go.selfHealTimer
528  #define     s_lockoutTimer      go.lockoutTimer
529  #endif  // ACCUMULATE_SELF_HEAL_TIMER
530  #  define drbgDefault go.drbgState
531  extern ORDERLY_DATA     go;
```

5.9.11.4 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).

```
532  typedef struct state_clear_data
533  {
534  //*****************************************************************************
535  //            Hierarchy Control
536  //*****************************************************************************
537      BOOL                shEnable;           // default reset is SET
538      BOOL                ehEnable;           // default reset is SET
539      BOOL                phEnableNV;         // default reset is SET
540      TPMI_ALG_HASH       platformAlg;        // default reset is TPM_ALG_NULL
541      TPM2B_DIGEST        platformPolicy;     // default reset is an Empty Buffer
542      TPM2B_AUTH          platformAuth;       // default reset is an Empty Buffer
543
544  //*****************************************************************************
545  //            PCR
546  //*****************************************************************************
547  // The set of PCR to be saved on Shutdown(STATE)
548      PCR_SAVE            pcrSave;            // default reset is 0...0
549
550  // This structure hold the authorization values for those PCR that have an
```



```
// update authorization.
// This implementation only supports a single group of PCR controlled by
// authorization. If more are required, then this structure would be changed to
// an array.
    PCR_AUTHVALUE       pcrAuthValues;
} STATE_CLEAR_DATA;
extern STATE_CLEAR_DATA gc;
```

5.9.11.5 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.

```
typedef struct state_reset_data
{
//*****************************************************************************
//          Hierarchy Control
//*****************************************************************************
    TPM2B_PROOF         nullProof;          // The proof value associated with
                                            // the TPM_RH_NULL hierarchy. The
                                            // default reset value is from the RNG.

    TPM2B_SEED          nullSeed;           // The seed value for the TPM_RN_NULL
                                            // hierarchy. The default reset value
                                            // is from the RNG.

//*****************************************************************************
//           Context
//*****************************************************************************
// The 'clearCount' counter is incremented each time the TPM successfully executes
// a TPM Resume. The counter is included in each saved context that has 'stClear'
// SET (including descendants of keys that have 'stClear' SET). This prevents these
// objects from being loaded after a TPM Resume.
// If 'clearCount' is at its maximum value when the TPM receives a Shutdown(STATE),
// the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
    UINT32              clearCount;         // The default reset value is 0.

    UINT64              objectContextID;    // This is the context ID for a saved
                                            //  object context. The default reset
                                            //  value is 0.
#ifndef NDEBUG
#undef  CONTEXT_SLOT
#define CONTEXT_SLOT     BYTE
#endif

    CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];    // This array contains
                                            // contains the values used to track
                                            // the version numbers of saved
                                            // contexts (see
                                            // Session.c in for details). The
                                            // default reset value is {0}.

    CONTEXT_COUNTER     contextCounter;     // This is the value from which the
                                            // 'contextID' is derived. The
                                            // default reset value is {0}.

//*****************************************************************************
//           Command Audit
//*****************************************************************************
// When an audited command completes, ExecuteCommand() checks the return
// value.  If it is TPM_RC_SUCCESS, and the command is an audited command, the
// TPM will extend the cpHash and rpHash for the command to this value. If this
// digest was the Zero Digest before the cpHash was extended, the audit counter
```

```
608  // is incremented.
609
610      TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest
611                                              // by TPM2_GetCommandAuditDigest() or a
612                                              // TPM Reset.
613
614  //*************************************************************************
615  //           Boot counter
616  //*************************************************************************
617
618      UINT32              restartCount;       // This counter counts TPM Restarts.
619                                              // The default reset value is 0.
620
621  //**************************************************************************
622  //            PCR
623  //**************************************************************************
624  // This counter increments whenever the PCR are updated. This counter is preserved
625  // across TPM Resume even though the PCR are not preserved. This is because
626  // sessions remain active across TPM Restart and the count value in the session
627  // is compared to this counter so this counter must have values that are unique
628  // as long as the sessions are active.
629  // NOTE: A platform-specific specification may designate that certain PCR changes
630  //       do not increment this counter to increment.
631      UINT32              pcrCounter;         // The default reset value is 0.
632
633  #if     ALG_ECC
634
635  //*************************************************************************
636  //         ECDAA
637  //*************************************************************************
638      UINT64              commitCounter;      // This counter increments each time
639                                              // TPM2_Commit() returns
640                                              // TPM_RC_SUCCESS. The default reset
641                                              // value is 0.
642
643      TPM2B_NONCE         commitNonce;        // This random value is used to compute
644                                              // the commit values. The default reset
645                                              // value is from the RNG.
646
647  // This implementation relies on the number of bits in g_commitArray being a
648  // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
649      BYTE                commitArray[16];    // The default reset value is {0}.
650
651  #endif // ALG_ECC
652  } STATE_RESET_DATA;
653  extern STATE_RESET_DATA gr;
```

5.9.12 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

```
654  #define NV_PERSISTENT_DATA  (0)
655  #define NV_STATE_RESET_DATA (NV_PERSISTENT_DATA + sizeof(PERSISTENT_DATA))
656  #define NV_STATE_CLEAR_DATA (NV_STATE_RESET_DATA + sizeof(STATE_RESET_DATA))
657  #define NV_ORDERLY_DATA     (NV_STATE_CLEAR_DATA + sizeof(STATE_CLEAR_DATA))
658  #define NV_INDEX_RAM_DATA   (NV_ORDERLY_DATA + sizeof(ORDERLY_DATA))
```



```
659  #define NV_USER_DYNAMIC     (NV_INDEX_RAM_DATA + sizeof(s_indexOrderlyRam))
660  #define NV_USER_DYNAMIC_END     NV_MEMORY_SIZE
```

5.9.13 Global Macro Definitions

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

```
661  #define NV_READ_PERSISTENT(to, from)      \
662              NvRead(&to, offsetof(PERSISTENT_DATA, from), sizeof(to))
663  #define NV_WRITE_PERSISTENT(to, from)     \
664              NvWrite(offsetof(PERSISTENT_DATA, to), sizeof(gp.to), &from)
665  #define CLEAR_PERSISTENT(item)                          \
666      NvClearPersistent(offsetof(PERSISTENT_DATA, item), sizeof(gp.item))
667  #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().

```
668  typedef UINT16      COMMAND_INDEX;
669  #define UNIMPLEMENTED_COMMAND_INDEX     ((COMMAND_INDEX)(~0))
670  typedef struct _COMMAND_FLAGS_
671  {
672      unsigned    trialPolicy : 1;    //1) If SET, one of the handles references a
673                                      //   trial policy and authorization may be
674                                      //   skipped. This is only allowed for a policy
675                                      //   command.
676  } 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.

```
677  typedef struct _COMMAND_
678  {
679      TPM_ST          tag;                // the parsed command tag
680      TPM_CC          code;               // the parsed command code
681      COMMAND_INDEX   index;              // the computed command index
682      UINT32          handleNum;          // the number of entity handles in the
683                                          //   handle area of the command
684      TPM_HANDLE      handles[MAX_HANDLE_NUM]; // the parsed handle values
685      UINT32          sessionNum;         // the number of sessions found
686      INT32           parameterSize;      // starts out with the parsed command size
687                                          // and is reduced and values are
688                                          // unmarshaled. Just before calling the
689                                          // command actions, this should be zero.
690                                          // After the command actions, this number
691                                          // should grow as values are marshaled
692                                          // in to the response buffer.
693      INT32           authSize;           // this is initialized with the parsed size
694                                          // of authorizationSize field and should
695                                          // be zero when the authorizations are
696                                          // parsed.
697      BYTE           *parameterBuffer;    // input to ExecuteCommand
698      BYTE           *responseBuffer;     // input to ExecuteCommand
699  #if ALG_SHA1
700      TPM2B_SHA1_DIGEST   sha1CpHash;
701      TPM2B_SHA1_DIGEST   sha1RpHash;
702  #endif
703  #if ALG_SHA256
704      TPM2B_SHA256_DIGEST sha256CpHash;
705      TPM2B_SHA256_DIGEST sha256RpHash;
706  #endif
```

```
#if ALG_SHA384
    TPM2B_SHA384_DIGEST sha384CpHash;
    TPM2B_SHA384_DIGEST sha384RpHash;
#endif
#if ALG_SHA512
    TPM2B_SHA512_DIGEST sha512CpHash;
    TPM2B_SHA512_DIGEST sha512RpHash;
#endif
#if ALG_SM3_256
    TPM2B_SM3_256_DIGEST sm3_256CpHash;
    TPM2B_SM3_256_DIGEST sm3_256RpHash;
#endif
} COMMAND;
```

Global sting constants for consistency in KDF function calls.

```
extern const TPM2B      *PRIMARY_OBJECT_CREATION;
extern const TPM2B      *SECRET_KEY;
extern const TPM2B      *SESSION_KEY;
extern const TPM2B      *STORAGE_KEY;
extern const TPM2B      *INTEGRITY_KEY;
extern const TPM2B      *CONTEXT_KEY;
extern const TPM2B      *CFB_KEY;
extern const TPM2B      *XOR_KEY;
extern const TPM2B      *DUPLICATE_STRING;
extern const TPM2B      *OBFUSCATE_STRING;
extern const TPM2B      *IDENTITY_STRING;
extern const TPM2B      *COMMIT_STRING;
#if SELF_TEST
extern const TPM2B      *OAEP_TEST_STRING;
#endif // SELF_TEST
```

From Manufacture.c

```
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.

```
extern BOOL             g_initialized;
```

5.9.14 Private data

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

From SessionProcess.c

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

Array of the authorization session handles

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

Array of authorization session attributes

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


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

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

Array of nonces provided by the caller for the corresponding sessions

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

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

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

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

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

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter

```
extern UINT32            s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
extern UINT32            s_decryptSessionIndex;
```

Index of a session used for audit

```
extern UINT32            s_auditSessionIndex;
```

The *cpHash* for command audit

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

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

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

From DA.c

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

```
#if !ACCUMULATE_SELF_HEAL_TIMER
extern UINT64        s_selfHealTimer;
```

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

```
extern UINT64        s_lockoutTimer;
#endif // ACCUMULATE_SELF_HEAL_TIMER
#endif // DA_C
#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.

```
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.

```
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.

```
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.

```
extern      NV_INDEX          s_cachedNvIndex;
extern      NV_REF            s_cachedNvRef;
extern      BYTE             *s_cachedNvRamRef;
```

Initial NV Index/evict object iterator value

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

From Object.c

This type is the container for an object.

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

From PCR.c

```
typedef struct
{
#if     ALG_SHA1
    // SHA1 PCR
    BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
#endif
#if     ALG_SHA256
    // SHA256 PCR
    BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
#endif
#if     ALG_SHA384
    // SHA384 PCR
    BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
#endif
#if     ALG_SHA512
```



```
787         // SHA512 PCR
788         BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
789     #endif
790     #if     ALG_SM3_256
791         // SHA256 PCR
792         BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
793     #endif
794     } PCR;
795     typedef struct
796     {
797         unsigned int    stateSave : 1;              // if the PCR value should be
798                                                     // saved in state save
799         unsigned int    resetLocality : 5;          // The locality that the PCR
800                                                     // can be reset
801         unsigned int    extendLocality : 5;         // The locality that the PCR
802                                                     // can be extend
803     } PCR_Attributes;
804     extern PCR          s_pcrs[IMPLEMENTATION_PCR];
805     #endif // PCR_C
806     #if defined SESSION_C || defined GLOBAL_C
```

From Session.c

Container for HMAC or policy session tracking information

```
807     typedef struct
808     {
809         BOOL                occupied;
810         SESSION             session;        // session structure
811     } SESSION_SLOT;
812     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.

```
813     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)

```
814     extern int              s_freeSessionSlots;
815     #endif // SESSION_C
816     #if defined IO_BUFFER_C || defined GLOBAL_C
```

Each command function is allowed a structure for the inputs to the function and a structure for the outputs. The command dispatch code unmarshals the input butter to the command action input structure starting at the first byte of *s_actionIoBuffer*. The value of *s_actionIoAllocation* is the number of UINT64 values allocated. It is used to set the pointer for the response structure. The command dispatch code will marshal the response values into the final output buffer.

```
817     extern UINT64   s_actionIoBuffer[768];      // action I/O buffer
818     extern UINT32   s_actionIoAllocation;       // number of UIN64 allocated for the
819                                                 // action input structure
820     #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.

```
821  extern BOOL      g_inFailureMode;       // Indicates that the TPM is in failure mode
822  #if SIMULATION
823  extern BOOL      g_forceFailureMode;    // flag to force failure mode during test
824  #endif
825  typedef void(FailFunction)(const char *function, int line, int code);
826  #if defined TPM_FAIL_C || defined GLOBAL_C || 1
827  extern UINT32    s_failFunction;
828  extern UINT32    s_failLine;            // the line in the file at which
829                                          // the error was signaled
830  extern UINT32    s_failCode;            // the error code used
831  extern FailFunction    *LibFailCallback;
832  #endif // TPM_FAIL_C
```

From CommandCodeAttributes.c

```
833  extern  const  TPMA_CC                s_ccAttr[];
834  extern  const  COMMAND_ATTRIBUTES     s_commandAttributes[];
835  #endif // GLOBAL_H
```

5.10 GpMacros.h

5.10.1 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"
```

5.10.2 For Self-test

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

```
#if 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
```

5.10.3 For Failures

```
#if defined _POSIX_
#   define FUNCTION_NAME        0
#else
#   define FUNCTION_NAME        __FUNCTION__
#endif
#if !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.

```
#define REQUIRE(condition, errorCode, returnCode)        \
    {                                                    \
        if(!!(condition))                                \
        {                                                \
            FAIL(FATAL_ERROR_errorCode);                 \
            FAIL_RETURN(returnCode);                     \
        }                                                \
    }
#define PARAMETER_CHECK(condition, returnCode)           \
    REQUIRE((condition), PARAMETER, returnCode)
#if (defined EMPTY_ASSERT) && (EMPTY_ASSERT != NO)
#   define pAssert(a)  ((void)0)
#else
#   define pAssert(a) {if(!(a)) FAIL(FATAL_ERROR_PARAMETER);}
#endif
```

5.10.4 Derived from Vendor-specific values

Values derived from vendor specific settings in Implementation.h

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

5.10.5 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 cryptographic 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

```
#if COMPILER_CHECKS
#   define  cAssert     pAssert
#else
#   define cAssert(value)
#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.

```
#define     ERROR_RETURN(returnCode)    \
    {                                   \
         retVal = returnCode;           \
         goto Exit;                     \
    }
#ifndef MAX
#  define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif
#ifndef MIN
#  define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif
#ifndef IsOdd
```



```
72  #  define IsOdd(a)        (((a) & 1) != 0)
73  #endif
74  #ifndef BITS_TO_BYTES
75  #  define BITS_TO_BYTES(bits) (((bits) + 7) >> 3)
76  #endif
```

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

```
77  #define TEST_BIT(bit, vector)   TestBit((bit), (BYTE *)&(vector), sizeof(vector))
78  #define SET_BIT(bit, vector)    SetBit((bit), (BYTE *)&(vector), sizeof(vector))
79  #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)

```
80  #ifndef LIB_EXPORT
81  #   define LIB_EXPORT
82  #   define LIB_IMPORT
83  #endif
84  #ifndef NORETURN
85  #   define NORETURN _Noreturn
86  #endif
87  #ifndef NOT_REFERENCED
88  #   define NOT_REFERENCED(x = x)    ((void) (x))
89  #endif
90  #define STD_RESPONSE_HEADER (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC))
91  #define JOIN(x,y) x##y
92  #define CONCAT(x,y) JOIN(x, y)
```

If CONTEXT_INTEGRITY_HASH_ALG is defined, then the vendor is using the old style table. Otherwise, pick the **strongest** implemented hash algorithm as the context hash.

```
93  #ifndef CONTEXT_HASH_ALGORITHM
94  #   if defined ALG_SHA512 && ALG_SHA512 == YES
95  #       define CONTEXT_HASH_ALGORITHM    SHA512
96  #   elif defined ALG_SHA384 && ALG_SHA384 == YES
97  #       define CONTEXT_HASH_ALGORITHM    SHA384
98  #   elif defined ALG_SHA256 && ALG_SHA256 == YES
99  #       define CONTEXT_HASH_ALGORITHM    SHA256
100 #   elif defined ALG_SM3_256 && ALG_SM3_256 == YES
101 #       define CONTEXT_HASH_ALGORITHM    SM3_256
102 #   elif defined ALG_SHA1 && ALG_SHA1 == YES
103 #       define CONTEXT_HASH_ALGORITHM  SHA1
104 #   endif
105 #   define CONTEXT_INTEGRITY_HASH_ALG  CONCAT(TPM_ALG_, CONTEXT_HASH_ALGORITHM)
106 #endif
107 #ifndef CONTEXT_INTEGRITY_HASH_SIZE
108 #define CONTEXT_INTEGRITY_HASH_SIZE CONCAT(CONTEXT_HASH_ALGORITHM, _DIGEST_SIZE)
109 #endif
110 #if     ALG_RSA
111 #define     RSA_SECURITY_STRENGTH (MAX_RSA_KEY_BITS >= 15360 ? 256 :          \
112                                    (MAX_RSA_KEY_BITS >=  7680 ? 192 :          \
113                                    (MAX_RSA_KEY_BITS >=  3072 ? 128 :          \
114                                    (MAX_RSA_KEY_BITS >=  2048 ? 112 :          \
115                                    (MAX_RSA_KEY_BITS >=  1024 ?  80 :  0)))))
116 #else
117 #define     RSA_SECURITY_STRENGTH   0
118 #endif // ALG_RSA
119 #if     ALG_ECC
120 #define     ECC_SECURITY_STRENGTH (MAX_ECC_KEY_BITS >= 521 ? 256 :              \
121                                    (MAX_ECC_KEY_BITS >= 384 ? 192 :              \
122                                    (MAX_ECC_KEY_BITS >= 256 ? 128 : 0)))
123 #else
124 #define     ECC_SECURITY_STRENGTH   0
```

```
125  #endif // ALG_ECC
126  #define     MAX_ASYM_SECURITY_STRENGTH                                    \
127                      MAX(RSA_SECURITY_STRENGTH, ECC_SECURITY_STRENGTH)
128  #define     MAX_HASH_SECURITY_STRENGTH  ((CONTEXT_INTEGRITY_HASH_SIZE * 8) / 2)
```

Unless some algorithm is broken...

```
129  #define     MAX_SYM_SECURITY_STRENGTH   MAX_SYM_KEY_BITS
130  #define MAX_SECURITY_STRENGTH_BITS                                        \
131                      MAX(MAX_ASYM_SECURITY_STRENGTH,                       \
132                      MAX(MAX_SYM_SECURITY_STRENGTH,                        \
133                          MAX_HASH_SECURITY_STRENGTH))
```

This is the size that was used before the 1.38 errata requiring that P1.14.4 be followed

```
134  #define PROOF_SIZE      CONTEXT_INTEGRITY_HASH_SIZE
```

As required by P1.14.4

```
135  #define COMPLIANT_PROOF_SIZE                                              \
136              (MAX(CONTEXT_INTEGRITY_HASH_SIZE, (2 * MAX_SYM_KEY_BYTES)))
```

As required by P1.14.3.1

```
137  #define COMPLIANT_PRIMARY_SEED_SIZE                                       \
138      BITS_TO_BYTES(MAX_SECURITY_STRENGTH_BITS * 2)
```

This is the pre-errata version

```
139  #ifndef PRIMARY_SEED_SIZE
140  #   define PRIMARY_SEED_SIZE    PROOF_SIZE
141  #endif
142  #if USE_SPEC_COMPLIANT_PROOFS
143  #   undef PROOF_SIZE
144  #   define PROOF_SIZE           COMPLIANT_PROOF_SIZE
145  #   undef PRIMARY_SEED_SIZE
146  #   define PRIMARY_SEED_SIZE    COMPLIANT_PRIMARY_SEED_SIZE
147  #endif  // USE_SPEC_COMPLIANT_PROOFS
148  #if !SKIP_PROOF_ERRORS
149  #   if PROOF_SIZE < COMPLIANT_PROOF_SIZE
150  #       error "PROOF_SIZE is not compliant with TPM specification"
151  #   endif
152  #   if PRIMARY_SEED_SIZE < COMPLIANT_PRIMARY_SEED_SIZE
153  #       error  "Implementation.h specifies a non-compliant PRIMARY_SEED_SIZE"
154  #   endif
155  #endif // !SKIP_PROOF_ERRORS
```

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

```
156  #ifndef CONTEXT_ENCRYPT_ALG
157  #define CONTEXT_ENCRYPT_ALG             CONCAT(TPM_ALG_, CONTEXT_ENCRYPT_ALGORITHM)
158  #define CONTEXT_ENCRYPT_KEY_BITS                                  \
159                  CONCAT(CONCAT(MAX_, CONTEXT_ENCRYPT_ALGORITHM), _KEY_BITS)
160  #define CONTEXT_ENCRYPT_KEY_BYTES       ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
161  #endif
```

This is updated to follow the requirement of P2 that the label not be larger than 32 bytes.

```
162  #ifndef LABEL_MAX_BUFFER
163  #define LABEL_MAX_BUFFER MIN(32, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE))
164  #endif
```


This bit is used to indicate that an authorization ticket expires on TPM Reset and TPM Restart.It is added to the timeout value returned by TPM2_PoliySigned() and TPM2_PolicySecret() and used by TPM2_PolicyTicket(). The timeout value is relative to Time (g_time). Time is reset whenever the TPM loses power and cannot be moved forward by the user (as can Clock). g_time is a 64-bit value expressing time in ms. Sealing the MSb() for a flag means that the TPM needs to be reset at least once every 292,471,208 years rather than once every 584,942,417 years.

```
#define EXPIRATION_BIT ((UINT64)1 << 63)
```

Check for consistency of the bit ordering of bit fields

```
#if BIG_ENDIAN_TPM && MOST_SIGNIFICANT_BIT_0 && USE_BIT_FIELD_STRUCTURES
#   error "Settings not consistent"
#endif
```

These macros are used to handle the variation in handling of bit fields. If

```
#if USE_BIT_FIELD_STRUCTURES // The default, old version, with bit fields
#   define IS_ATTRIBUTE(a, type, b)     ((a.b != 0))
#   define SET_ATTRIBUTE(a, type, b)       (a.b = SET)
#   define CLEAR_ATTRIBUTE(a, type, b)     (a.b = CLEAR)
#   define GET_ATTRIBUTE(a, type, b)        (a.b)
#else
#   define IS_ATTRIBUTE(a, type, b)         ((a & type##_##b) != 0)
#   define SET_ATTRIBUTE(a, type, b)        (a |= type##_##b)
#   define CLEAR_ATTRIBUTE(a, type, b)      (a &= ~type##_##b)
#   define GET_ATTRIBUTE(a, type, b)        \
        (type)((a & type##_##b) >> type##_##b##_SHIFT)
#endif
#endif // GP_MACROS_H
```

5.11 InternalRoutines.h

```
1   #ifndef     INTERNAL_ROUTINES_H
2   #define     INTERNAL_ROUTINES_H
3   #if !defined _LIB_SUPPORT_H_ && !defined _TPM_H_
4   #error "Should not be called"
5   #endif
```

DRTM functions

```
6   #include "_TPM_Hash_Start_fp.h"
7   #include "_TPM_Hash_Data_fp.h"
8   #include "_TPM_Hash_End_fp.h"
```

Internal subsystem functions

```
9   #include "Object_fp.h"
10  #include "Context_spt_fp.h"
11  #include "Object_spt_fp.h"
12  #include "Entity_fp.h"
13  #include "Session_fp.h"
14  #include "Hierarchy_fp.h"
15  #include "NvReserved_fp.h"
16  #include "NvDynamic_fp.h"
17  #include "NV_spt_fp.h"
18  #include "PCR_fp.h"
19  #include "DA_fp.h"
20  #include "TpmFail_fp.h"
21  #include "SessionProcess_fp.h"
```

Internal support functions

```
22  #include "CommandCodeAttributes_fp.h"
23  #include "Marshal_fp.h"
24  #include "Time_fp.h"
25  #include "Locality_fp.h"
26  #include "PP_fp.h"
27  #include "CommandAudit_fp.h"
28  #include "Manufacture_fp.h"
29  #include "Handle_fp.h"
30  #include "Power_fp.h"
31  #include "Response_fp.h"
32  #include "CommandDispatcher_fp.h"
33  #ifdef CC_AC_Send
34  #   include "AC_spt_fp.h"
35  #endif // CC_AC_Send
```

Miscellaneous

```
36  #include "Bits_fp.h"
37  #include "AlgorithmCap_fp.h"
38  #include "PropertyCap_fp.h"
39  #include "IoBuffers_fp.h"
40  #include "Memory_fp.h"
41  #include "ResponseCodeProcessing_fp.h"
```

Internal cryptographic functions

```
42  #include "BnConvert_fp.h"
43  #include "BnMath_fp.h"
44  #include "BnMemory_fp.h"
45  #include "Ticket_fp.h"
```



```
46  #include "CryptUtil_fp.h"
47  #include "CryptHash_fp.h"
48  #include "CryptSym_fp.h"
49  #include "CryptDes_fp.h"
50  #include "CryptPrime_fp.h"
51  #include "CryptRand_fp.h"
52  #include "CryptSelfTest_fp.h"
53  #include "MathOnByteBuffers_fp.h"
54  #include "CryptSym_fp.h"
55  #include "AlgorithmTests_fp.h"
56  #if ALG_RSA
57  #include "CryptRsa_fp.h"
58  #include "CryptPrimeSieve_fp.h"
59  #endif
60  #if ALG_ECC
61  #include "CryptEccMain_fp.h"
62  #include "CryptEccSignature_fp.h"
63  #include "CryptEccKeyExchange_fp.h"
64  #endif
65  #if CC_MAC || CC_MAC_Start
66  #   include "CryptSmac_fp.h"
67  #   if  ALG_CMAC
68  #       include "CryptCmac_fp.h"
69  #   endif
70  #endif
```

Support library

```
71  #include "SupportLibraryFunctionPrototypes_fp.h"
```

Linkage to platform functions

```
72  #include "Platform_fp.h"
73  #endif
```

5.12 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_
```

5.13 NV.h

5.13.1 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

```
1   #ifndef    _NV_H_
2   #define    _NV_H_
3   #ifdef     TPM_NT_ORDINARY
```

If TPM_NT_ORDINARY is defined, then the TPM_NT field is present in a TPMA_NV

```
4   #   define GET_TPM_NT(attributes) GET_ATTRIBUTE(attributes, TPMA_NV, TPM_NT)
5   #else
```

If TPM_NT_ORDINARY is not defined, then need to synthesize it from the attributes

```
6   #   define GetNv_TPM_NV(attributes)                                  \
7           (   IS_ATTRIBUTE(attributes, TPMA_NV, COUNTER)              \
8           +   (IS_ATTRIBUTE(attributes, TPMA_NV, BITS) << 1)          \
9           +   (IS_ATTRIBUTE(attributes, TPMA_NV, EXTEND) << 2)        \
10          )
11  #   define TPM_NT_ORDINARY (0)
12  #   define TPM_NT_COUNTER  (1)
13  #   define TPM_NT_BITS     (2)
14  #   define TPM_NT_EXTEND   (4)
15  #endif
```

5.13.2 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

```
16  #   define IsNvOrdinaryIndex(attributes)                            \
17                  (GET_TPM_NT(attributes) == TPM_NT_ORDINARY)
18  #   define  IsNvCounterIndex(attributes)                            \
19                  (GET_TPM_NT(attributes) == TPM_NT_COUNTER)
20  #   define  IsNvBitsIndex(attributes)                               \
21                 (GET_TPM_NT(attributes) == TPM_NT_BITS)
22  #   define  IsNvExtendIndex(attributes)                             \
23                  (GET_TPM_NT(attributes) == TPM_NT_EXTEND)
24  #ifdef TPM_NT_PIN_PASS
25  #   define  IsNvPinPassIndex(attributes)                            \
26                  (GET_TPM_NT(attributes) == TPM_NT_PIN_PASS)
27  #endif
28  #ifdef TPM_NT_PIN_FAIL
29  #   define  IsNvPinFailIndex(attributes)                            \
30                  (GET_TPM_NT(attributes) == TPM_NT_PIN_FAIL)
31  #endif
32  typedef struct {
33      UINT32      size;
34      TPM_HANDLE  handle;
35  } NV_ENTRY_HEADER;
36  #define NV_EVICT_OBJECT_SIZE        \
37       (sizeof(UINT32)  + sizeof(TPM_HANDLE) + sizeof(OBJECT))
38  #define NV_INDEX_COUNTER_SIZE       \
39      (sizeof(UINT32) + sizeof(NV_INDEX) + sizeof(UINT64))
40  #define NV_RAM_INDEX_COUNTER_SIZE   \
41      (sizeof(NV_RAM_HEADER) + sizeof(UINT64))
```



```
typedef struct {
    UINT32          size;
    TPM_HANDLE      handle;
    TPMA_NV         attributes;
} 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.

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

5.13.3 Orderly RAM Values

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

```
#define     NV_RAM_REF_INIT         0
```

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

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

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

```
#define     NV_ORDERLY_START                    \
                (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

```
#define     RAM_ORDERLY_END                     \
                (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.

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

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

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

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

```
#define RETURN_IF_ORDERLY                           \
{                                                   \
    if(NvClearOrderly() != TPM_RC_SUCCESS)          \
        return g_NvStatus;                          \
}
#define NV_IS_AVAILABLE     (g_NvStatus == TPM_RC_SUCCESS)
#define IS_ORDERLY(value)   (value < SU_DA_USED_VALUE)
#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.

```
72  #define SET_NV_UPDATE(type)     g_updateNV |= (type)
73  #endif  // _NV_H_
```


5.14 PRNG_TestVectors.h

```
1   #ifndef     _MSBN_DRBG_TEST_VECTORS_H
2   #define     _MSBN_DRBG_TEST_VECTORS_H
3   //#if DRBG_ALGORITHM == TPM_ALG_AES && DRBG_KEY_BITS == 256
4   #if DRBG_KEY_SIZE_BITS == 256
```

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

```
5   #   define DRBG_TEST_INITIATE_ENTROPY                      \
6           0x0d, 0x15, 0xaa, 0x80, 0xb1, 0x6c, 0x3a, 0x10, \
7           0x90, 0x6c, 0xfe, 0xdb, 0x79, 0x5d, 0xae, 0x0b, \
8           0x5b, 0x81, 0x04, 0x1c, 0x5c, 0x5b, 0xfa, 0xcb, \
9           0x37, 0x3d, 0x44, 0x40, 0xd9, 0x12, 0x0f, 0x7e, \
10          0x3d, 0x6c, 0xf9, 0x09, 0x86, 0xcf, 0x52, 0xd8, \
11          0x5d, 0x3e, 0x94, 0x7d, 0x8c, 0x06, 0x1f, 0x91
12  #   define DRBG_TEST_RESEED_ENTROPY                         \
13          0x6e, 0xe7, 0x93, 0xa3, 0x39, 0x55, 0xd7, 0x2a, \
14          0xd1, 0x2f, 0xd8, 0x0a, 0x8a, 0x3f, 0xcf, 0x95, \
15          0xed, 0x3b, 0x4d, 0xac, 0x57, 0x95, 0xfe, 0x25, \
16          0xcf, 0x86, 0x9f, 0x7c, 0x27, 0x57, 0x3b, 0xbc, \
17          0x56, 0xf1, 0xac, 0xae, 0x13, 0xa6, 0x50, 0x42, \
18          0xb3, 0x40, 0x09, 0x3c, 0x46, 0x4a, 0x7a, 0x22
19  #   define DRBG_TEST_GENERATED_INTERM                       \
20          0x28, 0xe0, 0xeb, 0xb8, 0x21, 0x01, 0x66, 0x50, \
21          0x8c, 0x8f, 0x65, 0xf2, 0x20, 0x7b, 0xd0, 0xa3
22  #   define DRBG_TEST_GENERATED                              \
23          0x94, 0x6f, 0x51, 0x82, 0xd5, 0x45, 0x10, 0xb9, \
24          0x46, 0x12, 0x48, 0xf5, 0x71, 0xca, 0x06, 0xc9
25  //#elif DRBG_ALGORITHM == TPM_ALG_AES && DRBG_KEY_BITS == 128
26  #elif DRBG_KEY_SIZE_BITS == 128
27  #   define DRBG_TEST_INITIATE_ENTROPY                       \
28          0x8f, 0xc1, 0x1b, 0xdb, 0x5a, 0xab, 0xb7, 0xe0, \
29          0x93, 0xb6, 0x14, 0x28, 0xe0, 0x90, 0x73, 0x03, \
30          0xcb, 0x45, 0x9f, 0x3b, 0x60, 0x0d, 0xad, 0x87, \
31          0x09, 0x55, 0xf2, 0x2d, 0xa8, 0x0a, 0x44, 0xf8
32  #   define DRBG_TEST_RESEED_ENTROPY                         \
33          0x0c, 0xd5, 0x3c, 0xd5, 0xec, 0xcd, 0x5a, 0x10, \
34          0xd7, 0xea, 0x26, 0x61, 0x11, 0x25, 0x9b, 0x05, \
35          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
```

5.15 SelfTest.h

5.15.1 Introduction

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

```
#ifndef                 _SELF_TEST_H_
#define                 _SELF_TEST_H_
```

5.15.2 Defines

Was typing this a lot

```
#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.

```
#if ALG_AES && defined  AES_KEY_SIZE_BITS_128
#   define  AES_128     YES
#   define  AES_128_INDEX   0
#else
#   define  AES_128     NO
#endif
#if ALG_AES && defined  AES_KEY_SIZE_BITS_192
#   define  AES_192     YES
#   define  AES_192_INDEX   (AES_128)
#else
#   define  AES_192     NO
#endif
#if ALG_AES && defined  AES_KEY_SIZE_BITS_256
#   define  AES_256     YES
#   define  AES_256_INDEX   (AES_128 + AES_192)
#else
#   define  AES_256     NO
#endif
#if ALG_SM4 && defined SM4_KEY_SIZE_BITS_128
#   define  SM4_128     YES
#   define  SM4_128_INDEX   (AES_128 + AES_192 + AES_256)
#else
#   define  SM4_128     NO
#endif
#define NUM_SYMS    (AES_128 + AES_192 + AES_256 + SM4_128)
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

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

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

```
#if NUM_SYM_MODES <= 0
#error  "No symmetric modes implemented"
#elif NUM_SYM_MODES <= 8
typedef BYTE    SYM_MODES;
#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  #if     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_
```

5.16 SupportLibraryFunctionPrototypes_fp.h

5.16.1 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.

Arithmetic operations return a BOOL to indicate if the operation completed successfully or not.

```
#ifndef SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H
#define SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H
```

5.16.2 SupportLibInit()

This function is called by CryptInit() so that necessary initializations can be performed on the cryptographic library.

```
LIB_EXPORT
int SupportLibInit(void);
```

5.16.3 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.

```
void
MathLibraryCompatibilityCheck(
    void
    );
```

5.16.4 BnModMult()

Does *op1 * op2* and divide by *modulus* returning the remainder of the divide.

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

5.16.5 BnMult()

Multiplies two numbers and returns the result

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

5.16.6 BnDiv()

This function divides two *bigNum* values. The function returns FALSE if there is an error in the operation.

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


5.16.7 BnMod()

```
#define BnMod(a, b)     BnDiv(NULL, (a), (a), (b))
```

5.16.8 BnGcd()

Get the greatest common divisor of two numbers. This function is only needed when the TPM implements RSA.

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

5.16.9 BnModExp()

Do modular exponentiation using *bigNum* values. This function is only needed when the TPM implements RSA.

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

5.16.10 BnModInverse()

Modular multiplicative inverse. This function is only needed when the TPM implements RSA.

```
LIB_EXPORT BOOL BnModInverse(bigNum result, bigConst number,
                             bigConst modulus);
```

5.16.11 BnEccModMult()

This function does a point multiply of the form R = [d]S. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

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

5.16.12 BnEccModMult2()

This function does a point multiply of the form R = [d]S + [u]Q. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

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

5.16.13 BnEccAdd()

This function does a point add R = S + Q. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

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

5.16.14 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. This function is only needed if the TPM supports ECC.

```
LIB_EXPORT bigCurve
BnCurveInitialize(bigCurve E, TPM_ECC_CURVE curveId);
#endif
```

5.17 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
```

5.18 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                                   // cryptographic 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_
```

5.19 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. If the switch is listed on the command line (-DSOME_SWITCH) with no setting, then the switch will be set to YES. If the switch setting is not on the command line or if the setting is other than YES or NO, then the switch will be set to the default value. The default can either be YES or NO as indicated on each line where the default is selected.

A caution. Do not try to test these macros by inserting #defines in this file. For some curious reason, a variable set on the command line with no setting will have a value of 1. An #if SOME_VARIABLE will work if the variable is not defined or is defined on the command line with no initial setting. However, a

```
1   //      #define SOME_VARIABLE
```

is a null string and when used in #if SOME_VARIABLE will not be a proper expression If you want to test various switches, either use the command line or change the default.

```
2   #ifndef _TPM_BUILD_SWITCHES_H_
3   #define _TPM_BUILD_SWITCHES_H_
4   #undef YES
5   #define YES 1
6   #undef NO
7   #define NO 0
```

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

```
8   #ifndef DEBUG
9   #   ifdef NDEBUG
10  #       define  DEBUG   NO
11  #   else
12  #       define  DEBUG   YES
13  #   endif
14  #elif (DEBUG != NO) && (DEBUG != YES)
15  #   undef   DEBUG
16  #   define  DEBUG       YES         // Default: Either YES or NO
17  #endif
18  #include "CompilerDependencies.h"
```

This definition is required for the re-factored code

```
19  #if (!defined USE_BN_ECC_DATA)                                          \
20                  || ((USE_BN_ECC_DATA != NO) && (USE_BN_ECC_DATA != YES))
21  #   undef   USE_BN_ECC_DATA
22  #   define  USE_BN_ECC_DATA     YES     // Default: Either YES or NO
23  #endif
```

The SIMULATION switch allows certain other macros to be enabled. The things that can be enabled in a simulation include key caching, reproducible **random** sequences, instrumentation of the RSA key generation process, and certain other debug code. SIMULATION Needs to be defined as either YES or NO. This grouping of macros will make sure that it is set correctly. A simulated TPM would include a Virtual TPM. The interfaces for a Virtual TPM should be modified from the standard ones in the Simulator project.

If SIMULATION is in the compile parameters without modifiers, make SIMULATION == YES

```
24  #if !(defined SIMULATION) || ((SIMULATION != NO) && (SIMULATION != YES))
25  #   undef   SIMULATION
26  #   define  SIMULATION      YES     // Default: Either YES or NO
```

```
27  #endif
```

Define this to run the function that checks the compatibility between the chosen big number math library and the TPM code. Not all ports use this.

```
28  #if !(defined LIBRARY_COMPATABILITY_CHECK)                                  \
29          || ((LIBRARY_COMPATABILITY_CHECK != NO)                             \
30              && (LIBRARY_COMPATABILITY_CHECK != YES))
31  #   undef   LIBRARY_COMPATABILITY_CHECK
32  #   define  LIBRARY_COMPATABILITY_CHECK     YES     // Default: Either YES or NO
33  #endif
34  #if !(defined FIPS_COMPLIANT) || ((FIPS_COMPLIANT != NO) && (FIPS_COMPLIANT != YES))
35  #   undef   FIPS_COMPLIANT
36  #   define  FIPS_COMPLIANT      YES     // Default: Either YES or NO
37  #endif
```

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

```
38  #if !(defined USE_DA_USED) || ((USE_DA_USED != NO) && (USE_DA_USED != YES))
39  #   undef   USE_DA_USED
40  #   define  USE_DA_USED     YES         // Default: Either YES or NO
41  #endif
```

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

```
42  #if !(defined TABLE_DRIVEN_DISPATCH)                                        \
43          || ((TABLE_DRIVEN_DISPATCH != NO) && (TABLE_DRIVEN_DISPATCH != YES))
44  #   undef   TABLE_DRIVEN_DISPATCH
45  #   define  TABLE_DRIVEN_DISPATCH   YES     // Default: Either YES or NO
46  #endif
```

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

```
47  #if !(defined SELF_TEST) || ((SELF_TEST != NO) && (SELF_TEST != YES))
48  #   undef   SELF_TEST
49  #   define  SELF_TEST       YES         // Default: Either YES or NO
50  #endif
```

Enable the generation of RSA primes using a sieve.

```
51  #if !(defined RSA_KEY_SIEVE) || ((RSA_KEY_SIEVE != NO) && (RSA_KEY_SIEVE != YES))
52  #   undef   RSA_KEY_SIEVE
53  #   define  RSA_KEY_SIEVE           YES         // Default: Either YES or NO
54  #endif
```

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

```
55  #if RSA_KEY_SIEVE && SIMULATION
56  #   if !(defined RSA_INSTRUMENT)                                            \
57              || ((RSA_INSTRUMENT != NO) && (RSA_INSTRUMENT != YES))
58  #       undef   RSA_INSTRUMENT
59  #       define  RSA_INSTRUMENT      NO          // Default: Either YES or NO
60  #   endif
61  #endif
```

This switch enables the RNG state save and restore

```
62  #if !(defined _DRBG_STATE_SAVE)                                             \
63      || ((_DRBG_STATE_SAVE != NO) && (_DRBG_STATE_SAVE != YES))
64  #   undef   _DRBG_STATE_SAVE
```



```
65  #   define  _DRBG_STATE_SAVE        YES     // Default: Either YES or NO
66  #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.

```
67  #if !(defined COMPRESSED_LISTS)                                          \
68      || ((COMPRESSED_LISTS != NO) && (COMPRESSED_LISTS != YES))
69  #   undef   COMPRESSED_LISTS
70  #   define  COMPRESSED_LISTS        YES     // Default: Either YES or NO
71  #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.

```
72  #if !(defined CLOCK_STOPS) || ((CLOCK_STOPS != NO) && (CLOCK_STOPS != YES))
73  #   undef   CLOCK_STOPS
74  #   define  CLOCK_STOPS             NO      // Default: Either YES or NO
75  #endif
```

This switch allows use of #defines in place of pass-through marshaling or unmarshaling code. A pass-through function just calls another function to do the required function and does no parameter checking of its own. The table-driven dispatcher calls directly to the lowest level marshaling/unmarshaling code and by-passes any pass-through functions.

```
76  #if (defined USE_MARSHALING_DEFINES) && (USE_MARSHALING_DEFINES != NO)
77  #   undef   USE_MARSHALING_DEFINES
78  #   define  USE_MARSHALING_DEFINES  YES
79  #else
80  #   define USE_MARSHALING_DEFINES   YES     // Default: Either YES or NO
81  #endif
```

The switches in this group can only be enabled when doing debug during simulation

```
82  #if SIMULATION && DEBUG
```

Enables use of the key cache. Default is YES

```
83  #   if !(defined USE_RSA_KEY_CACHE)                                      \
84      || ((USE_RSA_KEY_CACHE != NO) && (USE_RSA_KEY_CACHE != YES))
85  #       undef   USE_RSA_KEY_CACHE
86  #       define  USE_RSA_KEY_CACHE   YES   // Default: Either YES or NO
87  #   endif
```

Enables use of a file to store the key cache values so that the TPM will start faster during debug. Default for this is YES

```
88  #   if USE_RSA_KEY_CACHE
89  #       if !(defined USE_KEY_CACHE_FILE)                                 \
90              || ((USE_KEY_CACHE_FILE != NO) && (USE_KEY_CACHE_FILE != YES))
91  #           undef   USE_KEY_CACHE_FILE
92  #           define  USE_KEY_CACHE_FILE  YES     // Default: Either YES or NO
93  #       endif
94  #   else
95  #       undef   USE_KEY_CACHE_FILE
96  #       define  USE_KEY_CACHE_FILE      NO
97  #   endif   // USE_RSA_KEY_CACHE
```

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. There is no default value.

```
98   #   if !(defined USE_DEBUG_RNG) || ((USE_DEBUG_RNG != NO) && (USE_DEBUG_RNG != YES))
99   #       undef   USE_DEBUG_RNG
100  #       define  USE_DEBUG_RNG           YES      // Default: Either YES or NO
101  #   endif
```

Don't change these. They are the settings needed when not doing a simulation and not doing debug. Can't use the key cache except during debug. Otherwise, all of the key values end up being the same

```
102  #else
103  #   define USE_RSA_KEY_CACHE        NO
104  #   define USE_RSA_KEY_CACHE_FILE   NO
105  #   define USE_DEBUG_RNG            NO
106  #endif  // DEBUG && SIMULATION
107  #if DEBUG
```

In some cases, the relationship between two values may be dependent on things that change based on various selections like the chosen cryptographic 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.

```
108  #   if !(defined COMPILER_CHECKS)                                          \
109          || ((COMPILER_CHECKS != NO) && (COMPILER_CHECKS != YES))
110  #       undef   COMPILER_CHECKS
111  #       define  COMPILER_CHECKS     NO      // Default: Either YES or NO
112  #   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.

```
113  #   if !(defined RUNTIME_SIZE_CHECKS)                                      \
114      || ((RUNTIME_SIZE_CHECKS != NO) && (RUNTIME_SIZE_CHECKS != YES))
115  #       undef RUNTIME_SIZE_CHECKS
116  #       define RUNTIME_SIZE_CHECKS      NO      // Default: Either YES or NO
117  #   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)

```
118  #   if !(defined DRBG_DEBUG_PRINT)                                         \
119      || ((DRBG_DEBUG_PRINT != NO) && (DRBG_DEBUG_PRINT != YES))
120  #       undef   DRBG_DEBUG_PRINT
121  #       define  DRBG_DEBUG_PRINT    NO      // Default: Either YES or NO
122  #   endif
```

If an assertion event it not going to produce any trace information (function and line number) then make FAIL_TRACE == NO

```
123  #   if !(defined FAIL_TRACE) || ((FAIL_TRACE != NO) && (FAIL_TRACE != YES))
124  #       undef   FAIL_TRACE
125  #       define  FAIL_TRACE          YES      // Default: Either YES or NO
126  #   endif
```



```
127  #endif // DEBUG
```

Indicate if the implementation is going to give lockout time credit for time up to the last orderly shutdown.

```
128  #if !(defined ACCUMULATE_SELF_HEAL_TIMER)                                    \
129      || ((ACCUMULATE_SELF_HEAL_TIMER != NO) && (ACCUMULATE_SELF_HEAL_TIMER != YES))
130  #   undef   ACCUMULATE_SELF_HEAL_TIMER
131  #   define  ACCUMULATE_SELF_HEAL_TIMER      YES       // Default: Either YES or NO
132  #endif
```

Indicates if the implementation is to compute the sizes of the proof and primary seed size values based on the implemented algorithms.

```
133  #if !(defined USE_SPEC_COMPLIANT_PROOFS)                                     \
134      || ((USE_SPEC_COMPLIANT_PROOFS != NO) && (USE_SPEC_COMPLIANT_PROOFS != YES))
135  #   undef   USE_SPEC_COMPLIANT_PROOFS
136  #   define  USE_SPEC_COMPLIANT_PROOFS       YES       // Default: Either YES or NO
137  #endif
```

Comment this out to allow compile to continue even though the chosen proof values do not match the compliant values. This is written so that someone would have to proactively ignore errors.

```
138  #if !(defined SKIP_PROOF_ERRORS)                                             \
139      || ((SKIP_PROOF_ERRORS != NO) && (SKIP_PROOF_ERRORS != YES))
140  #   undef   SKIP_PROOF_ERRORS
141  #   define  SKIP_PROOF_ERRORS          NO       // Default: Either YES or NO
142  #endif
```

This define is used to eliminate the use of bit-fields. It can be enabled for big- or little-endian machines. For big-endian architectures that numbers bits in registers from left to right *(MSb0*()) this must be enabled. Little-endian machines number from right to left with the least significant bit having assigned a bit number of 0. These are LSb0() machines (they are also little-endian so they are also least-significant byte 0 (LSB0) machines. Big-endian (MSB0) machines may number in either direction *(MSb0*() or LSb0()). For an MSB0+MSb0() machine this value is required to be *NO*

```
143  #if !(defined USE_BIT_FIELD_STRUCTURES)                                      \
144      || ((USE_BIT_FIELD_STRUCTURES != NO) && (USE_BIT_FIELD_STRUCTURES != YES))
145  #   undef   USE_BIT_FIELD_STRUCTURES
146  #   define  USE_BIT_FIELD_STRUCTURES    YES         // Default: Either YES or NO
147  #endif
```

This define is used to enable any runtime checks of the interface between the cryptographic library (e.g., OpenSSL()) and the thunking layer.

```
148  #if !(defined LIBRARY_COMPATIBILITY_CHECK)                                   \
149      || ((LIBRARY_COMPATIBILITY_CHECK != NO) && (LIBRARY_COMPATIBILITY_CHECK != YES))
150  #   undef   LIBRARY_COMPATIBILITY_CHECK
151  #   define  LIBRARY_COMPATIBILITY_CHECK    NO        // Default: Either YES or NO
152  #endif
```

Change these definitions to turn all algorithms or commands ON or OFF. That is, to turn all algorithms on, set ALG_NO to YES. This is mostly useful as a debug feature.

```
153  #define      ALG_YES      YES
154  #define      ALG_NO       NO
155  #define      CC_YES       YES
156  #define      CC_NO        NO
157  #endif // _TPM_BUILD_SWITCHES_H_
```

5.20 TpmError.h

```
1   #ifndef _TPM_ERROR_H
2   #define _TPM_ERROR_H
3   #define     FATAL_ERROR_ALLOCATION              (1)
4   #define     FATAL_ERROR_DIVIDE_ZERO             (2)
5   #define     FATAL_ERROR_INTERNAL                (3)
6   #define     FATAL_ERROR_PARAMETER               (4)
7   #define     FATAL_ERROR_ENTROPY                 (5)
8   #define     FATAL_ERROR_SELF_TEST               (6)
9   #define     FATAL_ERROR_CRYPTO                  (7)
10  #define     FATAL_ERROR_NV_UNRECOVERABLE        (8)
11  #define     FATAL_ERROR_REMANUFACTURED          (9) // indicates that the TPM has
12                                                      // been re-manufactured after an
13                                                      // unrecoverable NV error
14  #define     FATAL_ERROR_DRBG                    (10)
15  #define     FATAL_ERROR_MOVE_SIZE               (11)
16  #define     FATAL_ERROR_COUNTER_OVERFLOW        (12)
17  #define     FATAL_ERROR_SUBTRACT                (13)
18  #define     FATAL_ERROR_FORCED                  (666)
19  #endif // _TPM_ERROR_H
```

5.21 TpmTypes.h

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

Table 2:5 - Definition of Types for Documentation Clarity

```
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

```
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            150
#define TPM_SPEC_VERSION        (TPM_SPEC)(SPEC_VERSION)
#define SPEC_YEAR               2018
#define TPM_SPEC_YEAR           (TPM_SPEC)(SPEC_YEAR)
#define SPEC_DAY_OF_YEAR        262
#define TPM_SPEC_DAY_OF_YEAR    (TPM_SPEC)(SPEC_DAY_OF_YEAR)
```

Table 2:7 - Definition of TPM_GENERATED Constants

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

Table 2:16 - Definition of TPM_RC Constants

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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


```
typedef UINT32              TPM_HANDLE;
```

Table 2:27 - Definition of TPM_HT Constants

```
typedef UINT8                   TPM_HT;
#define TPM_HT_PCR              (TPM_HT)(0x00)
#define TPM_HT_NV_INDEX         (TPM_HT)(0x01)
#define TPM_HT_HMAC_SESSION     (TPM_HT)(0x02)
#define TPM_HT_LOADED_SESSION   (TPM_HT)(0x02)
#define TPM_HT_POLICY_SESSION   (TPM_HT)(0x03)
#define TPM_HT_SAVED_SESSION    (TPM_HT)(0x03)
#define TPM_HT_PERMANENT        (TPM_HT)(0x40)
#define TPM_HT_TRANSIENT        (TPM_HT)(0x80)
#define TPM_HT_PERSISTENT       (TPM_HT)(0x81)
#define TPM_HT_AC               (TPM_HT)(0x90)
```

Table 2:28 - Definition of TPM_RH Constants

```
typedef TPM_HANDLE          TPM_RH;
#define TPM_RH_FIRST        (TPM_RH)(0x40000000)
#define TPM_RH_SRK          (TPM_RH)(0x40000000)
#define TPM_RH_OWNER        (TPM_RH)(0x40000001)
#define TPM_RH_REVOKE       (TPM_RH)(0x40000002)
#define TPM_RH_TRANSPORT    (TPM_RH)(0x40000003)
#define TPM_RH_OPERATOR     (TPM_RH)(0x40000004)
#define TPM_RH_ADMIN        (TPM_RH)(0x40000005)
#define TPM_RH_EK           (TPM_RH)(0x40000006)
#define TPM_RH_NULL         (TPM_RH)(0x40000007)
#define TPM_RH_UNASSIGNED   (TPM_RH)(0x40000008)
#define TPM_RS_PW           (TPM_RH)(0x40000009)
#define TPM_RH_LOCKOUT      (TPM_RH)(0x4000000A)
#define TPM_RH_ENDORSEMENT  (TPM_RH)(0x4000000B)
#define TPM_RH_PLATFORM     (TPM_RH)(0x4000000C)
#define TPM_RH_PLATFORM_NV  (TPM_RH)(0x4000000D)
#define TPM_RH_AUTH_00      (TPM_RH)(0x40000010)
#define TPM_RH_AUTH_FF      (TPM_RH)(0x4000010F)
#define TPM_RH_LAST         (TPM_RH)(0x4000010F)
```

Table 2:29 - Definition of TPM_HC Constants

```
typedef TPM_HANDLE              TPM_HC;
#define HR_HANDLE_MASK          (TPM_HC)(0x00FFFFFF)
#define HR_RANGE_MASK           (TPM_HC)(0xFF000000)
#define HR_SHIFT                (TPM_HC)(24)
#define HR_PCR                  (TPM_HC)((TPM_HT_PCR<<HR_SHIFT))
#define HR_HMAC_SESSION         (TPM_HC)((TPM_HT_HMAC_SESSION<<HR_SHIFT))
#define HR_POLICY_SESSION       (TPM_HC)((TPM_HT_POLICY_SESSION<<HR_SHIFT))
#define HR_TRANSIENT            (TPM_HC)((TPM_HT_TRANSIENT<<HR_SHIFT))
#define HR_PERSISTENT           (TPM_HC)((TPM_HT_PERSISTENT<<HR_SHIFT))
#define HR_NV_INDEX             (TPM_HC)((TPM_HT_NV_INDEX<<HR_SHIFT))
#define HR_PERMANENT            (TPM_HC)((TPM_HT_PERMANENT<<HR_SHIFT))
#define PCR_FIRST               (TPM_HC)((HR_PCR+0))
#define PCR_LAST                (TPM_HC)((PCR_FIRST+IMPLEMENTATION_PCR-1))
#define HMAC_SESSION_FIRST      (TPM_HC)((HR_HMAC_SESSION+0))
#define HMAC_SESSION_LAST       (TPM_HC)((HMAC_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
#define LOADED_SESSION_FIRST    (TPM_HC)(HMAC_SESSION_FIRST)
#define LOADED_SESSION_LAST     (TPM_HC)(HMAC_SESSION_LAST)
#define POLICY_SESSION_FIRST    (TPM_HC)((HR_POLICY_SESSION+0))
#define POLICY_SESSION_LAST     \
            (TPM_HC)((POLICY_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
#define TRANSIENT_FIRST         (TPM_HC)((HR_TRANSIENT+0))
#define ACTIVE_SESSION_FIRST    (TPM_HC)(POLICY_SESSION_FIRST)
#define ACTIVE_SESSION_LAST     (TPM_HC)(POLICY_SESSION_LAST)
#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)
406  #define HR_NV_AC                (TPM_HC)(((TPM_HT_NV_INDEX<<HR_SHIFT)+0xD00000))
407  #define NV_AC_FIRST             (TPM_HC)((HR_NV_AC+0))
408  #define NV_AC_LAST              (TPM_HC)((HR_NV_AC+0x0000FFFF))
409  #define HR_AC                   (TPM_HC)((TPM_HT_AC<<HR_SHIFT))
410  #define AC_FIRST                (TPM_HC)((HR_AC+0))
411  #define AC_LAST                 (TPM_HC)((HR_AC+0x0000FFFF))
412  #if USE_BIT_FIELD_STRUCTURES
413  typedef struct TPMA_ALGORITHM {                     // Table 2:30
414      unsigned    asymmetric           : 1;
415      unsigned    symmetric            : 1;
416      unsigned    hash                 : 1;
417      unsigned    object               : 1;
418      unsigned    Reserved_bits_at_4   : 4;
419      unsigned    signing              : 1;
420      unsigned    encrypting           : 1;
421      unsigned    method               : 1;
422      unsigned    Reserved_bits_at_11  : 21;
423  } TPMA_ALGORITHM;                                   /* Bits */
```

This is the initializer for a TPMA_ALGORITHM structure

```
424  #define TPMA_ALGORITHM_INITIALIZER(                                          \
425              asymmetric, symmetric,  hash,       object,     bits_at_4,       \
426              signing,    encrypting, method,     bits_at_11)                  \
427             {asymmetric, symmetric,  hash,       object,     bits_at_4,       \
428              signing,    encrypting, method,     bits_at_11}
429  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:30 TPMA_ALGORITHM using bit masking

```
430  typedef UINT32                      TPMA_ALGORITHM;
431  #define TPMA_ALGORITHM_asymmetric   ((TPMA_ALGORITHM)1 << 0)
432  #define TPMA_ALGORITHM_symmetric    ((TPMA_ALGORITHM)1 << 1)
433  #define TPMA_ALGORITHM_hash         ((TPMA_ALGORITHM)1 << 2)
434  #define TPMA_ALGORITHM_object       ((TPMA_ALGORITHM)1 << 3)
435  #define TPMA_ALGORITHM_signing      ((TPMA_ALGORITHM)1 << 8)
436  #define TPMA_ALGORITHM_encrypting   ((TPMA_ALGORITHM)1 << 9)
437  #define TPMA_ALGORITHM_method       ((TPMA_ALGORITHM)1 << 10)
```

This is the initializer for a TPMA_ALGORITHM bit array.

```
438  #define TPMA_ALGORITHM_INITIALIZER(                                          \
439              asymmetric, symmetric,  hash,       object,     bits_at_4,       \
440              signing,    encrypting, method,     bits_at_11)                  \
441             ((asymmetric << 0) + (symmetric << 1)  + (hash << 2)       +      \
442              (object << 3)     + (signing << 8)    + (encrypting << 9) +      \
443              (method << 10))
444  #endif // USE_BIT_FIELD_STRUCTURES
445  #if USE_BIT_FIELD_STRUCTURES
446  typedef struct TPMA_OBJECT {                        // Table 2:31
447      unsigned    Reserved_bit_at_0        : 1;
448      unsigned    fixedTPM                 : 1;
449      unsigned    stClear                  : 1;
450      unsigned    Reserved_bit_at_3        : 1;
451      unsigned    fixedParent              : 1;
452      unsigned    sensitiveDataOrigin      : 1;
453      unsigned    userWithAuth             : 1;
```



```
454         unsigned    adminWithPolicy          : 1;
455         unsigned    Reserved_bits_at_8       : 2;
456         unsigned    noDA                     : 1;
457         unsigned    encryptedDuplication     : 1;
458         unsigned    Reserved_bits_at_12      : 4;
459         unsigned    restricted               : 1;
460         unsigned    decrypt                  : 1;
461         unsigned    sign                     : 1;
462         unsigned    Reserved_bits_at_19      : 13;
463     } TPMA_OBJECT;                                      /* Bits */
```

This is the initializer for a TPMA_OBJECT structure

```
464     #define TPMA_OBJECT_INITIALIZER(                                                    \
465                 bit_at_0,           fixedtpm,           stclear,                        \
466                 bit_at_3,           fixedparent,        sensitivedataorigin,            \
467                 userwithauth,       adminwithpolicy,    bits_at_8,                      \
468                 noda,               encryptedduplication, bits_at_12,                   \
469                 restricted,         decrypt,            sign,                           \
470                 bits_at_19)                                                             \
471                {bit_at_0,           fixedtpm,           stclear,                        \
472                 bit_at_3,           fixedparent,        sensitivedataorigin,            \
473                 userwithauth,       adminwithpolicy,    bits_at_8,                      \
474                 noda,               encryptedduplication, bits_at_12,                   \
475                 restricted,         decrypt,            sign,                           \
476                 bits_at_19}
477     #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:31 TPMA_OBJECT using bit masking

```
478     typedef UINT32                          TPMA_OBJECT;
479     #define TPMA_OBJECT_fixedTPM            ((TPMA_OBJECT)1 << 1)
480     #define TPMA_OBJECT_stClear             ((TPMA_OBJECT)1 << 2)
481     #define TPMA_OBJECT_fixedParent         ((TPMA_OBJECT)1 << 4)
482     #define TPMA_OBJECT_sensitiveDataOrigin ((TPMA_OBJECT)1 << 5)
483     #define TPMA_OBJECT_userWithAuth        ((TPMA_OBJECT)1 << 6)
484     #define TPMA_OBJECT_adminWithPolicy     ((TPMA_OBJECT)1 << 7)
485     #define TPMA_OBJECT_noDA                ((TPMA_OBJECT)1 << 10)
486     #define TPMA_OBJECT_encryptedDuplication ((TPMA_OBJECT)1 << 11)
487     #define TPMA_OBJECT_restricted          ((TPMA_OBJECT)1 << 16)
488     #define TPMA_OBJECT_decrypt             ((TPMA_OBJECT)1 << 17)
489     #define TPMA_OBJECT_sign                ((TPMA_OBJECT)1 << 18)
```

This is the initializer for a TPMA_OBJECT bit array.

```
490     #define TPMA_OBJECT_INITIALIZER(                                                    \
491                 bit_at_0,           fixedtpm,           stclear,                        \
492                 bit_at_3,           fixedparent,        sensitivedataorigin,            \
493                 userwithauth,       adminwithpolicy,    bits_at_8,                      \
494                 noda,               encryptedduplication, bits_at_12,                   \
495                 restricted,         decrypt,            sign,                           \
496                 bits_at_19)                                                             \
497                ((fixedtpm << 1)             + (stclear << 2)              +             \
498                 (fixedparent << 4)          + (sensitivedataorigin << 5)  +             \
499                 (userwithauth << 6)         + (adminwithpolicy << 7)      +             \
500                 (noda << 10)                + (encryptedduplication << 11) +            \
501                 (restricted << 16)          + (decrypt << 17)             +             \
502                 (sign << 18))
503     #endif // USE_BIT_FIELD_STRUCTURES
504     #if USE_BIT_FIELD_STRUCTURES
505     typedef struct TPMA_SESSION {                       // Table 2:32
506         unsigned    continueSession      : 1;
507         unsigned    auditExclusive       : 1;
508         unsigned    auditReset           : 1;
```

```
509     unsigned    Reserved_bits_at_3   : 2;
510     unsigned    decrypt              : 1;
511     unsigned    encrypt              : 1;
512     unsigned    audit                : 1;
513 } TPMA_SESSION;                                  /* Bits */
```

This is the initializer for a TPMA_SESSION structure

```
514 #define TPMA_SESSION_INITIALIZER(                                                  \
515             continuesession, auditexclusive,  auditreset,      bits_at_3,         \
516             decrypt,         encrypt,         audit)                               \
517            {continuesession, auditexclusive,  auditreset,      bits_at_3,         \
518             decrypt,         encrypt,         audit}
519 #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:32 TPMA_SESSION using bit masking

```
520 typedef UINT8                           TPMA_SESSION;
521 #define TPMA_SESSION_continueSession    ((TPMA_SESSION)1 << 0)
522 #define TPMA_SESSION_auditExclusive     ((TPMA_SESSION)1 << 1)
523 #define TPMA_SESSION_auditReset         ((TPMA_SESSION)1 << 2)
524 #define TPMA_SESSION_decrypt            ((TPMA_SESSION)1 << 5)
525 #define TPMA_SESSION_encrypt            ((TPMA_SESSION)1 << 6)
526 #define TPMA_SESSION_audit              ((TPMA_SESSION)1 << 7)
```

This is the initializer for a TPMA_SESSION bit array.

```
527 #define TPMA_SESSION_INITIALIZER(                                                  \
528             continuesession, auditexclusive,  auditreset,      bits_at_3,         \
529             decrypt,         encrypt,         audit)                               \
530            ((continuesession << 0) + (auditexclusive << 1)  +                      \
531             (auditreset << 2)      + (decrypt << 5)         +                      \
532             (encrypt << 6)         + (audit << 7))
533 #endif // USE_BIT_FIELD_STRUCTURES
534 #if USE_BIT_FIELD_STRUCTURES
535 typedef struct TPMA_LOCALITY {                      // Table 2:33
536     unsigned    TPM_LOC_ZERO         : 1;
537     unsigned    TPM_LOC_ONE          : 1;
538     unsigned    TPM_LOC_TWO          : 1;
539     unsigned    TPM_LOC_THREE        : 1;
540     unsigned    TPM_LOC_FOUR         : 1;
541     unsigned    Extended             : 3;
542 } TPMA_LOCALITY;                                 /* Bits */
```

This is the initializer for a TPMA_LOCALITY structure

```
543 #define TPMA_LOCALITY_INITIALIZER(                                                 \
544             tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,            \
545             tpm_loc_four,  extended)                                               \
546            {tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,            \
547             tpm_loc_four,  extended}
548 #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:33 TPMA_LOCALITY using bit masking

```
549 typedef UINT8                           TPMA_LOCALITY;
550 #define TPMA_LOCALITY_TPM_LOC_ZERO      ((TPMA_LOCALITY)1 << 0)
551 #define TPMA_LOCALITY_TPM_LOC_ONE       ((TPMA_LOCALITY)1 << 1)
552 #define TPMA_LOCALITY_TPM_LOC_TWO       ((TPMA_LOCALITY)1 << 2)
553 #define TPMA_LOCALITY_TPM_LOC_THREE     ((TPMA_LOCALITY)1 << 3)
554 #define TPMA_LOCALITY_TPM_LOC_FOUR      ((TPMA_LOCALITY)1 << 4)
555 #define TPMA_LOCALITY_Extended_SHIFT    5
556 #define TPMA_LOCALITY_Extended          ((TPMA_LOCALITY)0x7 << 5)
```


This is the initializer for a TPMA_LOCALITY bit array.

```
557  #define TPMA_LOCALITY_INITIALIZER(                                                  \
558                 tpm_loc_zero,  tpm_loc_one,   tpm_loc_two,   tpm_loc_three,          \
559                 tpm_loc_four,  extended)                                             \
560                ((tpm_loc_zero << 0)  + (tpm_loc_one << 1)   + (tpm_loc_two << 2)   +  \
561                 (tpm_loc_three << 3) + (tpm_loc_four << 4)  + (extended << 5))
562  #endif // USE_BIT_FIELD_STRUCTURES
563  #if USE_BIT_FIELD_STRUCTURES
564  typedef struct TPMA_PERMANENT {                        // Table 2:34
565      unsigned    ownerAuthSet          : 1;
566      unsigned    endorsementAuthSet    : 1;
567      unsigned    lockoutAuthSet        : 1;
568      unsigned    Reserved_bits_at_3    : 5;
569      unsigned    disableClear          : 1;
570      unsigned    inLockout             : 1;
571      unsigned    tpmGeneratedEPS       : 1;
572      unsigned    Reserved_bits_at_11   : 21;
573  } TPMA_PERMANENT;                                   /* Bits */
```

This is the initializer for a TPMA_PERMANENT structure

```
574  #define TPMA_PERMANENT_INITIALIZER(                                                 \
575                 ownerauthset,       endorsementauthset, lockoutauthset,              \
576                 bits_at_3,          disableclear,       inlockout,                   \
577                 tpmgeneratedeps,    bits_at_11)                                      \
578                {ownerauthset,       endorsementauthset, lockoutauthset,              \
579                 bits_at_3,          disableclear,       inlockout,                   \
580                 tpmgeneratedeps,    bits_at_11}
581  #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:34 TPMA_PERMANENT using bit masking

```
582  typedef UINT32                          TPMA_PERMANENT;
583  #define TPMA_PERMANENT_ownerAuthSet         ((TPMA_PERMANENT)1 << 0)
584  #define TPMA_PERMANENT_endorsementAuthSet   ((TPMA_PERMANENT)1 << 1)
585  #define TPMA_PERMANENT_lockoutAuthSet       ((TPMA_PERMANENT)1 << 2)
586  #define TPMA_PERMANENT_disableClear         ((TPMA_PERMANENT)1 << 8)
587  #define TPMA_PERMANENT_inLockout            ((TPMA_PERMANENT)1 << 9)
588  #define TPMA_PERMANENT_tpmGeneratedEPS      ((TPMA_PERMANENT)1 << 10)
```

This is the initializer for a TPMA_PERMANENT bit array.

```
589  #define TPMA_PERMANENT_INITIALIZER(                                                 \
590                 ownerauthset,       endorsementauthset, lockoutauthset,              \
591                 bits_at_3,          disableclear,       inlockout,                   \
592                 tpmgeneratedeps,    bits_at_11)                                      \
593                ((ownerauthset << 0)       + (endorsementauthset << 1) +              \
594                 (lockoutauthset << 2)     + (disableclear << 8)       +              \
595                 (inlockout << 9)          + (tpmgeneratedeps << 10))
596  #endif // USE_BIT_FIELD_STRUCTURES
597  #if USE_BIT_FIELD_STRUCTURES
598  typedef struct TPMA_STARTUP_CLEAR {                    // Table 2:35
599      unsigned    phEnable              : 1;
600      unsigned    shEnable              : 1;
601      unsigned    ehEnable              : 1;
602      unsigned    phEnableNV            : 1;
603      unsigned    Reserved_bits_at_4    : 27;
604      unsigned    orderly               : 1;
605  } TPMA_STARTUP_CLEAR;                               /* Bits */
```

This is the initializer for a TPMA_STARTUP_CLEAR structure

```
#define TPMA_STARTUP_CLEAR_INITIALIZER(                                                 \
                phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)           \
               {phenable, shenable, ehenable, phenablenv, bits_at_4, orderly}
#else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:35 TPMA_STARTUP_CLEAR using bit masking

```
typedef UINT32                          TPMA_STARTUP_CLEAR;
#define TPMA_STARTUP_CLEAR_phEnable     ((TPMA_STARTUP_CLEAR)1 << 0)
#define TPMA_STARTUP_CLEAR_shEnable     ((TPMA_STARTUP_CLEAR)1 << 1)
#define TPMA_STARTUP_CLEAR_ehEnable     ((TPMA_STARTUP_CLEAR)1 << 2)
#define TPMA_STARTUP_CLEAR_phEnableNV   ((TPMA_STARTUP_CLEAR)1 << 3)
#define TPMA_STARTUP_CLEAR_orderly      ((TPMA_STARTUP_CLEAR)1 << 31)
```

This is the initializer for a TPMA_STARTUP_CLEAR bit array.

```
#define TPMA_STARTUP_CLEAR_INITIALIZER(                                                 \
                phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)           \
               ((phenable << 0)   + (shenable << 1)   + (ehenable << 2)   +             \
                (phenablenv << 3) + (orderly << 31))
#endif // USE_BIT_FIELD_STRUCTURES
#if USE_BIT_FIELD_STRUCTURES
typedef struct TPMA_MEMORY {                            // Table 2:36
    unsigned    sharedRAM           : 1;
    unsigned    sharedNV            : 1;
    unsigned    objectCopiedToRam   : 1;
    unsigned    Reserved_bits_at_3  : 29;
} TPMA_MEMORY;                                          /* Bits */
```

This is the initializer for a TPMA_MEMORY structure

```
#define TPMA_MEMORY_INITIALIZER(                                                        \
                sharedram, sharednv, objectcopiedtoram, bits_at_3)                      \
               {sharedram, sharednv, objectcopiedtoram, bits_at_3}
#else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:36 TPMA_MEMORY using bit masking

```
typedef UINT32                          TPMA_MEMORY;
#define TPMA_MEMORY_sharedRAM           ((TPMA_MEMORY)1 << 0)
#define TPMA_MEMORY_sharedNV            ((TPMA_MEMORY)1 << 1)
#define TPMA_MEMORY_objectCopiedToRam   ((TPMA_MEMORY)1 << 2)
```

This is the initializer for a TPMA_MEMORY bit array.

```
#define TPMA_MEMORY_INITIALIZER(                                                        \
                sharedram, sharednv, objectcopiedtoram, bits_at_3)                      \
               ((sharedram << 0) + (sharednv << 1) + (objectcopiedtoram << 2))
#endif // USE_BIT_FIELD_STRUCTURES
#if USE_BIT_FIELD_STRUCTURES
typedef struct TPMA_CC {                                // Table 2:37
    unsigned    commandIndex        : 16;
    unsigned    Reserved_bits_at_16 : 6;
    unsigned    nv                  : 1;
    unsigned    extensive           : 1;
    unsigned    flushed             : 1;
    unsigned    cHandles            : 3;
    unsigned    rHandle             : 1;
    unsigned    V                   : 1;
    unsigned    Reserved_bits_at_30 : 2;
} TPMA_CC;                                              /* Bits */
```

This is the initializer for a TPMA_CC structure


```
#define TPMA_CC_INITIALIZER(                                                    \
            commandindex, bits_at_16,   nv,          extensive,    flushed,     \
            chandles,     rhandle,      v,           bits_at_30)                \
           {commandindex, bits_at_16,   nv,          extensive,    flushed,     \
            chandles,     rhandle,      v,           bits_at_30}
#else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:37 TPMA_CC using bit masking

```
typedef UINT32                      TPMA_CC;
#define TPMA_CC_commandIndex_SHIFT  0
#define TPMA_CC_commandIndex        ((TPMA_CC)0xffff << 0)
#define TPMA_CC_nv                  ((TPMA_CC)1 << 22)
#define TPMA_CC_extensive           ((TPMA_CC)1 << 23)
#define TPMA_CC_flushed             ((TPMA_CC)1 << 24)
#define TPMA_CC_cHandles_SHIFT      25
#define TPMA_CC_cHandles            ((TPMA_CC)0x7 << 25)
#define TPMA_CC_rHandle             ((TPMA_CC)1 << 28)
#define TPMA_CC_V                   ((TPMA_CC)1 << 29)
```

This is the initializer for a TPMA_CC bit array.

```
#define TPMA_CC_INITIALIZER(                                                    \
            commandindex, bits_at_16,   nv,          extensive,    flushed,     \
            chandles,     rhandle,      v,           bits_at_30)                \
           ((commandindex << 0) + (nv << 22)         + (extensive << 23)   +    \
            (flushed << 24)     + (chandles << 25)   + (rhandle << 28)     +    \
            (v << 29))
#endif // USE_BIT_FIELD_STRUCTURES
#if USE_BIT_FIELD_STRUCTURES
typedef struct TPMA_MODES {                          // Table 2:38
    unsigned    FIPS_140_2          : 1;
    unsigned    Reserved_bits_at_1  : 31;
} TPMA_MODES;                                        /* Bits */
```

This is the initializer for a TPMA_MODES structure

```
#define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {fips_140_2, bits_at_1}
#else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:38 TPMA_MODES using bit masking

```
typedef UINT32                  TPMA_MODES;
#define TPMA_MODES_FIPS_140_2   ((TPMA_MODES)1 << 0)
```

This is the initializer for a TPMA_MODES bit array.

```
#define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) ((fips_140_2 << 0))
#endif // USE_BIT_FIELD_STRUCTURES
typedef BYTE                TPMI_YES_NO;            // Table 2:39  /* Interface */
typedef TPM_HANDLE          TPMI_DH_OBJECT;         // Table 2:40  /* Interface */
typedef TPM_HANDLE          TPMI_DH_PARENT;         // Table 2:41  /* Interface */
typedef TPM_HANDLE          TPMI_DH_PERSISTENT;     // Table 2:42  /* Interface */
typedef TPM_HANDLE          TPMI_DH_ENTITY;         // Table 2:43  /* Interface */
typedef TPM_HANDLE          TPMI_DH_PCR;            // Table 2:44  /* Interface */
typedef TPM_HANDLE          TPMI_SH_AUTH_SESSION;   // Table 2:45  /* Interface */
typedef TPM_HANDLE          TPMI_SH_HMAC;           // Table 2:46  /* Interface */
typedef TPM_HANDLE          TPMI_SH_POLICY;         // Table 2:47  /* Interface */
typedef TPM_HANDLE          TPMI_DH_CONTEXT;        // Table 2:48  /* Interface */
typedef TPM_HANDLE          TPMI_DH_SAVED;          // Table 2:49  /* Interface */
typedef TPM_HANDLE          TPMI_RH_HIERARCHY;      // Table 2:50  /* Interface */
typedef TPM_HANDLE          TPMI_RH_ENABLES;        // Table 2:51  /* Interface */
typedef TPM_HANDLE          TPMI_RH_HIERARCHY_AUTH; // Table 2:52  /* Interface */
```

```
700  typedef TPM_HANDLE          TPMI_RH_PLATFORM;       // Table 2:53  /* Interface */
701  typedef TPM_HANDLE          TPMI_RH_OWNER;          // Table 2:54  /* Interface */
702  typedef TPM_HANDLE          TPMI_RH_ENDORSEMENT;    // Table 2:55  /* Interface */
703  typedef TPM_HANDLE          TPMI_RH_PROVISION;      // Table 2:56  /* Interface */
704  typedef TPM_HANDLE          TPMI_RH_CLEAR;          // Table 2:57  /* Interface */
705  typedef TPM_HANDLE          TPMI_RH_NV_AUTH;        // Table 2:58  /* Interface */
706  typedef TPM_HANDLE          TPMI_RH_LOCKOUT;        // Table 2:59  /* Interface */
707  typedef TPM_HANDLE          TPMI_RH_NV_INDEX;       // Table 2:60  /* Interface */
708  typedef TPM_HANDLE          TPMI_RH_AC;             // Table 2:61  /* Interface */
709  typedef TPM_ALG_ID          TPMI_ALG_HASH;          // Table 2:62  /* Interface */
710  typedef TPM_ALG_ID          TPMI_ALG_ASYM;          // Table 2:63  /* Interface */
711  typedef TPM_ALG_ID          TPMI_ALG_SYM;           // Table 2:64  /* Interface */
712  typedef TPM_ALG_ID          TPMI_ALG_SYM_OBJECT;    // Table 2:65  /* Interface */
713  typedef TPM_ALG_ID          TPMI_ALG_SYM_MODE;      // Table 2:66  /* Interface */
714  typedef TPM_ALG_ID          TPMI_ALG_KDF;           // Table 2:67  /* Interface */
715  typedef TPM_ALG_ID          TPMI_ALG_SIG_SCHEME;    // Table 2:68  /* Interface */
716  typedef TPM_ALG_ID          TPMI_ECC_KEY_EXCHANGE;  // Table 2:69  /* Interface */
717  typedef TPM_ST              TPMI_ST_COMMAND_TAG;    // Table 2:70  /* Interface */
718  typedef TPM_ALG_ID          TPMI_ALG_MAC_SCHEME;    // Table 2:71  /* Interface */
719  typedef TPM_ALG_ID          TPMI_ALG_CIPHER_MODE;   // Table 2:72  /* Interface */
720  typedef BYTE                TPMS_EMPTY;             // Table 2:73
721  typedef struct {                                    // Table 2:74
722      TPM_ALG_ID              alg;
723      TPMA_ALGORITHM          attributes;
724  } TPMS_ALGORITHM_DESCRIPTION;                       /* Structure */
725  typedef union {                                     // Table 2:75
726  #if       ALG_SHA1
727      BYTE                    sha1[SHA1_DIGEST_SIZE];
728  #endif // ALG_SHA1
729  #if       ALG_SHA256
730      BYTE                    sha256[SHA256_DIGEST_SIZE];
731  #endif // ALG_SHA256
732  #if       ALG_SHA384
733      BYTE                    sha384[SHA384_DIGEST_SIZE];
734  #endif // ALG_SHA384
735  #if       ALG_SHA512
736      BYTE                    sha512[SHA512_DIGEST_SIZE];
737  #endif // ALG_SHA512
738  #if       ALG_SM3_256
739      BYTE                    sm3_256[SM3_256_DIGEST_SIZE];
740  #endif // ALG_SM3_256
741  } TPMU_HA;                                          /* Structure */
742  typedef struct {                                    // Table 2:76
743      TPMI_ALG_HASH           hashAlg;
744      TPMU_HA                 digest;
745  } TPMT_HA;                                          /* Structure */
746  typedef union {                                     // Table 2:77
747      struct {
748          UINT16              size;
749          BYTE                buffer[sizeof(TPMU_HA)];
750      }           t;
751      TPM2B       b;
752  } TPM2B_DIGEST;                                     /* Structure */
753  typedef union {                                     // Table 2:78
754      struct {
755          UINT16              size;
756          BYTE                buffer[sizeof(TPMT_HA)];
757      }           t;
758      TPM2B       b;
759  } TPM2B_DATA;                                       /* Structure */
```

Table 2:79 - Definition of Types for TPM2B_NONCE

```
760  typedef TPM2B_DIGEST        TPM2B_NONCE;
```


Table 2:80 - Definition of Types for TPM2B_AUTH

```
typedef TPM2B_DIGEST        TPM2B_AUTH;
```

Table 2:81 - Definition of Types for TPM2B_OPERAND

```
typedef TPM2B_DIGEST        TPM2B_OPERAND;
typedef union {                                 // Table 2:82
    struct {
        UINT16              size;
        BYTE                buffer[1024];
    }           t;
    TPM2B       b;
} TPM2B_EVENT;                                  /* Structure */
typedef union {                                 // Table 2:83
    struct {
        UINT16              size;
        BYTE                buffer[MAX_DIGEST_BUFFER];
    }           t;
    TPM2B       b;
} TPM2B_MAX_BUFFER;                             /* Structure */
typedef union {                                 // Table 2:84
    struct {
        UINT16              size;
        BYTE                buffer[MAX_NV_BUFFER_SIZE];
    }           t;
    TPM2B       b;
} TPM2B_MAX_NV_BUFFER;                          /* Structure */
typedef union {                                 // Table 2:85
    struct {
        UINT16              size;
        BYTE                buffer[sizeof(UINT64)];
    }           t;
    TPM2B       b;
} TPM2B_TIMEOUT;                                /* Structure */
typedef union {                                 // Table 2:86
    struct {
        UINT16              size;
        BYTE                buffer[MAX_SYM_BLOCK_SIZE];
    }           t;
    TPM2B       b;
} TPM2B_IV;                                     /* Structure */
typedef union {                                 // Table 2:87
    TPMT_HA                 digest;
    TPM_HANDLE              handle;
} TPMU_NAME;                                    /* Structure */
typedef union {                                 // Table 2:88
    struct {
        UINT16              size;
        BYTE                name[sizeof(TPMU_NAME)];
    }           t;
    TPM2B       b;
} TPM2B_NAME;                                   /* Structure */
typedef struct {                                // Table 2:89
    UINT8                   sizeofSelect;
    BYTE                    pcrSelect[PCR_SELECT_MAX];
} TPMS_PCR_SELECT;                              /* Structure */
typedef struct {                                // Table 2:90
    TPMI_ALG_HASH           hash;
    UINT8                   sizeofSelect;
    BYTE                    pcrSelect[PCR_SELECT_MAX];
} TPMS_PCR_SELECTION;                           /* Structure */
typedef struct {                                // Table 2:93
    TPM_ST                  tag;
    TPMI_RH_HIERARCHY       hierarchy;
```

```
821         TPM2B_DIGEST            digest;
822     } TPMT_TK_CREATION;                                 /* Structure */
823     typedef struct {                                    // Table 2:94
824         TPM_ST                  tag;
825         TPMI_RH_HIERARCHY       hierarchy;
826         TPM2B_DIGEST            digest;
827     } TPMT_TK_VERIFIED;                                 /* Structure */
828     typedef struct {                                    // Table 2:95
829         TPM_ST                  tag;
830         TPMI_RH_HIERARCHY       hierarchy;
831         TPM2B_DIGEST            digest;
832     } TPMT_TK_AUTH;                                     /* Structure */
833     typedef struct {                                    // Table 2:96
834         TPM_ST                  tag;
835         TPMI_RH_HIERARCHY       hierarchy;
836         TPM2B_DIGEST            digest;
837     } TPMT_TK_HASHCHECK;                                /* Structure */
838     typedef struct {                                    // Table 2:97
839         TPM_ALG_ID              alg;
840         TPMA_ALGORITHM          algProperties;
841     } TPMS_ALG_PROPERTY;                                /* Structure */
842     typedef struct {                                    // Table 2:98
843         TPM_PT                  property;
844         UINT32                  value;
845     } TPMS_TAGGED_PROPERTY;                             /* Structure */
846     typedef struct {                                    // Table 2:99
847         TPM_PT_PCR              tag;
848         UINT8                   sizeofSelect;
849         BYTE                    pcrSelect[PCR_SELECT_MAX];
850     } TPMS_TAGGED_PCR_SELECT;                           /* Structure */
851     typedef struct {                                    // Table 2:100
852         TPM_HANDLE              handle;
853         TPMT_HA                 policyHash;
854     } TPMS_TAGGED_POLICY;                               /* Structure */
855     typedef struct {                                    // Table 2:101
856         UINT32                  count;
857         TPM_CC                  commandCodes[MAX_CAP_CC];
858     } TPML_CC;                                          /* Structure */
859     typedef struct {                                    // Table 2:102
860         UINT32                  count;
861         TPMA_CC                 commandAttributes[MAX_CAP_CC];
862     } TPML_CCA;                                         /* Structure */
863     typedef struct {                                    // Table 2:103
864         UINT32                  count;
865         TPM_ALG_ID              algorithms[MAX_ALG_LIST_SIZE];
866     } TPML_ALG;                                         /* Structure */
867     typedef struct {                                    // Table 2:104
868         UINT32                  count;
869         TPM_HANDLE              handle[MAX_CAP_HANDLES];
870     } TPML_HANDLE;                                      /* Structure */
871     typedef struct {                                    // Table 2:105
872         UINT32                  count;
873         TPM2B_DIGEST            digests[8];
874     } TPML_DIGEST;                                      /* Structure */
875     typedef struct {                                    // Table 2:106
876         UINT32                  count;
877         TPMT_HA                 digests[HASH_COUNT];
878     } TPML_DIGEST_VALUES;                               /* Structure */
879     typedef struct {                                    // Table 2:107
880         UINT32                  count;
881         TPMS_PCR_SELECTION      pcrSelections[HASH_COUNT];
882     } TPML_PCR_SELECTION;                               /* Structure */
883     typedef struct {                                    // Table 2:108
884         UINT32                  count;
885         TPMS_ALG_PROPERTY       algProperties[MAX_CAP_ALGS];
886     } TPML_ALG_PROPERTY;                                /* Structure */
```



```
887  typedef struct {                                    // Table 2:109
888      UINT32                      count;
889      TPMS_TAGGED_PROPERTY        tpmProperty[MAX_TPM_PROPERTIES];
890  } TPML_TAGGED_TPM_PROPERTY;                         /* Structure */
891  typedef struct {                                    // Table 2:110
892      UINT32                      count;
893      TPMS_TAGGED_PCR_SELECT      pcrProperty[MAX_PCR_PROPERTIES];
894  } TPML_TAGGED_PCR_PROPERTY;                         /* Structure */
895  typedef struct {                                    // Table 2:111
896      UINT32                  count;
897      TPM_ECC_CURVE           eccCurves[MAX_ECC_CURVES];
898  } TPML_ECC_CURVE;                                   /* Structure */
899  typedef struct {                                    // Table 2:112
900      UINT32                  count;
901      TPMS_TAGGED_POLICY      policies[MAX_TAGGED_POLICIES];
902  } TPML_TAGGED_POLICY;                               /* Structure */
903  typedef union {                                     // Table 2:113
904      TPML_ALG_PROPERTY               algorithms;
905      TPML_HANDLE                     handles;
906      TPML_CCA                        command;
907      TPML_CC                         ppCommands;
908      TPML_CC                         auditCommands;
909      TPML_PCR_SELECTION              assignedPCR;
910      TPML_TAGGED_TPM_PROPERTY        tpmProperties;
911      TPML_TAGGED_PCR_PROPERTY        pcrProperties;
912  #if       ALG_ECC
913      TPML_ECC_CURVE                  eccCurves;
914  #endif // ALG_ECC
915      TPML_TAGGED_POLICY              authPolicies;
916  } TPMU_CAPABILITIES;                                /* Structure */
917  typedef struct {                                    // Table 2:114
918      TPM_CAP                 capability;
919      TPMU_CAPABILITIES       data;
920  } TPMS_CAPABILITY_DATA;                             /* Structure */
921  typedef struct {                                    // Table 2:115
922      UINT64                  clock;
923      UINT32                  resetCount;
924      UINT32                  restartCount;
925      TPMI_YES_NO             safe;
926  } TPMS_CLOCK_INFO;                                  /* Structure */
927  typedef struct {                                    // Table 2:116
928      UINT64                  time;
929      TPMS_CLOCK_INFO         clockInfo;
930  } TPMS_TIME_INFO;                                   /* Structure */
931  typedef struct {                                    // Table 2:117
932      TPMS_TIME_INFO          time;
933      UINT64                  firmwareVersion;
934  } TPMS_TIME_ATTEST_INFO;                            /* Structure */
935  typedef struct {                                    // Table 2:118
936      TPM2B_NAME              name;
937      TPM2B_NAME              qualifiedName;
938  } TPMS_CERTIFY_INFO;                                /* Structure */
939  typedef struct {                                    // Table 2:119
940      TPML_PCR_SELECTION      pcrSelect;
941      TPM2B_DIGEST            pcrDigest;
942  } TPMS_QUOTE_INFO;                                  /* Structure */
943  typedef struct {                                    // Table 2:120
944      UINT64                  auditCounter;
945      TPM_ALG_ID              digestAlg;
946      TPM2B_DIGEST            auditDigest;
947      TPM2B_DIGEST            commandDigest;
948  } TPMS_COMMAND_AUDIT_INFO;                          /* Structure */
949  typedef struct {                                    // Table 2:121
950      TPMI_YES_NO             exclusiveSession;
951      TPM2B_DIGEST            sessionDigest;
952  } TPMS_SESSION_AUDIT_INFO;                          /* Structure */
```

```
953  typedef struct {                                    // Table 2:122
954      TPM2B_NAME              objectName;
955      TPM2B_DIGEST            creationHash;
956  } TPMS_CREATION_INFO;                               /* Structure */
957  typedef struct {                                    // Table 2:123
958      TPM2B_NAME                  indexName;
959      UINT16                      offset;
960      TPM2B_MAX_NV_BUFFER         nvContents;
961  } TPMS_NV_CERTIFY_INFO;                             /* Structure */
962  typedef TPM_ST              TPMI_ST_ATTEST;         // Table 2:124  /* Interface */
963  typedef union {                                     // Table 2:125
964      TPMS_CERTIFY_INFO               certify;
965      TPMS_CREATION_INFO              creation;
966      TPMS_QUOTE_INFO                 quote;
967      TPMS_COMMAND_AUDIT_INFO         commandAudit;
968      TPMS_SESSION_AUDIT_INFO         sessionAudit;
969      TPMS_TIME_ATTEST_INFO           time;
970      TPMS_NV_CERTIFY_INFO            nv;
971  } TPMU_ATTEST;                                      /* Structure */
972  typedef struct {                                    // Table 2:126
973      TPM_GENERATED           magic;
974      TPMI_ST_ATTEST          type;
975      TPM2B_NAME              qualifiedSigner;
976      TPM2B_DATA              extraData;
977      TPMS_CLOCK_INFO         clockInfo;
978      UINT64                  firmwareVersion;
979      TPMU_ATTEST             attested;
980  } TPMS_ATTEST;                                      /* Structure */
981  typedef union {                                     // Table 2:127
982      struct {
983          UINT16              size;
984          BYTE                attestationData[sizeof(TPMS_ATTEST)];
985      }           t;
986      TPM2B       b;
987  } TPM2B_ATTEST;                                     /* Structure */
988  typedef struct {                                    // Table 2:128
989      TPMI_SH_AUTH_SESSION        sessionHandle;
990      TPM2B_NONCE                 nonce;
991      TPMA_SESSION                sessionAttributes;
992      TPM2B_AUTH                  hmac;
993  } TPMS_AUTH_COMMAND;                                /* Structure */
994  typedef struct {                                    // Table 2:129
995      TPM2B_NONCE             nonce;
996      TPMA_SESSION            sessionAttributes;
997      TPM2B_AUTH              hmac;
998  } TPMS_AUTH_RESPONSE;                               /* Structure */
999  typedef TPM_KEY_BITS        TPMI_TDES_KEY_BITS;     // Table 2:130  /* Interface */
1000 typedef TPM_KEY_BITS        TPMI_AES_KEY_BITS;      // Table 2:130  /* Interface */
1001 typedef TPM_KEY_BITS        TPMI_SM4_KEY_BITS;      // Table 2:130  /* Interface */
1002 typedef TPM_KEY_BITS        TPMI_CAMELLIA_KEY_BITS; // Table 2:130  /* Interface */
1003 typedef union {                                     // Table 2:131
1004 #if       ALG_TDES
1005     TPMI_TDES_KEY_BITS          tdes;
1006 #endif // ALG_TDES
1007 #if       ALG_AES
1008     TPMI_AES_KEY_BITS           aes;
1009 #endif // ALG_AES
1010 #if       ALG_SM4
1011     TPMI_SM4_KEY_BITS           sm4;
1012 #endif // ALG_SM4
1013 #if       ALG_CAMELLIA
1014     TPMI_CAMELLIA_KEY_BITS      camellia;
1015 #endif // ALG_CAMELLIA
1016     TPM_KEY_BITS                sym;
1017 #if       ALG_XOR
1018     TPMI_ALG_HASH               xor;
```



```
1019  #endif // ALG_XOR
1020  } TPMU_SYM_KEY_BITS;                                    /* Structure */
1021  typedef union {                                         // Table 2:132
1022  #if       ALG_TDES
1023      TPMI_ALG_SYM_MODE       tdes;
1024  #endif // ALG_TDES
1025  #if       ALG_AES
1026      TPMI_ALG_SYM_MODE       aes;
1027  #endif // ALG_AES
1028  #if       ALG_SM4
1029      TPMI_ALG_SYM_MODE       sm4;
1030  #endif // ALG_SM4
1031  #if       ALG_CAMELLIA
1032      TPMI_ALG_SYM_MODE       camellia;
1033  #endif // ALG_CAMELLIA
1034      TPMI_ALG_SYM_MODE       sym;
1035  } TPMU_SYM_MODE;                                        /* Structure */
1036  typedef struct {                                        // Table 2:134
1037      TPMI_ALG_SYM            algorithm;
1038      TPMU_SYM_KEY_BITS       keyBits;
1039      TPMU_SYM_MODE           mode;
1040  } TPMT_SYM_DEF;                                         /* Structure */
1041  typedef struct {                                        // Table 2:135
1042      TPMI_ALG_SYM_OBJECT         algorithm;
1043      TPMU_SYM_KEY_BITS           keyBits;
1044      TPMU_SYM_MODE               mode;
1045  } TPMT_SYM_DEF_OBJECT;                                  /* Structure */
1046  typedef union {                                         // Table 2:136
1047      struct {
1048          UINT16              size;
1049          BYTE                buffer[MAX_SYM_KEY_BYTES];
1050      }           t;
1051      TPM2B       b;
1052  } TPM2B_SYM_KEY;                                        /* Structure */
1053  typedef struct {                                        // Table 2:137
1054      TPMT_SYM_DEF_OBJECT         sym;
1055  } TPMS_SYMCIPHER_PARMS;                                 /* Structure */
1056  typedef union {                                         // Table 2:138
1057      struct {
1058          UINT16              size;
1059          BYTE                buffer[LABEL_MAX_BUFFER];
1060      }           t;
1061      TPM2B       b;
1062  } TPM2B_LABEL;                                          /* Structure */
1063  typedef struct {                                        // Table 2:139
1064      TPM2B_LABEL             label;
1065      TPM2B_LABEL             context;
1066  } TPMS_DERIVE;                                          /* Structure */
1067  typedef union {                                         // Table 2:140
1068      struct {
1069          UINT16              size;
1070          BYTE                buffer[sizeof(TPMS_DERIVE)];
1071      }           t;
1072      TPM2B       b;
1073  } TPM2B_DERIVE;                                         /* Structure */
1074  typedef union {                                         // Table 2:141
1075      BYTE                    create[MAX_SYM_DATA];
1076      TPMS_DERIVE             derive;
1077  } TPMU_SENSITIVE_CREATE;                                /* Structure */
1078  typedef union {                                         // Table 2:142
1079      struct {
1080          UINT16              size;
1081          BYTE                buffer[sizeof(TPMU_SENSITIVE_CREATE)];
1082      }           t;
1083      TPM2B       b;
1084  } TPM2B_SENSITIVE_DATA;                                 /* Structure */
```



```
typedef struct {                                    // Table 2:143
    TPM2B_AUTH                  userAuth;
    TPM2B_SENSITIVE_DATA        data;
} TPMS_SENSITIVE_CREATE;                            /* Structure */
typedef struct {                                    // Table 2:144
    UINT16                      size;
    TPMS_SENSITIVE_CREATE       sensitive;
} TPM2B_SENSITIVE_CREATE;                           /* Structure */
typedef struct {                                    // Table 2:145
    TPMI_ALG_HASH           hashAlg;
} TPMS_SCHEME_HASH;                                 /* Structure */
typedef struct {                                    // Table 2:146
    TPMI_ALG_HASH           hashAlg;
    UINT16                  count;
} TPMS_SCHEME_ECDAA;                                /* Structure */
typedef TPM_ALG_ID          TPMI_ALG_KEYEDHASH_SCHEME;
```

Table 2:148 - Definition of Types for HMAC_SIG_SCHEME

```
typedef TPMS_SCHEME_HASH    TPMS_SCHEME_HMAC;
typedef struct {                                    // Table 2:149
    TPMI_ALG_HASH           hashAlg;
    TPMI_ALG_KDF            kdf;
} TPMS_SCHEME_XOR;                                  /* Structure */
typedef union {                                     // Table 2:150
#if       ALG_HMAC
    TPMS_SCHEME_HMAC        hmac;
#endif // ALG_HMAC
#if       ALG_XOR
    TPMS_SCHEME_XOR         xor;
#endif // ALG_XOR
} TPMU_SCHEME_KEYEDHASH;                            /* Structure */
typedef struct {                                    // Table 2:151
    TPMI_ALG_KEYEDHASH_SCHEME       scheme;
    TPMU_SCHEME_KEYEDHASH           details;
} TPMT_KEYEDHASH_SCHEME;                            /* Structure */
```

Table 2:152 - Definition of Types for RSA Signature Schemes

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

Table 2:153 - Definition of Types for ECC Signature Schemes

```
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;
typedef union {                                     // Table 2:154
#if       ALG_ECC
    TPMS_SIG_SCHEME_ECDAA           ecdaa;
#endif // ALG_ECC
#if       ALG_RSASSA
    TPMS_SIG_SCHEME_RSASSA          rsassa;
#endif // ALG_RSASSA
#if       ALG_RSAPSS
    TPMS_SIG_SCHEME_RSAPSS          rsapss;
#endif // ALG_RSAPSS
#if       ALG_ECDSA
    TPMS_SIG_SCHEME_ECDSA           ecdsa;
#endif // ALG_ECDSA
#if       ALG_SM2
    TPMS_SIG_SCHEME_SM2             sm2;
#endif // ALG_SM2
```



```
1140    #if         ALG_ECSCHNORR
1141        TPMS_SIG_SCHEME_ECSCHNORR       ecschnorr;
1142    #endif // ALG_ECSCHNORR
1143    #if         ALG_HMAC
1144        TPMS_SCHEME_HMAC                hmac;
1145    #endif // ALG_HMAC
1146        TPMS_SCHEME_HASH                any;
1147    } TPMU_SIG_SCHEME;                                  /* Structure */
1148    typedef struct {                                    // Table 2:155
1149        TPMI_ALG_SIG_SCHEME         scheme;
1150        TPMU_SIG_SCHEME             details;
1151    } TPMT_SIG_SCHEME;                                  /* Structure */
```

Table 2:156 - Definition of Types for Encryption Schemes

```
1152    typedef TPMS_SCHEME_HASH    TPMS_ENC_SCHEME_OAEP;
1153    typedef TPMS_EMPTY          TPMS_ENC_SCHEME_RSAES;
```

Table 2:157 - Definition of Types for ECC Key Exchange

```
1154    typedef TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECDH;
1155    typedef TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECMQV;
```

Table 2:158 - Definition of Types for KDF Schemes

```
1156    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_MGF1;
1157    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF1_SP800_56A;
1158    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF2;
1159    typedef TPMS_SCHEME_HASH    TPMS_SCHEME_KDF1_SP800_108;
1160    typedef union {                                     // Table 2:159
1161    #if         ALG_MGF1
1162        TPMS_SCHEME_MGF1                mgf1;
1163    #endif // ALG_MGF1
1164    #if         ALG_KDF1_SP800_56A
1165        TPMS_SCHEME_KDF1_SP800_56A      kdf1_sp800_56a;
1166    #endif // ALG_KDF1_SP800_56A
1167    #if         ALG_KDF2
1168        TPMS_SCHEME_KDF2                kdf2;
1169    #endif // ALG_KDF2
1170    #if         ALG_KDF1_SP800_108
1171        TPMS_SCHEME_KDF1_SP800_108      kdf1_sp800_108;
1172    #endif // ALG_KDF1_SP800_108
1173    } TPMU_KDF_SCHEME;                                  /* Structure */
1174    typedef struct {                                    // Table 2:160
1175        TPMI_ALG_KDF            scheme;
1176        TPMU_KDF_SCHEME         details;
1177    } TPMT_KDF_SCHEME;                                  /* Structure */
1178    typedef TPM_ALG_ID          TPMI_ALG_ASYM_SCHEME;   // Table 2:161  /* Interface */
1179    typedef union {                                     // Table 2:162
1180    #if         ALG_ECDH
1181        TPMS_KEY_SCHEME_ECDH            ecdh;
1182    #endif // ALG_ECDH
1183    #if         ALG_ECMQV
1184        TPMS_KEY_SCHEME_ECMQV           ecmqv;
1185    #endif // ALG_ECMQV
1186    #if         ALG_ECC
1187        TPMS_SIG_SCHEME_ECDAA           ecdaa;
1188    #endif // ALG_ECC
1189    #if         ALG_RSASSA
1190        TPMS_SIG_SCHEME_RSASSA          rsassa;
1191    #endif // ALG_RSASSA
1192    #if         ALG_RSAPSS
1193        TPMS_SIG_SCHEME_RSAPSS          rsapss;
1194    #endif // ALG_RSAPSS
```



```
1195  #if       ALG_ECDSA
1196      TPMS_SIG_SCHEME_ECDSA           ecdsa;
1197  #endif // ALG_ECDSA
1198  #if       ALG_SM2
1199      TPMS_SIG_SCHEME_SM2             sm2;
1200  #endif // ALG_SM2
1201  #if       ALG_ECSCHNORR
1202      TPMS_SIG_SCHEME_ECSCHNORR       ecschnorr;
1203  #endif // ALG_ECSCHNORR
1204  #if       ALG_RSAES
1205      TPMS_ENC_SCHEME_RSAES           rsaes;
1206  #endif // ALG_RSAES
1207  #if       ALG_OAEP
1208      TPMS_ENC_SCHEME_OAEP            oaep;
1209  #endif // ALG_OAEP
1210      TPMS_SCHEME_HASH                anySig;
1211  } TPMU_ASYM_SCHEME;                                 /* Structure */
1212  typedef struct {                                    // Table 2:163
1213      TPMI_ALG_ASYM_SCHEME        scheme;
1214      TPMU_ASYM_SCHEME            details;
1215  } TPMT_ASYM_SCHEME;                                 /* Structure */
1216  typedef TPM_ALG_ID          TPMI_ALG_RSA_SCHEME;    // Table 2:164  /* Interface */
1217  typedef struct {                                    // Table 2:165
1218      TPMI_ALG_RSA_SCHEME         scheme;
1219      TPMU_ASYM_SCHEME            details;
1220  } TPMT_RSA_SCHEME;                                  /* Structure */
1221  typedef TPM_ALG_ID          TPMI_ALG_RSA_DECRYPT;   // Table 2:166  /* Interface */
1222  typedef struct {                                    // Table 2:167
1223      TPMI_ALG_RSA_DECRYPT        scheme;
1224      TPMU_ASYM_SCHEME            details;
1225  } TPMT_RSA_DECRYPT;                                 /* Structure */
1226  typedef union {                                     // Table 2:168
1227      struct {
1228          UINT16              size;
1229          BYTE                buffer[MAX_RSA_KEY_BYTES];
1230      }           t;
1231      TPM2B       b;
1232  } TPM2B_PUBLIC_KEY_RSA;                             /* Structure */
1233  typedef TPM_KEY_BITS        TPMI_RSA_KEY_BITS;      // Table 2:169  /* Interface */
1234  typedef union {                                     // Table 2:170
1235      struct {
1236          UINT16              size;
1237          BYTE                buffer[MAX_RSA_KEY_BYTES/2];
1238      }           t;
1239      TPM2B       b;
1240  } TPM2B_PRIVATE_KEY_RSA;                            /* Structure */
1241  typedef union {                                     // Table 2:171
1242      struct {
1243          UINT16              size;
1244          BYTE                buffer[MAX_ECC_KEY_BYTES];
1245      }           t;
1246      TPM2B       b;
1247  } TPM2B_ECC_PARAMETER;                              /* Structure */
1248  typedef struct {                                    // Table 2:172
1249      TPM2B_ECC_PARAMETER         x;
1250      TPM2B_ECC_PARAMETER         y;
1251  } TPMS_ECC_POINT;                                   /* Structure */
1252  typedef struct {                                    // Table 2:173
1253      UINT16                  size;
1254      TPMS_ECC_POINT          point;
1255  } TPM2B_ECC_POINT;                                  /* Structure */
1256  typedef TPM_ALG_ID          TPMI_ALG_ECC_SCHEME;    // Table 2:174  /* Interface */
1257  typedef TPM_ECC_CURVE       TPMI_ECC_CURVE;         // Table 2:175  /* Interface */
1258  typedef struct {                                    // Table 2:176
1259      TPMI_ALG_ECC_SCHEME         scheme;
1260      TPMU_ASYM_SCHEME            details;
```



```
1261    } TPMT_ECC_SCHEME;                                  /* Structure */
1262    typedef struct {                                    // Table 2:177
1263        TPM_ECC_CURVE               curveID;
1264        UINT16                      keySize;
1265        TPMT_KDF_SCHEME             kdf;
1266        TPMT_ECC_SCHEME             sign;
1267        TPM2B_ECC_PARAMETER         p;
1268        TPM2B_ECC_PARAMETER         a;
1269        TPM2B_ECC_PARAMETER         b;
1270        TPM2B_ECC_PARAMETER         gX;
1271        TPM2B_ECC_PARAMETER         gY;
1272        TPM2B_ECC_PARAMETER         n;
1273        TPM2B_ECC_PARAMETER         h;
1274    } TPMS_ALGORITHM_DETAIL_ECC;                        /* Structure */
1275    typedef struct {                                    // Table 2:178
1276        TPMI_ALG_HASH               hash;
1277        TPM2B_PUBLIC_KEY_RSA        sig;
1278    } TPMS_SIGNATURE_RSA;                               /* Structure */
```

Table 2:179 - Definition of Types for Signature

```
1279    typedef TPMS_SIGNATURE_RSA  TPMS_SIGNATURE_RSASSA;
1280    typedef TPMS_SIGNATURE_RSA  TPMS_SIGNATURE_RSAPSS;
1281    typedef struct {                                    // Table 2:180
1282        TPMI_ALG_HASH               hash;
1283        TPM2B_ECC_PARAMETER         signatureR;
1284        TPM2B_ECC_PARAMETER         signatureS;
1285    } TPMS_SIGNATURE_ECC;                               /* Structure */
```

Table 2:181 - Definition of Types for TPMS_SIGNATURE_ECC

```
1286    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECDAA;
1287    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECDSA;
1288    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_SM2;
1289    typedef TPMS_SIGNATURE_ECC  TPMS_SIGNATURE_ECSCHNORR;
1290    typedef union {                                     // Table 2:182
1291    #if       ALG_ECC
1292        TPMS_SIGNATURE_ECDAA            ecdaa;
1293    #endif // ALG_ECC
1294    #if       ALG_RSA
1295        TPMS_SIGNATURE_RSASSA           rsassa;
1296    #endif // ALG_RSA
1297    #if       ALG_RSA
1298        TPMS_SIGNATURE_RSAPSS           rsapss;
1299    #endif // ALG_RSA
1300    #if       ALG_ECC
1301        TPMS_SIGNATURE_ECDSA            ecdsa;
1302    #endif // ALG_ECC
1303    #if       ALG_ECC
1304        TPMS_SIGNATURE_SM2              sm2;
1305    #endif // ALG_ECC
1306    #if       ALG_ECC
1307        TPMS_SIGNATURE_ECSCHNORR        ecschnorr;
1308    #endif // ALG_ECC
1309    #if       ALG_HMAC
1310        TPMT_HA                         hmac;
1311    #endif // ALG_HMAC
1312        TPMS_SCHEME_HASH                any;
1313    } TPMU_SIGNATURE;                                   /* Structure */
1314    typedef struct {                                    // Table 2:183
1315        TPMI_ALG_SIG_SCHEME         sigAlg;
1316        TPMU_SIGNATURE              signature;
1317    } TPMT_SIGNATURE;                                   /* Structure */
1318    typedef union {                                     // Table 2:184
1319    #if       ALG_ECC
```



```
1320        BYTE                    ecc[sizeof(TPMS_ECC_POINT)];
1321    #endif // ALG_ECC
1322    #if       ALG_RSA
1323        BYTE                    rsa[MAX_RSA_KEY_BYTES];
1324    #endif // ALG_RSA
1325    #if       ALG_SYMCIPHER
1326        BYTE                    symmetric[sizeof(TPM2B_DIGEST)];
1327    #endif // ALG_SYMCIPHER
1328    #if       ALG_KEYEDHASH
1329        BYTE                    keyedHash[sizeof(TPM2B_DIGEST)];
1330    #endif // ALG_KEYEDHASH
1331    } TPMU_ENCRYPTED_SECRET;                            /* Structure */
1332    typedef union {                                     // Table 2:185
1333        struct {
1334            UINT16              size;
1335            BYTE                secret[sizeof(TPMU_ENCRYPTED_SECRET)];
1336        }           t;
1337        TPM2B       b;
1338    } TPM2B_ENCRYPTED_SECRET;                           /* Structure */
1339    typedef TPM_ALG_ID          TPMI_ALG_PUBLIC;        // Table 2:186  /* Interface */
1340    typedef union {                                     // Table 2:187
1341    #if       ALG_KEYEDHASH
1342        TPM2B_DIGEST                keyedHash;
1343    #endif // ALG_KEYEDHASH
1344    #if       ALG_SYMCIPHER
1345        TPM2B_DIGEST                sym;
1346    #endif // ALG_SYMCIPHER
1347    #if       ALG_RSA
1348        TPM2B_PUBLIC_KEY_RSA        rsa;
1349    #endif // ALG_RSA
1350    #if       ALG_ECC
1351        TPMS_ECC_POINT              ecc;
1352    #endif // ALG_ECC
1353        TPMS_DERIVE                 derive;
1354    } TPMU_PUBLIC_ID;                                   /* Structure */
1355    typedef struct {                                    // Table 2:188
1356        TPMT_KEYEDHASH_SCHEME       scheme;
1357    } TPMS_KEYEDHASH_PARMS;                             /* Structure */
1358    typedef struct {                                    // Table 2:189
1359        TPMT_SYM_DEF_OBJECT         symmetric;
1360        TPMT_ASYM_SCHEME            scheme;
1361    } TPMS_ASYM_PARMS;                                  /* Structure */
1362    typedef struct {                                    // Table 2:190
1363        TPMT_SYM_DEF_OBJECT         symmetric;
1364        TPMT_RSA_SCHEME             scheme;
1365        TPMI_RSA_KEY_BITS           keyBits;
1366        UINT32                      exponent;
1367    } TPMS_RSA_PARMS;                                   /* Structure */
1368    typedef struct {                                    // Table 2:191
1369        TPMT_SYM_DEF_OBJECT         symmetric;
1370        TPMT_ECC_SCHEME             scheme;
1371        TPMI_ECC_CURVE              curveID;
1372        TPMT_KDF_SCHEME             kdf;
1373    } TPMS_ECC_PARMS;                                   /* Structure */
1374    typedef union {                                     // Table 2:192
1375    #if       ALG_KEYEDHASH
1376        TPMS_KEYEDHASH_PARMS        keyedHashDetail;
1377    #endif // ALG_KEYEDHASH
1378    #if       ALG_SYMCIPHER
1379        TPMS_SYMCIPHER_PARMS        symDetail;
1380    #endif // ALG_SYMCIPHER
1381    #if       ALG_RSA
1382        TPMS_RSA_PARMS              rsaDetail;
1383    #endif // ALG_RSA
1384    #if       ALG_ECC
1385        TPMS_ECC_PARMS              eccDetail;
```



```
1386  #endif // ALG_ECC
1387      TPMS_ASYM_PARMS             asymDetail;
1388  } TPMU_PUBLIC_PARMS;                                    /* Structure */
1389  typedef struct {                                        // Table 2:193
1390      TPMI_ALG_PUBLIC         type;
1391      TPMU_PUBLIC_PARMS       parameters;
1392  } TPMT_PUBLIC_PARMS;                                    /* Structure */
1393  typedef struct {                                        // Table 2:194
1394      TPMI_ALG_PUBLIC         type;
1395      TPMI_ALG_HASH           nameAlg;
1396      TPMA_OBJECT             objectAttributes;
1397      TPM2B_DIGEST            authPolicy;
1398      TPMU_PUBLIC_PARMS       parameters;
1399      TPMU_PUBLIC_ID          unique;
1400  } TPMT_PUBLIC;                                          /* Structure */
1401  typedef struct {                                        // Table 2:195
1402      UINT16                  size;
1403      TPMT_PUBLIC             publicArea;
1404  } TPM2B_PUBLIC;                                         /* Structure */
1405  typedef union {                                         // Table 2:196
1406      struct {
1407          UINT16              size;
1408          BYTE                buffer[sizeof(TPMT_PUBLIC)];
1409      }           t;
1410      TPM2B       b;
1411  } TPM2B_TEMPLATE;                                       /* Structure */
1412  typedef union {                                         // Table 2:197
1413      struct {
1414          UINT16              size;
1415          BYTE                buffer[PRIVATE_VENDOR_SPECIFIC_BYTES];
1416      }           t;
1417      TPM2B       b;
1418  } TPM2B_PRIVATE_VENDOR_SPECIFIC;                        /* Structure */
1419  typedef union {                                         // Table 2:198
1420  #if       ALG_RSA
1421      TPM2B_PRIVATE_KEY_RSA               rsa;
1422  #endif // ALG_RSA
1423  #if       ALG_ECC
1424      TPM2B_ECC_PARAMETER                 ecc;
1425  #endif // ALG_ECC
1426  #if       ALG_KEYEDHASH
1427      TPM2B_SENSITIVE_DATA                bits;
1428  #endif // ALG_KEYEDHASH
1429  #if       ALG_SYMCIPHER
1430      TPM2B_SYM_KEY                       sym;
1431  #endif // ALG_SYMCIPHER
1432      TPM2B_PRIVATE_VENDOR_SPECIFIC       any;
1433  } TPMU_SENSITIVE_COMPOSITE;                             /* Structure */
1434  typedef struct {                                        // Table 2:199
1435      TPMI_ALG_PUBLIC             sensitiveType;
1436      TPM2B_AUTH                  authValue;
1437      TPM2B_DIGEST                seedValue;
1438      TPMU_SENSITIVE_COMPOSITE    sensitive;
1439  } TPMT_SENSITIVE;                                       /* Structure */
1440  typedef struct {                                        // Table 2:200
1441      UINT16                  size;
1442      TPMT_SENSITIVE          sensitiveArea;
1443  } TPM2B_SENSITIVE;                                      /* Structure */
1444  typedef struct {                                        // Table 2:201
1445      TPM2B_DIGEST            integrityOuter;
1446      TPM2B_DIGEST            integrityInner;
1447      TPM2B_SENSITIVE         sensitive;
1448  } _PRIVATE;                                             /* Structure */
1449  typedef union {                                         // Table 2:202
1450      struct {
1451          UINT16              size;
```

```
1452            BYTE                buffer[sizeof(_PRIVATE)];
1453        }           t;
1454        TPM2B       b;
1455    } TPM2B_PRIVATE;                                    /* Structure */
1456    typedef struct {                                    // Table 2:203
1457        TPM2B_DIGEST            integrityHMAC;
1458        TPM2B_DIGEST            encIdentity;
1459    } TPMS_ID_OBJECT;                                   /* Structure */
1460    typedef union {                                     // Table 2:204
1461        struct {
1462            UINT16              size;
1463            BYTE                credential[sizeof(TPMS_ID_OBJECT)];
1464        }           t;
1465        TPM2B       b;
1466    } TPM2B_ID_OBJECT;                                  /* Structure */
1467    #if USE_BIT_FIELD_STRUCTURES
1468    typedef struct TPM_NV_INDEX {                       // Table 2:205
1469        unsigned    index               : 24;
1470        unsigned    RH_NV               : 8;
1471    } TPM_NV_INDEX;                                     /* Bits */
```

This is the initializer for a TPM_NV_INDEX structure

```
1472    #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) {index, rh_nv}
1473    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:205 TPM_NV_INDEX using bit masking

```
1474    typedef UINT32                      TPM_NV_INDEX;
1475    #define TPM_NV_INDEX_index_SHIFT    0
1476    #define TPM_NV_INDEX_index          ((TPM_NV_INDEX)0xffffff << 0)
1477    #define TPM_NV_INDEX_RH_NV_SHIFT    24
1478    #define TPM_NV_INDEX_RH_NV          ((TPM_NV_INDEX)0xff << 24)
```

This is the initializer for a TPM_NV_INDEX bit array.

```
1479    #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) ((index << 0) + (rh_nv << 24))
1480    #endif // USE_BIT_FIELD_STRUCTURES
```

Table 2:206 - Definition of TPM_NT Constants

```
1481    typedef UINT32              TPM_NT;
1482    #define TPM_NT_ORDINARY     (TPM_NT)(0x0)
1483    #define TPM_NT_COUNTER      (TPM_NT)(0x1)
1484    #define TPM_NT_BITS         (TPM_NT)(0x2)
1485    #define TPM_NT_EXTEND       (TPM_NT)(0x4)
1486    #define TPM_NT_PIN_FAIL     (TPM_NT)(0x8)
1487    #define TPM_NT_PIN_PASS     (TPM_NT)(0x9)
1488    typedef struct {                                    // Table 2:207
1489        UINT32                  pinCount;
1490        UINT32                  pinLimit;
1491    } TPMS_NV_PIN_COUNTER_PARAMETERS;                   /* Structure */
1492    #if USE_BIT_FIELD_STRUCTURES
1493    typedef struct TPMA_NV {                            // Table 2:208
1494        unsigned    PPWRITE             : 1;
1495        unsigned    OWNERWRITE          : 1;
1496        unsigned    AUTHWRITE           : 1;
1497        unsigned    POLICYWRITE         : 1;
1498        unsigned    TPM_NT              : 4;
1499        unsigned    Reserved_bits_at_8  : 2;
1500        unsigned    POLICY_DELETE       : 1;
1501        unsigned    WRITELOCKED         : 1;
1502        unsigned    WRITEALL            : 1;
1503        unsigned    WRITEDEFINE         : 1;
```



```
1504        unsigned    WRITE_STCLEAR        : 1;
1505        unsigned    GLOBALLOCK           : 1;
1506        unsigned    PPREAD               : 1;
1507        unsigned    OWNERREAD            : 1;
1508        unsigned    AUTHREAD             : 1;
1509        unsigned    POLICYREAD           : 1;
1510        unsigned    Reserved_bits_at_20  : 5;
1511        unsigned    NO_DA                : 1;
1512        unsigned    ORDERLY              : 1;
1513        unsigned    CLEAR_STCLEAR        : 1;
1514        unsigned    READLOCKED           : 1;
1515        unsigned    WRITTEN              : 1;
1516        unsigned    PLATFORMCREATE       : 1;
1517        unsigned    READ_STCLEAR         : 1;
1518    } TPMA_NV;                                          /* Bits */
```

This is the initializer for a TPMA_NV structure

```
1519    #define TPMA_NV_INITIALIZER(                                                    \
1520                ppwrite,        ownerwrite,     authwrite,      policywrite,        \
1521                tpm_nt,         bits_at_8,      policy_delete,  writelocked,        \
1522                writeall,       writedefine,    write_stclear,  globallock,         \
1523                ppread,         ownerread,      authread,       policyread,         \
1524                bits_at_20,     no_da,          orderly,        clear_stclear,      \
1525                readlocked,     written,        platformcreate, read_stclear)       \
1526               {ppwrite,        ownerwrite,     authwrite,      policywrite,        \
1527                tpm_nt,         bits_at_8,      policy_delete,  writelocked,        \
1528                writeall,       writedefine,    write_stclear,  globallock,         \
1529                ppread,         ownerread,      authread,       policyread,         \
1530                bits_at_20,     no_da,          orderly,        clear_stclear,      \
1531                readlocked,     written,        platformcreate, read_stclear}
1532    #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:208 TPMA_NV using bit masking

```
1533    typedef UINT32                  TPMA_NV;
1534    #define TPMA_NV_PPWRITE         ((TPMA_NV)1 << 0)
1535    #define TPMA_NV_OWNERWRITE      ((TPMA_NV)1 << 1)
1536    #define TPMA_NV_AUTHWRITE       ((TPMA_NV)1 << 2)
1537    #define TPMA_NV_POLICYWRITE     ((TPMA_NV)1 << 3)
1538    #define TPMA_NV_TPM_NT_SHIFT    4
1539    #define TPMA_NV_TPM_NT          ((TPMA_NV)0xf << 4)
1540    #define TPMA_NV_POLICY_DELETE   ((TPMA_NV)1 << 10)
1541    #define TPMA_NV_WRITELOCKED     ((TPMA_NV)1 << 11)
1542    #define TPMA_NV_WRITEALL        ((TPMA_NV)1 << 12)
1543    #define TPMA_NV_WRITEDEFINE     ((TPMA_NV)1 << 13)
1544    #define TPMA_NV_WRITE_STCLEAR   ((TPMA_NV)1 << 14)
1545    #define TPMA_NV_GLOBALLOCK      ((TPMA_NV)1 << 15)
1546    #define TPMA_NV_PPREAD          ((TPMA_NV)1 << 16)
1547    #define TPMA_NV_OWNERREAD       ((TPMA_NV)1 << 17)
1548    #define TPMA_NV_AUTHREAD        ((TPMA_NV)1 << 18)
1549    #define TPMA_NV_POLICYREAD      ((TPMA_NV)1 << 19)
1550    #define TPMA_NV_NO_DA           ((TPMA_NV)1 << 25)
1551    #define TPMA_NV_ORDERLY         ((TPMA_NV)1 << 26)
1552    #define TPMA_NV_CLEAR_STCLEAR   ((TPMA_NV)1 << 27)
1553    #define TPMA_NV_READLOCKED      ((TPMA_NV)1 << 28)
1554    #define TPMA_NV_WRITTEN         ((TPMA_NV)1 << 29)
1555    #define TPMA_NV_PLATFORMCREATE  ((TPMA_NV)1 << 30)
1556    #define TPMA_NV_READ_STCLEAR    ((TPMA_NV)1 << 31)
```

This is the initializer for a TPMA_NV bit array.

```
1557    #define TPMA_NV_INITIALIZER(                                                    \
1558                ppwrite,        ownerwrite,     authwrite,      policywrite,        \
```

```
1559                tpm_nt,          bits_at_8,       policy_delete,  writelocked,          \
1560                writeall,        writedefine,     write_stclear,  globallock,           \
1561                ppread,          ownerread,       authread,       policyread,           \
1562                bits_at_20,      no_da,           orderly,        clear_stclear,        \
1563                readlocked,      written,         platformcreate, read_stclear)         \
1564               ((ppwrite << 0)          + (ownerwrite << 1)      +                       \
1565                (authwrite << 2)        + (policywrite << 3)     +                       \
1566                (tpm_nt << 4)           + (policy_delete << 10)  +                       \
1567                (writelocked << 11)     + (writeall << 12)       +                       \
1568                (writedefine << 13)     + (write_stclear << 14)  +                       \
1569                (globallock << 15)      + (ppread << 16)         +                       \
1570                (ownerread << 17)       + (authread << 18)       +                       \
1571                (policyread << 19)      + (no_da << 25)          +                       \
1572                (orderly << 26)         + (clear_stclear << 27)  +                       \
1573                (readlocked << 28)      + (written << 29)        +                       \
1574                (platformcreate << 30) + (read_stclear << 31))
1575   #endif // USE_BIT_FIELD_STRUCTURES
1576   typedef struct {                                          // Table 2:209
1577       TPMI_RH_NV_INDEX        nvIndex;
1578       TPMI_ALG_HASH           nameAlg;
1579       TPMA_NV                 attributes;
1580       TPM2B_DIGEST            authPolicy;
1581       UINT16                  dataSize;
1582   } TPMS_NV_PUBLIC;                                         /* Structure */
1583   typedef struct {                                          // Table 2:210
1584       UINT16                  size;
1585       TPMS_NV_PUBLIC          nvPublic;
1586   } TPM2B_NV_PUBLIC;                                        /* Structure */
1587   typedef union {                                           // Table 2:211
1588       struct {
1589           UINT16              size;
1590           BYTE                buffer[MAX_CONTEXT_SIZE];
1591       }            t;
1592       TPM2B        b;
1593   } TPM2B_CONTEXT_SENSITIVE;                                /* Structure */
1594   typedef struct {                                          // Table 2:212
1595       TPM2B_DIGEST                integrity;
1596       TPM2B_CONTEXT_SENSITIVE     encrypted;
1597   } TPMS_CONTEXT_DATA;                                      /* Structure */
1598   typedef union {                                           // Table 2:213
1599       struct {
1600           UINT16              size;
1601           BYTE                buffer[sizeof(TPMS_CONTEXT_DATA)];
1602       }            t;
1603       TPM2B        b;
1604   } TPM2B_CONTEXT_DATA;                                     /* Structure */
1605   typedef struct {                                          // Table 2:214
1606       UINT64                  sequence;
1607       TPMI_DH_SAVED           savedHandle;
1608       TPMI_RH_HIERARCHY       hierarchy;
1609       TPM2B_CONTEXT_DATA      contextBlob;
1610   } TPMS_CONTEXT;                                           /* Structure */
1611   typedef struct {                                          // Table 2:216
1612       TPML_PCR_SELECTION      pcrSelect;
1613       TPM2B_DIGEST            pcrDigest;
1614       TPMA_LOCALITY           locality;
1615       TPM_ALG_ID              parentNameAlg;
1616       TPM2B_NAME              parentName;
1617       TPM2B_NAME              parentQualifiedName;
1618       TPM2B_DATA              outsideInfo;
1619   } TPMS_CREATION_DATA;                                     /* Structure */
1620   typedef struct {                                          // Table 2:217
1621       UINT16                  size;
1622       TPMS_CREATION_DATA      creationData;
1623   } TPM2B_CREATION_DATA;                                    /* Structure */
```


Table 2:218 - Definition of TPM_AT Constants

```
typedef UINT32              TPM_AT;
#define TPM_AT_ANY          (TPM_AT)(0x00000000)
#define TPM_AT_ERROR        (TPM_AT)(0x00000001)
#define TPM_AT_PV1          (TPM_AT)(0x00000002)
#define TPM_AT_VEND         (TPM_AT)(0x80000000)
```

Table 2:219 - Definition of TPM_AE Constants

```
typedef UINT32              TPM_AE;
#define TPM_AE_NONE         (TPM_AE)(0x00000000)
typedef struct {                                    // Table 2:220
    TPM_AT                  tag;
    UINT32                  data;
} TPMS_AC_OUTPUT;                                   /* Structure */
typedef struct {                                    // Table 2:221
    UINT32                  count;
    TPMS_AC_OUTPUT          acCapabilities[MAX_AC_CAPABILITIES];
} TPML_AC_CAPABILITIES;                             /* Structure */
#endif    //  _TPM_TYPES_H_
```

5.22 VendorString.h

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

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

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

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

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

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

```
#define       VENDOR_STRING_1       "xCG "
#define       VENDOR_STRING_2       "fTPM"
// #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
```

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

```
#define   FIRMWARE_V1         (0x20170619)
```

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

```
#define   FIRMWARE_V2         (0x00163636)
```

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

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


5.23 swap.h

```
1   #ifndef _SWAP_H
2   #define _SWAP_H
3   #if LITTLE_ENDIAN_TPM
4   #define TO_BIG_ENDIAN_UINT16(i)     REVERSE_ENDIAN_16(i)
5   #define FROM_BIG_ENDIAN_UINT16(i)   REVERSE_ENDIAN_16(i)
6   #define TO_BIG_ENDIAN_UINT32(i)     REVERSE_ENDIAN_32(i)
7   #define FROM_BIG_ENDIAN_UINT32(i)   REVERSE_ENDIAN_32(i)
8   #define TO_BIG_ENDIAN_UINT64(i)     REVERSE_ENDIAN_64(i)
9   #define FROM_BIG_ENDIAN_UINT64(i)   REVERSE_ENDIAN_64(i)
10  #else
11  #define TO_BIG_ENDIAN_UINT16(i)     (i)
12  #define FROM_BIG_ENDIAN_UINT16(i)   (i)
13  #define TO_BIG_ENDIAN_UINT32(i)     (i)
14  #define FROM_BIG_ENDIAN_UINT32(i)   (i)
15  #define TO_BIG_ENDIAN_UINT64(i)     (i)
16  #define FROM_BIG_ENDIAN_UINT64(i)   (i)
17  #endif
18  #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

```
19  #define BYTE_ARRAY_TO_UINT8(b)   (uint8_t)((b)[0])
20  #define BYTE_ARRAY_TO_UINT16(b) ByteArrayToUint16((BYTE *)(b))
21  #define BYTE_ARRAY_TO_UINT32(b) ByteArrayToUint32((BYTE *)(b))
22  #define BYTE_ARRAY_TO_UINT64(b) ByteArrayToUint64((BYTE *)(b))
23  #define UINT8_TO_BYTE_ARRAY(i, b) ((b)[0] = (uint8_t)(i))
24  #define UINT16_TO_BYTE_ARRAY(i, b)  Uint16ToByteArray((i), (BYTE *)(b))
25  #define UINT32_TO_BYTE_ARRAY(i, b)  Uint32ToByteArray((i), (BYTE *)(b))
26  #define UINT64_TO_BYTE_ARRAY(i, b)  Uint64ToByteArray((i), (BYTE *)(b))
27  #else // AUTO_ALIGN
28  #if BIG_ENDIAN_TPM
```

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

```
29  #define BYTE_ARRAY_TO_UINT8(b)        *((uint8_t  *)(b))
30  #define BYTE_ARRAY_TO_UINT16(b)       *((uint16_t *)(b))
31  #define BYTE_ARRAY_TO_UINT32(b)       *((uint32_t *)(b))
32  #define BYTE_ARRAY_TO_UINT64(b)       *((uint64_t *)(b))
```

Disaggregate a UINT into a byte array

```
33  #define UINT8_TO_BYTE_ARRAY(i, b)   {*((uint8_t  *)(b)) = (i);}
34  #define UINT16_TO_BYTE_ARRAY(i, b)  {*((uint16_t *)(b)) = (i);}
35  #define UINT32_TO_BYTE_ARRAY(i, b)  {*((uint32_t *)(b)) = (i);}
36  #define UINT64_TO_BYTE_ARRAY(i, b)  {*((uint64_t *)(b)) = (i);}
37  #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

```
38  #define BYTE_ARRAY_TO_UINT8(b)        *((uint8_t  *)(b))
39  #define BYTE_ARRAY_TO_UINT16(b)       REVERSE_ENDIAN_16(*((uint16_t *)(b)))
40  #define BYTE_ARRAY_TO_UINT32(b)       REVERSE_ENDIAN_32(*((uint32_t *)(b)))
41  #define BYTE_ARRAY_TO_UINT64(b)       REVERSE_ENDIAN_64(*((uint64_t *)(b)))
```

Disaggregate a UINT into a byte array

```
42  #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

```
#include "Tpm.h"
#include "ExecCommand_fp.h"
```

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

```
// #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().

```
4   LIB_EXPORT void
5   ExecuteCommand(
6       uint32_t          requestSize,    // IN: command buffer size
7       unsigned char    *request,        // IN: command buffer
8       uint32_t         *responseSize,   // IN/OUT: response buffer size
9       unsigned char    **response       // IN/OUT: response buffer
10      )
11  {
12      // Command local variables
13      UINT32                commandSize;
14      COMMAND               command;
15
16      // Response local variables
17      UINT32                maxResponse = *responseSize;
18      TPM_RC                result;            // return code for the command
19
20  // This next function call is used in development to size the command and response
21  // buffers. The values printed are the sizes of the internal structures and
22  // not the sizes of the canonical forms of the command response structures. Also,
23  // the sizes do not include the tag, command.code, requestSize, or the authorization
24  // fields.
25  //CommandResponseSizes();
26      // Set flags for NV access state. This should happen before any other
27      // operation that may require a NV write. Note, that this needs to be done
```



```
28      // even when in failure mode. Otherwise, g_updateNV would stay SET while in
29      // Failure mode and the NV would be written on each call.
30      g_updateNV = UT_NONE;
31      g_clearOrderly = FALSE;
32      if(g_inFailureMode)
33      {
34          // Do failure mode processing
35          TpmFailureMode(requestSize, request, responseSize, response);
36          return;
37      }
38      // Query platform to get the NV state.  The result state is saved internally
39      // and will be reported by NvIsAvailable(). The reference code requires that
40      // accessibility of NV does not change during the execution of a command.
41      // Specifically, if NV is available when the command execution starts and then
42      // is not available later when it is necessary to write to NV, then the TPM
43      // will go into failure mode.
44      NvCheckState();
45
46      // Due to the limitations of the simulation, TPM clock must be explicitly
47      // synchronized with the system clock whenever a command is received.
48      // This function call is not necessary in a hardware TPM. However, taking
49      // a snapshot of the hardware timer at the beginning of the command allows
50      // the time value to be consistent for the duration of the command execution.
51      TimeUpdateToCurrent();
52
53      // Any command through this function will unceremoniously end the
54      // _TPM_Hash_Data/_TPM_Hash_End sequence.
55      if(g_DRTMHandle != TPM_RH_UNASSIGNED)
56          ObjectTerminateEvent();
57
58      // Get command buffer size and command buffer.
59      command.parameterBuffer = request;
60      command.parameterSize = requestSize;
61
62      // Parse command header: tag, commandSize and command.code.
63      // First parse the tag. The unmarshaling routine will validate
64      // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
65      result = TPMI_ST_COMMAND_TAG_Unmarshal(&command.tag,
66                                             &command.parameterBuffer,
67                                             &command.parameterSize);
68      if(result != TPM_RC_SUCCESS)
69          goto Cleanup;
70      // Unmarshal the commandSize indicator.
71      result = UINT32_Unmarshal(&commandSize,
72                                &command.parameterBuffer,
73                                &command.parameterSize);
74      if(result != TPM_RC_SUCCESS)
75          goto Cleanup;
76      // On a TPM that receives bytes on a port, the number of bytes that were
77      // received on that port is requestSize it must be identical to commandSize.
78      // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
79      // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
80      // as the input processing (the function that receives the command bytes and
81      // places them in the input buffer) would likely have the input truncated when
82      // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
83      if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
84      {
85          result = TPM_RC_COMMAND_SIZE;
86          goto Cleanup;
87      }
88      // Unmarshal the command code.
89      result = TPM_CC_Unmarshal(&command.code, &command.parameterBuffer,
90                                &command.parameterSize);
91      if(result != TPM_RC_SUCCESS)
92          goto Cleanup;
93      // Check to see if the command is implemented.
```

```
 94         command.index = CommandCodeToCommandIndex(command.code);
 95         if(UNIMPLEMENTED_COMMAND_INDEX == command.index)
 96         {
 97             result = TPM_RC_COMMAND_CODE;
 98             goto Cleanup;
 99         }
100     #if  FIELD_UPGRADE_IMPLEMENTED  == YES
101         // If the TPM is in FUM, then the only allowed command is
102         // TPM_CC_FieldUpgradeData.
103         if(IsFieldUgradeMode() && (command.code != TPM_CC_FieldUpgradeData))
104         {
105             result = TPM_RC_UPGRADE;
106             goto Cleanup;
107         }
108         else
109     #endif
110         // Excepting FUM, the TPM only accepts TPM2_Startup() after
111         // _TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
112         // is no longer allowed.
113             if((!TPMIsStarted() && command.code != TPM_CC_Startup)
114                || (TPMIsStarted() && command.code == TPM_CC_Startup))
115             {
116                 result = TPM_RC_INITIALIZE;
117                 goto Cleanup;
118             }
119     // Start regular command process.
120         NvIndexCacheInit();
121         // Parse Handle buffer.
122         result = ParseHandleBuffer(&command);
123         if(result != TPM_RC_SUCCESS)
124             goto Cleanup;
125         // All handles in the handle area are required to reference TPM-resident
126         // entities.
127         result = EntityGetLoadStatus(&command);
128         if(result != TPM_RC_SUCCESS)
129             goto Cleanup;
130         // Authorization session handling for the command.
131         ClearCpRpHashes(&command);
132         if(command.tag == TPM_ST_SESSIONS)
133         {
134             // Find out session buffer size.
135             result = UINT32_Unmarshal((UINT32 *)&command.authSize,
136                                       &command.parameterBuffer,
137                                       &command.parameterSize);
138             if(result != TPM_RC_SUCCESS)
139                 goto Cleanup;
140             // Perform sanity check on the unmarshaled value. If it is smaller than
141             // the smallest possible session or larger than the remaining size of
142             // the command, then it is an error. NOTE: This check could pass but the
143             // session size could still be wrong. That will be determined after the
144             // sessions are unmarshaled.
145             if(command.authSize < 9
146                || command.authSize > command.parameterSize)
147             {
148                 result = TPM_RC_SIZE;
149                 goto Cleanup;
150             }
151             command.parameterSize -= command.authSize;
152
153             // The actions of ParseSessionBuffer() are described in the introduction.
154             // As the sessions are parsed command.parameterBuffer is advanced so, on a
155             // successful return, command.parameterBuffer should be pointing at the
156             // first byte of the parameters.
157             result = ParseSessionBuffer(&command);
158             if(result != TPM_RC_SUCCESS)
159                 goto Cleanup;
```



```
160         }
161         else
162         {
163             command.authSize = 0;
164             // The command has no authorization sessions.
165             // If the command requires authorizations, then CheckAuthNoSession() will
166             // return an error.
167             result = CheckAuthNoSession(&command);
168             if(result != TPM_RC_SUCCESS)
169                 goto Cleanup;
170         }
171         // Set up the response buffer pointers. CommandDispatch will marshal the
172         // response parameters starting at the address in command.responseBuffer.
173     //    *response = MemoryGetResponseBuffer(command.index);
174         // leave space for the command header
175         command.responseBuffer = *response + STD_RESPONSE_HEADER;
176
177         // leave space for the parameter size field if needed
178         if(command.tag == TPM_ST_SESSIONS)
179             command.responseBuffer += sizeof(UINT32);
180         if(IsHandleInResponse(command.index))
181             command.responseBuffer += sizeof(TPM_HANDLE);
182
183         // CommandDispatcher returns a response handle buffer and a response parameter
184         // buffer if it succeeds. It will also set the parameterSize field in the
185         // buffer if the tag is TPM_RC_SESSIONS.
186         result = CommandDispatcher(&command);
187         if(result != TPM_RC_SUCCESS)
188             goto Cleanup;
189
190         // Build the session area at the end of the parameter area.
191         BuildResponseSession(&command);
192
193     Cleanup:
194         if(g_clearOrderly == TRUE
195             && NV_IS_ORDERLY)
196         {
197     #if USE_DA_USED
198             gp.orderlyState = g_daUsed ? SU_DA_USED_VALUE : SU_NONE_VALUE;
199     #else
200             gp.orderlyState = SU_NONE_VALUE;
201     #endif
202             NV_SYNC_PERSISTENT(orderlyState);
203         }
204         // This implementation loads an "evict" object to a transient object slot in
205         // RAM whenever an "evict" object handle is used in a command so that the
206         // access to any object is the same. These temporary objects need to be
207         // cleared from RAM whether the command succeeds or fails.
208         ObjectCleanupEvict();
209
210         // The parameters and sessions have been marshaled. Now tack on the header and
211         // set the sizes
212         BuildResponseHeader(&command, *response, result);
213
214         // Try to commit all the writes to NV if any NV write happened during this
215         // command execution. This check should be made for both succeeded and failed
216         // commands, because a failed one may trigger a NV write in DA logic as well.
217         // This is the only place in the command execution path that may call the NV
218         // commit. If the NV commit fails, the TPM should be put in failure mode.
219         if((g_updateNV != UT_NONE) && !g_inFailureMode)
220         {
221             if(g_updateNV == UT_ORDERLY)
222                 NvUpdateIndexOrderlyData();
223             if(!NvCommit())
224                 FAIL(FATAL_ERROR_INTERNAL);
225             g_updateNV = UT_NONE;
```

```
226         }
227         pAssert((UINT32)command.parameterSize <= maxResponse);
228
229         // Clear unused bits in response buffer.
230         MemorySet(*response + *responseSize, 0, maxResponse - *responseSize);
231
232         // as a final act, and not before, update the response size.
233         *responseSize = (UINT32)command.parameterSize;
234
235         return;
236     }
```


6.3 CommandDispatcher.c

6.3.1 Introduction

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;

NOTE 1 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 2 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 3 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.

6.3.1.1 Includes and Typedefs

```
#include "Tpm.h"
#if 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.

```
typedef struct
{
    COMMAND_t       command;        // Address of the command
    UINT16          inSize;         // Maximum size of the input structure
    UINT16          outSize;        // Maximum size of the output structure
    UINT16          typesOffset;    // address of the types field
    UINT16          offsets[1];
} 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:

0	unmarshaling for handles that do not take flags
HANDLE_FIRST_FLAG_TYPE	unmarshaling for handles that take flags
PARAMETER_FIRST_TYPE	unmarshaling for parameters that do not take flags
PARAMETER_FIRST_FLAG_TYPE	unmarshaling for parameters that take flags
PARAMETER_LAST_TYPE + 1	marshaling for handles
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.

```
#if 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
```

6.3.1.2 Marshal/Unmarshal Functions

6.3.1.2.1 ParseHandleBuffer()

This is the table-driven version of the handle buffer unmarshaling code

```
TPM_RC
ParseHandleBuffer(
    COMMAND                 *command
    )
{
    TPM_RC                   result;
#if 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;
```

```
65
66      // Get the first type value
67      for(type = *types++;
68          // check each byte to make sure that we have not hit the start
69          // of the parameters
70      (dtype = (type & 0x7F)) < PARAMETER_FIRST_TYPE;
71      // get the next type
72          type = *types++)
73      {
74          // See if unmarshaling of this handle type requires a flag
75          if(dtype < HANDLE_FIRST_FLAG_TYPE)
76          {
77              // Look up the function to do the unmarshaling
78              NoFlagFunction  *f = (NoFlagFunction *)UnmarshalArray[dtype];
79              // call it
80              result = f(&(command->handles[command->handleNum]),
81                         &command->parameterBuffer,
82                         &command->parameterSize);
83          }
84          else
85          {
86              //  Look up the function
87              FlagFunction    *f = UnmarshalArray[dtype];
88
89              // Call it setting the flag to the appropriate value
90              result = f(&(command->handles[command->handleNum]),
91                         &command->parameterBuffer,
92                         &command->parameterSize, (type & 0x80) != 0);
93          }
94          // Got a handle
95          // We do this first so that the match for the handle offset of the
96          // response code works correctly.
97          command->handleNum += 1;
98          if(result != TPM_RC_SUCCESS)
99              // if the unmarshaling failed, return the response code with the
100             // handle indication set
101             return result + TPM_RC_H + (command->handleNum * TPM_RC_1);
102     }
103 #else
104     BYTE            **handleBufferStart = &command->parameterBuffer;
105     INT32           *bufferRemainingSize = &command->parameterSize;
106     TPM_HANDLE      *handles = &command->handles[0];
107     UINT32          *handleCount = &command->handleNum;
108     *handleCount = 0;
109     switch(command->code)
110     {
111 #include "HandleProcess.h"
112 #undef handles
113         default:
114             FAIL(FATAL_ERROR_INTERNAL);
115             break;
116     }
117 #endif
118     return TPM_RC_SUCCESS;
119 }
```

6.3.1.2.2 CommandDispatcher()

Function to unmarshal the command parameters, call the selected action code, and marshal the response parameters.

```
120 TPM_RC
121 CommandDispatcher(
122     COMMAND                 *command
```



```
123         )
124     {
125     #if !TABLE_DRIVEN_DISPATCH
126         TPM_RC        result;
127         BYTE         **paramBuffer = &command->parameterBuffer;
128         INT32        *paramBufferSize = &command->parameterSize;
129         BYTE         **responseBuffer = &command->responseBuffer;
130         INT32        *respParmSize = &command->parameterSize;
131         INT32         rSize;
132         TPM_HANDLE   *handles = &command->handles[0];
133     //
134         command->handleNum = 0;                 // The command-specific code knows how
135                                                 // many handles there are. This is for
136                                                 // cataloging the number of response
137                                                 // handles
138         MemoryIoBufferAllocationReset();        // Initialize so that allocation will
139                                                 // work properly
140         switch(GetCommandCode(command->index))
141         {
142     #include "CommandDispatcher.h"
143
144             default:
145                 FAIL(FATAL_ERROR_INTERNAL);
146                 break;
147         }
148     Exit:
149         MemoryIoBufferZero();
150         return result;
151     #else
152         COMMAND_DESCRIPTOR_t    *desc;
153         BYTE                    *types;
154         BYTE                     type;
155         UINT16                  *offsets;
156         UINT16                   offset = 0;
157         UINT32                   maxInSize;
158         BYTE                    *commandIn;
159         INT32                    maxOutSize;
160         BYTE                    *commandOut;
161         COMMAND_t                cmd;
162         TPM_HANDLE              *handles;
163         UINT32                   hasInParameters = 0;
164         BOOL                     hasOutParameters = FALSE;
165         UINT32                   pNum = 0;
166         BYTE                     dType;     // dispatch type
167         TPM_RC                   result;
168     //
169         // Get the address of the descriptor for this command
170         pAssert(command->index
171                 < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
172         desc = s_CommandDataArray[command->index];
173
174         // Get the list of parameter types for this command
175         pAssert(desc != NULL);
176         types = &((BYTE *)desc)[desc->typesOffset];
177
178         // Get a pointer to the list of parameter offsets
179         offsets = &desc->offsets[0];
180         // pointer to handles
181         handles = command->handles;
182
183         // Get the size required to hold all the unmarshaled parameters for this command
184         maxInSize = desc->inSize;
185         // and the size of the output parameter structure returned by this command
186         maxOutSize = desc->outSize;
187
188         MemoryIoBufferAllocationReset();
```

```
189     // Get a buffer for the input parameters
190     commandIn = MemoryGetInBuffer(maxInSize);
191     // And the output parameters
192     commandOut = (BYTE *)MemoryGetOutBuffer((UINT32)maxOutSize);
193
194     // Get the address of the action code dispatch
195     cmd = desc->command;
196
197     // Copy any handles into the input buffer
198     for(type = *types++; (type & 0x7F) < PARAMETER_FIRST_TYPE; type = *types++)
199     {
200         // 'offset' was initialized to zero so the first unmarshaling will always
201         // be to the start of the data structure
202         *(TPM_HANDLE *)&(commandIn[offset]) = *handles++;
203         // This check is used so that we don't have to add an additional offset
204         // value to the offsets list to correspond to the stop value in the
205         // command parameter list.
206         if(*types != 0xFF)
207             offset = *offsets++;
208 //        maxInSize -= sizeof(TPM_HANDLE);
209         hasInParameters++;
210     }
211     // Exit loop with type containing the last value read from types
212     // maxInSize has the amount of space remaining in the command action input
213     // buffer. Make sure that we don't have more data to unmarshal than is going to
214     // fit.
215
216     // type contains the last value read from types so it is not necessary to
217     // reload it, which is good because *types now points to the next value
218     for(; (dType = (type & 0x7F)) <= PARAMETER_LAST_TYPE; type = *types++)
219     {
220         pNum++;
221         if(dType < PARAMETER_FIRST_FLAG_TYPE)
222         {
223             NoFlagFunction      *f = (NoFlagFunction *)UnmarshalArray[dType];
224             result = f(&commandIn[offset], &command->parameterBuffer,
225                        &command->parameterSize);
226         }
227         else
228         {
229             FlagFunction        *f = UnmarshalArray[dType];
230             result = f(&commandIn[offset], &command->parameterBuffer,
231                        &command->parameterSize,
232                        (type & 0x80) != 0);
233         }
234         if(result != TPM_RC_SUCCESS)
235         {
236             result += TPM_RC_P + (TPM_RC_1 * pNum);
237             goto Exit;
238         }
239
240         // This check is used so that we don't have to add an additional offset
241         // value to the offsets list to correspond to the stop value in the
242         // command parameter list.
243         if(*types != 0xFF)
244             offset = *offsets++;
245         hasInParameters++;
246     }
247     // Should have used all the bytes in the input
248     if(command->parameterSize != 0)
249     {
250         result = TPM_RC_SIZE;
251         goto Exit;
252     }
253
254     // The command parameter unmarshaling stopped when it hit a value that was out
```



```
255         // of range for unmarshaling values and left *types pointing to the first
256         // marshaling type. If that type happens to be the STOP value, then there
257         // are no response parameters. So, set the flag to indicate if there are
258         // output parameters.
259         hasOutParameters = *types != 0xFF;
260
261         // There are four cases for calling, with and without input parameters and with
262         // and without output parameters.
263         if(hasInParameters > 0)
264         {
265             if(hasOutParameters)
266                 result = cmd.inOutArg(commandIn, commandOut);
267             else
268                 result = cmd.inArg(commandIn);
269         }
270         else
271         {
272             if(hasOutParameters)
273                 result = cmd.outArg(commandOut);
274             else
275                 result = cmd.noArgs();
276         }
277         if(result != TPM_RC_SUCCESS)
278            goto Exit;
279
280         // Offset in the marshaled output structure
281         offset = 0;
282
283         // Process the return handles, if any
284         command->handleNum = 0;
285
286         // Could make this a loop to process output handles but there is only ever
287         // one handle in the outputs (for now).
288         type = *types++;
289         if((dType = (type & 0x7F)) < RESPONSE_PARAMETER_FIRST_TYPE)
290         {
291             // The out->handle value was referenced as TPM_HANDLE in the
292             // action code so it has to be properly aligned.
293             command->handles[command->handleNum++] =
294                 *((TPM_HANDLE *)&(commandOut[offset]));
295             maxOutSize -= sizeof(UINT32);
296             type = *types++;
297             offset = *offsets++;
298         }
299         // Use the size of the command action output buffer as the maximum for the
300         // number of bytes that can get marshaled. Since the marshaling code has
301         // no pointers to data, all of the data being returned has to be in the
302         // command action output buffer. If we try to marshal more bytes than
303         // could fit into the output buffer, we need to fail.
304         for(;(dType = (type & 0x7F)) <= RESPONSE_PARAMETER_LAST_TYPE
305             && !g_inFailureMode; type = *types++)
306         {
307             const MARSHAL_t     f = MarshalArray[dType];
308
309             command->parameterSize += f(&commandOut[offset], &command->responseBuffer,
310                                         &maxOutSize);
311             offset = *offsets++;
312         }
313         result = (maxOutSize < 0) ? TPM_RC_FAILURE : TPM_RC_SUCCESS;
314     Exit:
315         MemoryIoBufferZero();
316         return result;
317     #endif
318     }
```


6.4 SessionProcess.c

6.4.1 Introduction

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

6.4.2 Includes and Data Definitions

```
#define SESSION_PROCESS_C
#include "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(1)	handle is exempted from DA logic
FALSE(0)	handle is not exempted from DA logic

```
BOOL
IsDAExempted(
    TPM_HANDLE        handle          // IN: entity handle
    )
{
    BOOL        result = FALSE;
//
    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            // All permanent handles, other than TPM_RH_LOCKOUT, are exempt from
            // DA protection.
            result = (handle != TPM_RH_LOCKOUT);
            break;
        // When this function is called, a persistent object will have been loaded
        // into an object slot and assigned a transient handle.
        case TPM_HT_TRANSIENT:
        {
            TPMA_OBJECT     attributes = ObjectGetPublicAttributes(handle);
            result = IS_ATTRIBUTE(attributes, TPMA_OBJECT, noDA);
            break;
        }
        case TPM_HT_NV_INDEX:
        {
            NV_INDEX            *nvIndex = NvGetIndexInfo(handle, NULL);
            result = IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, NO_DA);
            break;
        }
```



```
31          case TPM_HT_PCR:
32              // PCRs are always exempted from DA.
33              result = TRUE;
34              break;
35          default:
36              break;
37      }
38      return result;
39  }
```

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

```
40  static TPM_RC
41  IncrementLockout(
42      UINT32            sessionIndex
43      )
44  {
45      TPM_HANDLE        handle = s_associatedHandles[sessionIndex];
46      TPM_HANDLE        sessionHandle = s_sessionHandles[sessionIndex];
47      SESSION          *session = NULL;
48  //
49      // Don't increment lockout unless the handle associated with the session
50      // is DA protected or the session is bound to a DA protected entity.
51      if(sessionHandle == TPM_RS_PW)
52      {
53          if(IsDAExempted(handle))
54              return TPM_RC_BAD_AUTH;
55      }
56      else
57      {
58          session = SessionGet(sessionHandle);
59          // If the session is bound to lockout, then use that as the relevant
60          // handle. This means that an authorization failure with a bound session
61          // bound to lockoutAuth will take precedence over any other
62          // lockout check
63          if(session->attributes.isLockoutBound == SET)
64              handle = TPM_RH_LOCKOUT;
65          if(session->attributes.isDaBound == CLEAR
66             && (IsDAExempted(handle) || session->attributes.includeAuth == CLEAR))
67             // If the handle was changed to TPM_RH_LOCKOUT, this will not return
68             // TPM_RC_BAD_AUTH
69              return TPM_RC_BAD_AUTH;
70      }
71      if(handle == TPM_RH_LOCKOUT)
72      {
73          pAssert(gp.lockOutAuthEnabled == TRUE);
74
75          // lockout is no longer enabled
76          gp.lockOutAuthEnabled = FALSE;
77
78          // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
79          // the lockout authorization will be reset at startup.
80          if(gp.lockoutRecovery != 0)
81          {
```

```
82                  if(NV_IS_AVAILABLE)
83                      // Update NV.
84                      NV_SYNC_PERSISTENT(lockOutAuthEnabled);
85                  else
86                      // No NV access for now. Put the TPM in pending mode.
87                      s_DAPendingOnNV = TRUE;
88              }
89          }
90          else
91          {
92              if(gp.recoveryTime != 0)
93              {
94                  gp.failedTries++;
95                  if(NV_IS_AVAILABLE)
96                      // Record changes to NV. NvWrite will SET g_updateNV
97                      NV_SYNC_PERSISTENT(failedTries);
98                  else
99                      // No NV access for now.  Put the TPM in pending mode.
100                     s_DAPendingOnNV = TRUE;
101             }
102         }
103         // Register a DA failure and reset the timers.
104         DARegisterFailure(handle);
105
106         return TPM_RC_AUTH_FAIL;
107     }
```

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(1)	handle points to the session start entity
FALSE(0)	handle does not point to the session start entity

```
108 static BOOL
109 IsSessionBindEntity(
110     TPM_HANDLE        associatedHandle,  // IN: handle to be authorized
111     SESSION          *session            // IN: associated session
112     )
113 {
114     TPM2B_NAME     entity;             // The bind value for the entity
115 //
116     // If the session is not bound, return FALSE.
117     if(session->attributes.isBound)
118     {
119         // Compute the bind value for the entity.
120         SessionComputeBoundEntity(associatedHandle, &entity);
121
122         // Compare to the bind value in the session.
123         return MemoryEqual2B(&entity.b, &session->u1.boundEntity.b);
124     }
125     return FALSE;
126 }
```


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(1)	policy session is required
FALSE(0)	policy session is not required

```
127 static BOOL
128 IsPolicySessionRequired(
129     COMMAND_INDEX    commandIndex,  // IN: command index
130     UINT32           sessionIndex   // IN: session index
131     )
132 {
133     AUTH_ROLE       role = CommandAuthRole(commandIndex, sessionIndex);
134     TPM_HT          type = HandleGetType(s_associatedHandles[sessionIndex]);
135 //
136     if(role == AUTH_DUP)
137         return TRUE;
138     if(role == AUTH_ADMIN)
139     {
140         // We allow an exception for ADMIN role in a transient object. If the object
141         // allows ADMIN role actions with authorization, then policy is not
142         // required. For all other cases, there is no way to override the command
143         // requirement that a policy be used
144         if(type == TPM_HT_TRANSIENT)
145         {
146             OBJECT      *object = HandleToObject(s_associatedHandles[sessionIndex]);
147
148             if(!IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT,
149                              adminWithPolicy))
150                 return FALSE;
151         }
152         return TRUE;
153     }
154
155     if(type == TPM_HT_PCR)
156     {
157         if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
158         {
159             TPM2B_DIGEST        policy;
160             TPMI_ALG_HASH       policyAlg;
161             policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
162                                          &policy);
163             if(policyAlg != TPM_ALG_NULL)
164                 return TRUE;
165         }
166     }
167     return FALSE;
168 }
```

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(1)	*authValue* is available
FALSE(0)	*authValue* is not available

```
169  static BOOL
170  IsAuthValueAvailable(
171      TPM_HANDLE        handle,         // IN: handle of entity
172      COMMAND_INDEX     commandIndex,   // IN: command index
173      UINT32            sessionIndex    // IN: session index
174      )
175  {
176      BOOL              result = FALSE;
177  //
178      switch(HandleGetType(handle))
179      {
180          case TPM_HT_PERMANENT:
181              switch(handle)
182              {
183                      // At this point hierarchy availability has already been
184                      // checked so primary seed handles are always available here
185                  case TPM_RH_OWNER:
186                  case TPM_RH_ENDORSEMENT:
187                  case TPM_RH_PLATFORM:
188  #ifdef VENDOR_PERMANENT
189                      // This vendor defined handle associated with the
190                      // manufacturer's shared secret
191                  case VENDOR_PERMANENT:
192  #endif
193                      // The DA checking has been performed on LockoutAuth but we
194                      // bypass the DA logic if we are using lockout policy. The
195                      // policy would allow execution to continue an lockoutAuth
196                      // could be used, even if direct use of lockoutAuth is disabled
197                  case TPM_RH_LOCKOUT:
198                      // NullAuth is always available.
199                  case TPM_RH_NULL:
200                      result = TRUE;
201                      break;
202                  default:
203                      // Otherwise authValue is not available.
204                      break;
205              }
206              break;
207          case TPM_HT_TRANSIENT:
208              // A persistent object has already been loaded and the internal
209              // handle changed.
210          {
211              OBJECT          *object;
212              TPMA_OBJECT      attributes;
213  //
214              object = HandleToObject(handle);
215              attributes = object->publicArea.objectAttributes;
216
```



```
217             // authValue is always available for a sequence object.
218             // An alternative for this is to
219             // SET_ATTRIBUTE(object->publicArea, TPMA_OBJECT, userWithAuth) when the
220             // sequence is started.
221             if(ObjectIsSequence(object))
222             {
223                 result = TRUE;
224                 break;
225             }
226             // authValue is available for an object if it has its sensitive
227             // portion loaded and
228             //  1. userWithAuth bit is SET, or
229             //  2. ADMIN role is required
230             if(object->attributes.publicOnly == CLEAR
231                && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, userWithAuth)
232                    || (CommandAuthRole(commandIndex, sessionIndex) == AUTH_ADMIN
233                        && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, adminWithPolicy))))
234                 result = TRUE;
235         }
236         break;
237         case TPM_HT_NV_INDEX:
238             // NV Index.
239         {
240             NV_REF           locator;
241             NV_INDEX        *nvIndex = NvGetIndexInfo(handle, &locator);
242             TPMA_NV          nvAttributes;
243 //
244             pAssert(nvIndex != 0);
245
246             nvAttributes = nvIndex->publicArea.attributes;
247
248             if(IsWriteOperation(commandIndex))
249             {
250                 // AuthWrite can't be set for a PIN index
251                 if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHWRITE))
252                     result = TRUE;
253             }
254             else
255             {
256                 // A "read" operation
257                 // For a PIN Index, the authValue is available as long as the
258                 // Index has been written and the pinCount is less than pinLimit
259                 if(IsNvPinFailIndex(nvAttributes)
260                    || IsNvPinPassIndex(nvAttributes))
261                 {
262                     NV_PIN          pin;
263                     if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
264                         break; // return false
265                     // get the index values
266                     pin.intVal = NvGetUINT64Data(nvIndex, locator);
267                     if(pin.pin.pinCount < pin.pin.pinLimit)
268                         result = TRUE;
269                 }
270                 // For non-PIN Indexes, need to allow use of the authValue
271                 else if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHREAD))
272                     result = TRUE;
273             }
274         }
275         break;
276         case TPM_HT_PCR:
277             // PCR handle.
278             // authValue is always allowed for PCR
279             result = TRUE;
280             break;
281         default:
282             // Otherwise, authValue is not available
```

```
283                 break;
284         }
285         return result;
286     }
```

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(1)	*authPolicy* is available
FALSE(0)	*authPolicy* is not available

```
287 static BOOL
288 IsAuthPolicyAvailable(
289     TPM_HANDLE        handle,         // IN: handle of entity
290     COMMAND_INDEX     commandIndex,   // IN: command index
291     UINT32            sessionIndex    // IN: session index
292     )
293 {
294     BOOL              result = FALSE;
295 //
296     switch(HandleGetType(handle))
297     {
298         case TPM_HT_PERMANENT:
299             switch(handle)
300             {
301                 // At this point hierarchy availability has already been checked.
302                 case TPM_RH_OWNER:
303                     if(gp.ownerPolicy.t.size != 0)
304                         result = TRUE;
305                     break;
306                 case TPM_RH_ENDORSEMENT:
307                     if(gp.endorsementPolicy.t.size != 0)
308                         result = TRUE;
309                     break;
310                 case TPM_RH_PLATFORM:
311                     if(gc.platformPolicy.t.size != 0)
312                         result = TRUE;
313                     break;
314                 case TPM_RH_LOCKOUT:
315                     if(gp.lockoutPolicy.t.size != 0)
316                         result = TRUE;
317                     break;
318                 default:
319                     break;
320             }
321             break;
322         case TPM_HT_TRANSIENT:
323         {
324             // Object handle.
325             // An evict object would already have been loaded and given a
326             // transient object handle by this point.
327             OBJECT  *object = HandleToObject(handle);
328             // Policy authorization is not available for an object with only
329             // public portion loaded.
330             if(object->attributes.publicOnly == CLEAR)
331             {
332                 // Policy authorization is always available for an object but
```



```
333                     // is never available for a sequence.
334                     if(!ObjectIsSequence(object))
335                         result = TRUE;
336                 }
337                 break;
338             }
339             case TPM_HT_NV_INDEX:
340                 // An NV Index.
341             {
342                 NV_INDEX         *nvIndex = NvGetIndexInfo(handle, NULL);
343                 TPMA_NV           nvAttributes = nvIndex->publicArea.attributes;
344 //
345                 // If the policy size is not zero, check if policy can be used.
346                 if(nvIndex->publicArea.authPolicy.t.size != 0)
347                 {
348                     // If policy session is required for this handle, always
349                     // uses policy regardless of the attributes bit setting
350                     if(IsPolicySessionRequired(commandIndex, sessionIndex))
351                         result = TRUE;
352                     // Otherwise, the presence of the policy depends on the NV
353                     // attributes.
354                     else if(IsWriteOperation(commandIndex))
355                     {
356                         if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYWRITE))
357                             result = TRUE;
358                     }
359                     else
360                     {
361                         if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYREAD))
362                             result = TRUE;
363                     }
364                 }
365             }
366             break;
367             case TPM_HT_PCR:
368                 // PCR handle.
369                 if(PCRPolicyIsAvailable(handle))
370                     result = TRUE;
371                 break;
372             default:
373                 break;
374         }
375         return result;
376     }
```

6.4.4 Session Parsing Functions

6.4.4.1 ClearCpRpHashes()

```
377     void
378     ClearCpRpHashes(
379         COMMAND         *command
380         )
381     {
382     #if ALG_SHA1
383         command->sha1CpHash.t.size = 0;
384         command->sha1RpHash.t.size = 0;
385     #endif
386     #if ALG_SHA256
387         command->sha256CpHash.t.size = 0;
388         command->sha256RpHash.t.size = 0;
389     #endif
390     #if ALG_SHA384
391         command->sha384CpHash.t.size = 0;
```

```
392         command->sha384RpHash.t.size = 0;
393     #endif
394     #if ALG_SHA512
395         command->sha512CpHash.t.size = 0;
396         command->sha512RpHash.t.size = 0;
397     #endif
398     #if ALG_SM3_256
399         command->sm3_256CpHash.t.size = 0;
400         command->sm3_256RpHash.t.size = 0;
401     #endif
402     }
```

6.4.4.2 GetCpHashPointer()

Function to get a pointer to the *cpHash* of the command

```
403     static TPM2B_DIGEST *
404     GetCpHashPointer(
405         COMMAND         *command,
406         TPMI_ALG_HASH    hashAlg
407         )
408     {
409         switch(hashAlg)
410         {
411     #if ALG_SHA1
412             case ALG_SHA1_VALUE:
413                 return (TPM2B_DIGEST *)&command->sha1CpHash;
414     #endif
415     #if ALG_SHA256
416             case ALG_SHA256_VALUE:
417                 return (TPM2B_DIGEST *)&command->sha256CpHash;
418     #endif
419     #if ALG_SHA384
420             case ALG_SHA384_VALUE:
421                 return (TPM2B_DIGEST *)&command->sha384CpHash;
422     #endif
423     #if ALG_SHA512
424             case ALG_SHA512_VALUE:
425                 return (TPM2B_DIGEST *)&command->sha512CpHash;
426     #endif
427     #if ALG_SM3_256
428             case ALG_SM3_256_VALUE:
429                 return (TPM2B_DIGEST *)&command->sm3_256CpHash;
430     #endif
431             default:
432                 break;
433         }
434         return NULL;
435     }
```

6.4.4.3 GetRpHashPointer()

Function to get a pointer to the RpHash() of the command

```
436     static TPM2B_DIGEST *
437     GetRpHashPointer(
438         COMMAND         *command,
439         TPMI_ALG_HASH    hashAlg
440         )
441     {
442         switch(hashAlg)
443         {
444     #if ALG_SHA1
```



```
445             case ALG_SHA1_VALUE:
446                 return (TPM2B_DIGEST *)&command->sha1RpHash;
447     #endif
448     #if ALG_SHA256
449             case ALG_SHA256_VALUE:
450                 return (TPM2B_DIGEST *)&command->sha256RpHash;
451     #endif
452     #if ALG_SHA384
453             case ALG_SHA384_VALUE:
454                 return (TPM2B_DIGEST *)&command->sha384RpHash;
455     #endif
456     #if ALG_SHA512
457             case ALG_SHA512_VALUE:
458                 return (TPM2B_DIGEST *)&command->sha512RpHash;
459     #endif
460     #if ALG_SM3_256
461             case ALG_SM3_256_VALUE:
462                 return (TPM2B_DIGEST *)&command->sm3_256RpHash;
463     #endif
464             default:
465                 break;
466         }
467         return NULL;
468     }
```

6.4.4.4 ComputeCpHash()

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

```
469     static TPM2B_DIGEST *
470     ComputeCpHash(
471         COMMAND         *command,       // IN: command parsing structure
472         TPMI_ALG_HASH    hashAlg        // IN: hash algorithm
473         )
474     {
475         UINT32              i;
476         HASH_STATE          hashState;
477         TPM2B_NAME          name;
478         TPM2B_DIGEST       *cpHash;
479     //
480         // cpHash = hash(commandCode [ || authName1
481         //                            [ || authName2
482         //                            [ || authName 3 ]]]
483         //                            [ || parameters])
484         // A cpHash can contain just a commandCode only if the lone session is
485         // an audit session.
486         // Get pointer to the hash value
487         cpHash = GetCpHashPointer(command, hashAlg);
488         if(cpHash->t.size == 0)
489         {
490             cpHash->t.size = CryptHashStart(&hashState, hashAlg);
491                 //  Add commandCode.
492             CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
493                 //  Add authNames for each of the handles.
494             for(i = 0; i < command->handleNum; i++)
495                 CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
496                                                                &name)->b);
497                 //  Add the parameters.
498             CryptDigestUpdate(&hashState, command->parameterSize,
499                               command->parameterBuffer);
500                 //  Complete the hash.
501             CryptHashEnd2B(&hashState, &cpHash->b);
502         }
503         return cpHash;
```



```
}
```

6.4.4.5 GetCpHash()

This function is used to access a precomputed *cpHash*.

```
static TPM2B_DIGEST *
GetCpHash(
    COMMAND         *command,
    TPMI_ALG_HASH    hashAlg
    )
{
    TPM2B_DIGEST        *cpHash = GetCpHashPointer(command, hashAlg);
 //
    pAssert(cpHash->t.size != 0);
    return cpHash;
}
```

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_CreateLoaded()

Return Value	Meaning
TRUE(1)	template hash equal to session->*templateHash*
FALSE(0)	template hash not equal to session->*templateHash*

```
static BOOL
CompareTemplateHash(
    COMMAND         *command,       // IN: parsing structure
    SESSION         *session        // IN: session data
    )
{
    BYTE                *pBuffer = command->parameterBuffer;
    INT32                pSize = command->parameterSize;
    TPM2B_DIGEST         tHash;
    UINT16               size;
//
    // Only try this for the three commands for which it is intended
    if(command->code != TPM_CC_Create
       && command->code != TPM_CC_CreatePrimary
#if CC_CreateLoaded
       && command->code != TPM_CC_CreateLoaded
#endif
       )
        return FALSE;
    // Assume that the first parameter is a TPM2B and unmarshal the size field
    // Note: this will not affect the parameter buffer and size in the calling
    // function.
    if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
        return FALSE;
    // reduce the space in the buffer.
    // NOTE: this could make pSize go negative if the parameters are not correct but
    // the unmarshaling code does not try to unmarshal if the remaining size is
    // negative.
    pSize -= size;

    // Advance the pointer
    pBuffer += size;

```



```
549     // Get the size of what should be the template
550     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
551         return FALSE;
552     // See if this is reasonable
553     if(size > pSize)
554         return FALSE;
555     // Hash the template data
556     tHash.t.size = CryptHashBlock(session->authHashAlg, size, pBuffer,
557                                   sizeof(tHash.t.buffer), tHash.t.buffer);
558     return(MemoryEqual2B(&session->u1.templateHash.b, &tHash.b));
559 }
```

6.4.4.7 CompareNameHash()

This function computes the name hash and compares it to the *nameHash* in the session data.

```
560 BOOL
561 CompareNameHash(
562     COMMAND         *command,       // IN: main parsing structure
563     SESSION         *session        // IN: session structure with nameHash
564     )
565 {
566     HASH_STATE          hashState;
567     TPM2B_DIGEST        nameHash;
568     UINT32              i;
569     TPM2B_NAME          name;
570 //
571     nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
572     //  Add names.
573     for(i = 0; i < command->handleNum; i++)
574         CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
575                                                        &name)->b);
576     //  Complete hash.
577     CryptHashEnd2B(&hashState, &nameHash.b);
578     // and compare
579     return MemoryEqual(session->u1.nameHash.t.buffer, nameHash.t.buffer,
580                        nameHash.t.size);
581 }
```

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

```
582 static TPM_RC
583 CheckPWAuthSession(
584     UINT32          sessionIndex    // IN: index of session to be processed
585     )
586 {
587     TPM2B_AUTH      authValue;
588     TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];
589 //
590     // Strip trailing zeros from the password.
591     MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
592
593     // 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
    )
{
    TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
    TPM2B_KEY        key;
    BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
    BYTE            *buffer;
    UINT32           marshalSize;
    HMAC_STATE       hmacState;
    TPM2B_NONCE     *nonceDecrypt;
    TPM2B_NONCE     *nonceEncrypt;
    SESSION         *session;
//
    nonceDecrypt = NULL;
    nonceEncrypt = NULL;

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



```
653             }
654         }
655
656         // Continue with the HMAC processing.
657         session = SessionGet(s_sessionHandles[sessionIndex]);
658
659         // Generate HMAC key.
660         MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
661
662         // Check if the session has an associated handle and if the associated entity
663         // is the one to which the session is bound. If not, add the authValue of
664         // this entity to the HMAC key.
665         // If the session is bound to the object or the session is a policy session
666         // with no authValue required, do not include the authValue in the HMAC key.
667         // Note: For a policy session, its isBound attribute is CLEARED.
668         //
669         // Include the entity authValue if it is needed
670         if(session->attributes.includeAuth == SET)
671         {
672             TPM2B_AUTH          authValue;
673             // Get the entity authValue with trailing zeros removed
674             EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
675             // add the authValue to the HMAC key
676             MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
677         }
678          // if the HMAC key size is 0, a NULL string HMAC is allowed
679         if(key.t.size == 0
680            && s_inputAuthValues[sessionIndex].t.size == 0)
681         {
682             hmac->t.size = 0;
683             return hmac;
684         }
685         // Start HMAC
686         hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
687
688             //  Add cpHash
689         CryptDigestUpdate2B(&hmacState.hashState,
690                             &ComputeCpHash(command, session->authHashAlg)->b);
691             //  Add nonces as required
692         CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
693         CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
694         if(nonceDecrypt != NULL)
695             CryptDigestUpdate2B(&hmacState.hashState, &nonceDecrypt->b);
696         if(nonceEncrypt != NULL)
697             CryptDigestUpdate2B(&hmacState.hashState, &nonceEncrypt->b);
698             //  Add sessionAttributes
699         buffer = marshalBuffer;
700         marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
701                                            &buffer, NULL);
702         CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
703             // Complete the HMAC computation
704         CryptHmacEnd2B(&hmacState, &hmac->b);
705
706         return hmac;
707     }
```

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

```
static TPM_RC
CheckSessionHMAC(
    COMMAND         *command,       // IN: primary control structure
    UINT32           sessionIndex   // IN: index of session to be processed
    )
{
    TPM2B_DIGEST        hmac;           // authHMAC for comparing
//
    // Compute authHMAC
    ComputeCommandHMAC(command, sessionIndex, &hmac);

    // Compare the input HMAC with the authHMAC computed above.
    if(!MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &hmac.b))
    {
        // If an HMAC session has a failure, invoke the anti-hammering
        // if it applies to the authorized entity or the session.
        // Otherwise, just indicate that the authorization is bad.
        return IncrementLockout(sessionIndex);
    }
    return TPM_RC_SUCCESS;
}
```

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

```
static TPM_RC
CheckPolicyAuthSession(
    COMMAND         *command,       // IN: primary parsing structure
```



```
732     UINT32            sessionIndex   // IN: index of session to be processed
733     )
734 {
735     SESSION             *session;
736     TPM2B_DIGEST         authPolicy;
737     TPMI_ALG_HASH        policyAlg;
738     UINT8                locality;
739 //
740     // Initialize pointer to the authorization session.
741     session = SessionGet(s_sessionHandles[sessionIndex]);
742
743     // If the command is TPM2_PolicySecret(), make sure that
744     // either password or authValue is required
745     if(command->code == TPM_CC_PolicySecret
746        &&  session->attributes.isPasswordNeeded == CLEAR
747        &&  session->attributes.isAuthValueNeeded == CLEAR)
748         return TPM_RC_MODE;
749     // See if the PCR counter for the session is still valid.
750     if(!SessionPCRValueIsCurrent(session))
751         return TPM_RC_PCR_CHANGED;
752     // Get authPolicy.
753     policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
754                                     &authPolicy);
755     // Compare authPolicy.
756     if(!MemoryEqual2B(&session->u2.policyDigest.b, &authPolicy.b))
757         return TPM_RC_POLICY_FAIL;
758     // Policy is OK so check if the other factors are correct
759
760     // Compare policy hash algorithm.
761     if(policyAlg != session->authHashAlg)
762         return TPM_RC_POLICY_FAIL;
763
764     // Compare timeout.
765     if(session->timeout != 0)
766     {
767         // Cannot compare time if clock stop advancing.  An TPM_RC_NV_UNAVAILABLE
768         // or TPM_RC_NV_RATE error may be returned here. This doesn't mean that
769         // a new nonce will be created just that, because TPM time can't advance
770         // we can't do time-based operations.
771         RETURN_IF_NV_IS_NOT_AVAILABLE;
772
773         if((session->timeout < g_time)
774            || (session->epoch != g_timeEpoch))
775             return TPM_RC_EXPIRED;
776     }
777     // If command code is provided it must match
778     if(session->commandCode != 0)
779     {
780         if(session->commandCode != command->code)
781             return TPM_RC_POLICY_CC;
782     }
783     else
784     {
785         // If command requires a DUP or ADMIN authorization, the session must have
786         // command code set.
787         AUTH_ROLE   role = CommandAuthRole(command->index, sessionIndex);
788         if(role == AUTH_ADMIN || role == AUTH_DUP)
789             return TPM_RC_POLICY_FAIL;
790     }
791     // Check command locality.
792     {
793         BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
794         BYTE        *buffer = sessionLocality;
795
796         // Get existing locality setting in canonical form
797         sessionLocality[0] = 0;
```

```
798         TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
799
800         // See if the locality has been set
801         if(sessionLocality[0] != 0)
802         {
803             // If so, get the current locality
804             locality = _plat__LocalityGet();
805             if(locality < 5)
806             {
807                 if(((sessionLocality[0] & (1 << locality)) == 0)
808                    || sessionLocality[0] > 31)
809                     return TPM_RC_LOCALITY;
810             }
811             else if(locality > 31)
812             {
813                 if(sessionLocality[0] != locality)
814                     return TPM_RC_LOCALITY;
815             }
816             else
817             {
818                 // Could throw an assert here but a locality error is just
819                 // as good. It just means that, whatever the locality is, it isn't
820                 // the locality requested so...
821                 return TPM_RC_LOCALITY;
822             }
823         }
824     } // end of locality check
825     // Check physical presence.
826     if(session->attributes.isPPRequired == SET
827        && !_plat__PhysicalPresenceAsserted())
828         return TPM_RC_PP;
829     // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
830     // DUP role for this handle.
831     if(session->u1.cpHash.b.size != 0)
832     {
833         BOOL        OK;
834         if(session->attributes.isCpHashDefined)
835             // Compare cpHash.
836             OK = MemoryEqual2B(&session->u1.cpHash.b,
837                                &ComputeCpHash(command, session->authHashAlg)->b);
838         else if(session->attributes.isTemplateSet)
839             OK = CompareTemplateHash(command, session);
840         else
841             OK = CompareNameHash(command, session);
842         if(!OK)
843             return TPM_RCS_POLICY_FAIL;
844     }
845     if(session->attributes.checkNvWritten)
846     {
847         NV_REF          locator;
848         NV_INDEX        *nvIndex;
849 //
850         // If this is not an NV index, the policy makes no sense so fail it.
851         if(HandleGetType(s_associatedHandles[sessionIndex]) != TPM_HT_NV_INDEX)
852             return TPM_RC_POLICY_FAIL;
853         // Get the index data
854         nvIndex = NvGetIndexInfo(s_associatedHandles[sessionIndex], &locator);
855
856         // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
857         if((IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
858            != (session->attributes.nvWrittenState == SET))
859             return TPM_RC_POLICY_FAIL;
860     }
861     return TPM_RC_SUCCESS;
862 }
```


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

```
863  static TPM_RC
864  RetrieveSessionData(
865      COMMAND         *command        // IN: main parsing structure for command
866      )
867  {
868      int              i;
869      TPM_RC           result;
870      SESSION         *session;
871      TPMA_SESSION     sessionAttributes;
872      TPM_HT           sessionType;
873      INT32            sessionIndex;
874      TPM_RC           errorIndex;
875  //
876      s_decryptSessionIndex = UNDEFINED_INDEX;
877      s_encryptSessionIndex = UNDEFINED_INDEX;
878      s_auditSessionIndex = UNDEFINED_INDEX;
879
880      for(sessionIndex = 0; command->authSize > 0; sessionIndex++)
881      {
882          errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
883
884          // If maximum allowed number of sessions has been parsed, return a size
885          // error with a session number that is larger than the number of allowed
886          // sessions
887          if(sessionIndex == MAX_SESSION_NUM)
888              return TPM_RCS_SIZE + errorIndex;
889          // make sure that the associated handle for each session starts out
890          // unassigned
891          s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
892
893          // First parameter: Session handle.
894          result = TPMI_SH_AUTH_SESSION_Unmarshal(
895              &s_sessionHandles[sessionIndex],
896              &command->parameterBuffer,
897              &command->authSize, TRUE);
898          if(result != TPM_RC_SUCCESS)
899              return result + TPM_RC_S + g_rcIndex[sessionIndex];
900          // Second parameter: Nonce.
901          result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
902                                         &command->parameterBuffer,
903                                         &command->authSize);
904          if(result != TPM_RC_SUCCESS)
905              return result + TPM_RC_S + g_rcIndex[sessionIndex];
906          // Third parameter: sessionAttributes.
907          result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
908                                          &command->parameterBuffer,
909                                          &command->authSize);
910          if(result != TPM_RC_SUCCESS)
911              return result + TPM_RC_S + g_rcIndex[sessionIndex];
912          // Fourth parameter: authValue (PW or HMAC).
913          result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
914                                        &command->parameterBuffer,
915                                        &command->authSize);
```

```
916         if(result != TPM_RC_SUCCESS)
917             return result + errorIndex;
918
919         sessionAttributes = s_attributes[sessionIndex];
920         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
921         {
922             // A PWAP session needs additional processing.
923             //     Can't have any attributes set other than continueSession bit
924             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt)
925                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt)
926                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit)
927                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
928                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset))
929                 return TPM_RCS_ATTRIBUTES + errorIndex;
930             //     The nonce size must be zero.
931             if(s_nonceCaller[sessionIndex].t.size != 0)
932                 return TPM_RCS_NONCE + errorIndex;
933             continue;
934         }
935         // For not password sessions...
936         // Find out if the session is loaded.
937         if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
938             return TPM_RC_REFERENCE_S0 + sessionIndex;
939         sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
940         session = SessionGet(s_sessionHandles[sessionIndex]);
941
942         // Check if the session is an HMAC/policy session.
943         if((session->attributes.isPolicy == SET
944             && sessionType == TPM_HT_HMAC_SESSION)
945            || (session->attributes.isPolicy == CLEAR
946                && sessionType == TPM_HT_POLICY_SESSION))
947             return TPM_RCS_HANDLE + errorIndex;
948         // Check that this handle has not previously been used.
949         for(i = 0; i < sessionIndex; i++)
950         {
951             if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
952                 return TPM_RCS_HANDLE + errorIndex;
953         }
954         // If the session is used for parameter encryption or audit as well, set
955         // the corresponding Indexes.
956
957         // First process decrypt.
958         if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt))
959         {
960             // Check if the commandCode allows command parameter encryption.
961             if(DecryptSize(command->index) == 0)
962                 return TPM_RCS_ATTRIBUTES + errorIndex;
963             // Encrypt attribute can only appear in one session
964             if(s_decryptSessionIndex != UNDEFINED_INDEX)
965                 return TPM_RCS_ATTRIBUTES + errorIndex;
966             // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
967             if(session->symmetric.algorithm == TPM_ALG_NULL)
968                 return TPM_RCS_SYMMETRIC + errorIndex;
969             // All checks passed, so set the index for the session used to decrypt
970             // a command parameter.
971             s_decryptSessionIndex = sessionIndex;
972         }
973         // Now process encrypt.
974         if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt))
975         {
976             // Check if the commandCode allows response parameter encryption.
977             if(EncryptSize(command->index) == 0)
978                 return TPM_RCS_ATTRIBUTES + errorIndex;
979             // Encrypt attribute can only appear in one session.
980             if(s_encryptSessionIndex != UNDEFINED_INDEX)
981                 return TPM_RCS_ATTRIBUTES + errorIndex;
```



```
982                 // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
983                 if(session->symmetric.algorithm == TPM_ALG_NULL)
984                     return TPM_RCS_SYMMETRIC + errorIndex;
985                 // All checks passed, so set the index for the session used to encrypt
986                 // a response parameter.
987                 s_encryptSessionIndex = sessionIndex;
988             }
989             // At last process audit.
990             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit))
991             {
992                 // Audit attribute can only appear in one session.
993                 if(s_auditSessionIndex != UNDEFINED_INDEX)
994                     return TPM_RCS_ATTRIBUTES + errorIndex;
995                 // An audit session can not be policy session.
996                 if(HandleGetType(s_sessionHandles[sessionIndex])
997                    == TPM_HT_POLICY_SESSION)
998                     return TPM_RCS_ATTRIBUTES + errorIndex;
999                 // If this is a reset of the audit session, or the first use
1000                // of the session as an audit session, it doesn't matter what
1001                // the exclusive state is. The session will become exclusive.
1002                if(!IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset)
1003                   && session->attributes.isAudit == SET)
1004                {
1005                    // Not first use or reset. If auditExlusive is SET, then this
1006                    // session must be the current exclusive session.
1007                    if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
1008                       && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
1009                        return TPM_RC_EXCLUSIVE;
1010                }
1011                s_auditSessionIndex = sessionIndex;
1012            }
1013            // Initialize associated handle as undefined. This will be changed when
1014            // the handles are processed.
1015            s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
1016        }
1017        command->sessionNum = sessionIndex;
1018        return TPM_RC_SUCCESS;
1019    }
```

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

```
1020    static TPM_RC
1021    CheckLockedOut(
1022        BOOL            lockoutAuthCheck   // IN: TRUE if checking is for lockoutAuth
1023        )
1024    {
1025        // If NV is unavailable, and current cycle state recorded in NV is not
1026        // SU_NONE_VALUE, refuse to check any authorization because we would
1027        // not be able to handle a DA failure.
1028        if(!NV_IS_AVAILABLE && NV_IS_ORDERLY)
1029            return g_NvStatus;
```

```
1030        // Check if DA info needs to be updated in NV.
1031        if(s_DAPendingOnNV)
1032        {
1033            // If NV is accessible,
1034            RETURN_IF_NV_IS_NOT_AVAILABLE;
1035
1036            // ... write the pending DA data and proceed.
1037            NV_SYNC_PERSISTENT(lockOutAuthEnabled);
1038            NV_SYNC_PERSISTENT(failedTries);
1039            s_DAPendingOnNV = FALSE;
1040        }
1041        // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
1042        // is disabled...
1043        if(lockoutAuthCheck)
1044        {
1045            if(gp.lockOutAuthEnabled == FALSE)
1046                return TPM_RC_LOCKOUT;
1047        }
1048        else
1049        {
1050            // ... or if the number of failed tries has been maxed out.
1051            if(gp.failedTries >= gp.maxTries)
1052                return TPM_RC_LOCKOUT;
1053    #if USE_DA_USED
1054            // If the daUsed flag is not SET, then no DA validation until the
1055            // daUsed state is written to NV
1056            if(!g_daUsed)
1057            {
1058                RETURN_IF_NV_IS_NOT_AVAILABLE;
1059                g_daUsed = TRUE;
1060                gp.orderlyState = SU_DA_USED_VALUE;
1061                NV_SYNC_PERSISTENT(orderlyState);
1062                return TPM_RC_RETRY;
1063            }
1064    #endif
1065        }
1066        return TPM_RC_SUCCESS;
1067    }
```

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

```
1068    static TPM_RC
```



```
1069  CheckAuthSession(
1070      COMMAND         *command,       // IN: primary parsing structure
1071      UINT32           sessionIndex   // IN: index of session to be processed
1072      )
1073  {
1074      TPM_RC           result = TPM_RC_SUCCESS;
1075      SESSION         *session = NULL;
1076      TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
1077      TPM_HANDLE       associatedHandle = s_associatedHandles[sessionIndex];
1078      TPM_HT           sessionHandleType = HandleGetType(sessionHandle);
1079  //
1080      pAssert(sessionHandle != TPM_RH_UNASSIGNED);
1081
1082      // Take care of physical presence
1083      if(associatedHandle == TPM_RH_PLATFORM)
1084      {
1085          // If the physical presence is required for this command, check for PP
1086          // assertion. If it isn't asserted, no point going any further.
1087          if(PhysicalPresenceIsRequired(command->index)
1088             && !_plat__PhysicalPresenceAsserted())
1089              return TPM_RC_PP;
1090      }
1091      if(sessionHandle != TPM_RS_PW)
1092      {
1093          session = SessionGet(sessionHandle);
1094
1095          // Set includeAuth to indicate if DA checking will be required and if the
1096          // authValue will be included in any HMAC.
1097          if(sessionHandleType == TPM_HT_POLICY_SESSION)
1098          {
1099              // For a policy session, will check the DA status of the entity if either
1100              // isAuthValueNeeded or isPasswordNeeded is SET.
1101              session->attributes.includeAuth =
1102                  session->attributes.isAuthValueNeeded
1103                  || session->attributes.isPasswordNeeded;
1104          }
1105          else
1106          {
1107              // For an HMAC session, need to check unless the session
1108              // is bound.
1109              session->attributes.includeAuth =
1110                  !IsSessionBindEntity(s_associatedHandles[sessionIndex], session);
1111          }
1112      }
1113      // If the authorization session is going to use an authValue, then make sure
1114      // that access to that authValue isn't locked out.
1115      // Note: session == NULL for a PW session.
1116      if(session == NULL || session->attributes.includeAuth)
1117      {
1118          // See if entity is subject to lockout.
1119          if(!IsDAExempted(associatedHandle))
1120          {
1121              // See if in lockout
1122              result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1123              if(result != TPM_RC_SUCCESS)
1124                  return result;
1125          }
1126      }
1127      // Policy or HMAC+PW?
1128      if(sessionHandleType != TPM_HT_POLICY_SESSION)
1129      {
1130          // for non-policy session make sure that a policy session is not required
1131          if(IsPolicySessionRequired(command->index, sessionIndex))
1132              return TPM_RC_AUTH_TYPE;
1133          // The authValue must be available.
1134          // Note: The authValue is going to be "used" even if it is an EmptyAuth.
```

```
1135            // and the session is bound.
1136            if(!IsAuthValueAvailable(associatedHandle, command->index, sessionIndex))
1137                return TPM_RC_AUTH_UNAVAILABLE;
1138        }
1139        else
1140        {
1141            // ... see if the entity has a policy, ...
1142            // Note: IsAuthPolicyAvalable will return FALSE if the sensitive area of the
1143            // object is not loaded
1144            if(!IsAuthPolicyAvailable(associatedHandle, command->index, sessionIndex))
1145                return TPM_RC_AUTH_UNAVAILABLE;
1146            // ... and check the policy session.
1147            result = CheckPolicyAuthSession(command, sessionIndex);
1148            if(result != TPM_RC_SUCCESS)
1149                return result;
1150        }
1151        // Check authorization according to the type
1152        if(session == NULL || session->attributes.isPasswordNeeded == SET)
1153            result = CheckPWAuthSession(sessionIndex);
1154        else
1155            result = CheckSessionHMAC(command, sessionIndex);
1156        // Do processing for PIN Indexes are only three possibilities for 'result' at
1157        // this point: TPM_RC_SUCCESS, TPM_RC_AUTH_FAIL, and TPM_RC_BAD_AUTH.
1158        // For all these cases, we would have to process a PIN index if the
1159        // authValue of the index was used for authorization.
1160        // See if we need to do anything to a PIN index
1161        if(TPM_HT_NV_INDEX == HandleGetType(associatedHandle))
1162        {
1163            NV_REF           locator;
1164            NV_INDEX        *nvIndex = NvGetIndexInfo(associatedHandle, &locator);
1165            NV_PIN           pinData;
1166            TPMA_NV          nvAttributes;
1167    //
1168            pAssert(nvIndex != NULL);
1169            nvAttributes = nvIndex->publicArea.attributes;
1170            // If this is a PIN FAIL index and the value has been written
1171            // then we can update the counter (increment or clear)
1172            if(IsNvPinFailIndex(nvAttributes)
1173               && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
1174            {
1175                pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1176                if(result != TPM_RC_SUCCESS)
1177                    pinData.pin.pinCount++;
1178                else
1179                    pinData.pin.pinCount = 0;
1180                NvWriteUINT64Data(nvIndex, pinData.intVal);
1181            }
1182            // If this is a PIN PASS Index, increment if we have used the
1183            // authorization value for anything other than NV_Read.
1184            // NOTE: If the counter has already hit the limit, then we
1185            // would not get here because the authorization value would not
1186            // be available and the TPM would have returned before it gets here
1187            else if(IsNvPinPassIndex(nvAttributes)
1188                    && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN)
1189                    && result == TPM_RC_SUCCESS)
1190            {
1191                // If the access is valid, then increment the use counter
1192                pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1193                pinData.pin.pinCount++;
1194                NvWriteUINT64Data(nvIndex, pinData.intVal);
1195            }
1196        }
1197        return result;
1198    }
1199    #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

```
1200  static TPM_RC
1201  CheckCommandAudit(
1202      COMMAND         *command
1203      )
1204  {
1205      // If the audit digest is clear and command audit is required, NV must be
1206      // available so that TPM2_GetCommandAuditDigest() is able to increment
1207      // audit counter. If NV is not available, the function bails out to prevent
1208      // the TPM from attempting an operation that would fail anyway.
1209      if(gr.commandAuditDigest.t.size == 0
1210         || GetCommandCode(command->index) == TPM_CC_GetCommandAuditDigest)
1211      {
1212          RETURN_IF_NV_IS_NOT_AVAILABLE;
1213      }
1214      // Make sure that the cpHash is computed for the algorithm
1215      ComputeCpHash(command, gp.auditHashAlg);
1216      return TPM_RC_SUCCESS;
1217  }
1218  #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

```
1219  TPM_RC
1220  ParseSessionBuffer(
1221      COMMAND         *command        // IN: the structure that contains
1222      )
1223  {
1224      TPM_RC              result;
1225      UINT32              i;
1226      INT32               size = 0;
1227      TPM2B_AUTH          extraKey;
1228      UINT32              sessionIndex;
1229      TPM_RC              errorIndex;
1230      SESSION            *session = NULL;
1231  //
1232      // Check if a command allows any session in its session area.
1233      if(!IsSessionAllowed(command->index))
1234          return TPM_RC_AUTH_CONTEXT;
1235      // Default-initialization.
1236      command->sessionNum = 0;
1237
1238      result = RetrieveSessionData(command);
1239      if(result != TPM_RC_SUCCESS)
1240          return result;
```

```
1241        // There is no command in the TPM spec that has more handles than
1242        // MAX_SESSION_NUM.
1243        pAssert(command->handleNum <= MAX_SESSION_NUM);
1244
1245        // Associate the session with an authorization handle.
1246        for(i = 0; i < command->handleNum; i++)
1247        {
1248            if(CommandAuthRole(command->index, i) != AUTH_NONE)
1249            {
1250                // If the received session number is less than the number of handles
1251                // that requires authorization, an error should be returned.
1252                // Note: for all the TPM 2.0 commands, handles requiring
1253                // authorization come first in a command input and there are only ever
1254                // two values requiring authorization
1255                if(i > (command->sessionNum - 1))
1256                    return TPM_RC_AUTH_MISSING;
1257                // Record the handle associated with the authorization session
1258                s_associatedHandles[i] = command->handles[i];
1259            }
1260        }
1261        // Consistency checks are done first to avoid authorization failure when the
1262        // command will not be executed anyway.
1263        for(sessionIndex = 0; sessionIndex < command->sessionNum; sessionIndex++)
1264        {
1265            errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
1266            // PW session must be an authorization session
1267            if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1268            {
1269                if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1270                    return TPM_RCS_HANDLE + errorIndex;
1271                // a password session can't be audit, encrypt or decrypt
1272                if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1273                   || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1274                   || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1275                    return TPM_RCS_ATTRIBUTES + errorIndex;
1276                session = NULL;
1277            }
1278            else
1279            {
1280                session = SessionGet(s_sessionHandles[sessionIndex]);
1281
1282                // A trial session can not appear in session area, because it cannot
1283                // be used for authorization, audit or encrypt/decrypt.
1284                if(session->attributes.isTrialPolicy == SET)
1285                    return TPM_RCS_ATTRIBUTES + errorIndex;
1286
1287                // See if the session is bound to a DA protected entity
1288                // NOTE: Since a policy session is never bound, a policy is still
1289                // usable even if the object is DA protected and the TPM is in
1290                // lockout.
1291                if(session->attributes.isDaBound == SET)
1292                {
1293                    result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1294                    if(result != TPM_RC_SUCCESS)
1295                        return result;
1296                }
1297                // If this session is for auditing, make sure the cpHash is computed.
1298                if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit))
1299                    ComputeCpHash(command, session->authHashAlg);
1300            }
1301
1302            // if the session has an associated handle, check the authorization
1303            if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1304            {
1305                result = CheckAuthSession(command, sessionIndex);
1306                if(result != TPM_RC_SUCCESS)
```



```
1307                    return RcSafeAddToResult(result, errorIndex);
1308            }
1309            else
1310            {
1311                // a session that is not for authorization must either be encrypt,
1312                // decrypt, or audit
1313                if(!IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1314                   &&  !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1315                   &&  !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1316                    return TPM_RCS_ATTRIBUTES + errorIndex;
1317
1318                // no authValue included in any of the HMAC computations
1319                pAssert(session != NULL);
1320                session->attributes.includeAuth = CLEAR;
1321
1322                // check HMAC for encrypt/decrypt/audit only sessions
1323                result = CheckSessionHMAC(command, sessionIndex);
1324                if(result != TPM_RC_SUCCESS)
1325                    return RcSafeAddToResult(result, errorIndex);
1326            }
1327        }
1328    #ifdef  TPM_CC_GetCommandAuditDigest
1329        // Check if the command should be audited. Need to do this before any parameter
1330        // encryption so that the cpHash for the audit is correct
1331        if(CommandAuditIsRequired(command->index))
1332        {
1333            result = CheckCommandAudit(command);
1334            if(result != TPM_RC_SUCCESS)
1335                return result;              // No session number to reference
1336        }
1337    #endif
1338        // Decrypt the first parameter if applicable. This should be the last operation
1339        // in session processing.
1340        // If the encrypt session is associated with a handle and the handle's
1341        // authValue is available, then authValue is concatenated with sessionKey to
1342        // generate encryption key, no matter if the handle is the session bound entity
1343        // or not.
1344        if(s_decryptSessionIndex != UNDEFINED_INDEX)
1345        {
1346            // If this is an authorization session, include the authValue in the
1347            // generation of the decryption key
1348            if(s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED)
1349            {
1350                EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1351                                   &extraKey);
1352            }
1353            else
1354            {
1355                extraKey.b.size = 0;
1356            }
1357            size = DecryptSize(command->index);
1358            result = CryptParameterDecryption(s_sessionHandles[s_decryptSessionIndex],
1359                                              &s_nonceCaller[s_decryptSessionIndex].b,
1360                                              command->parameterSize, (UINT16)size,
1361                                              &extraKey,
1362                                              command->parameterBuffer);
1363            if(result != TPM_RC_SUCCESS)
1364                return RcSafeAddToResult(result,
1365                                         TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1366        }
1367
1368        return TPM_RC_SUCCESS;
1369    }
```


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

```
TPM_RC
CheckAuthNoSession(
    COMMAND         *command        // IN: command parsing structure
    )
{
    UINT32 i;
    TPM_RC           result = TPM_RC_SUCCESS;
//
    // Check if the command requires authorization
    for(i = 0; i < command->handleNum; i++)
    {
        if(CommandAuthRole(command->index, i) != AUTH_NONE)
            return TPM_RC_AUTH_MISSING;
    }
#ifdef  TPM_CC_GetCommandAuditDigest
    // Check if the command should be audited.
    if(CommandAuditIsRequired(command->index))
    {
        result = CheckCommandAudit(command);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
#endif
    // Initialize number of sessions to be 0
    command->sessionNum = 0;

    return TPM_RC_SUCCESS;
}
```

6.4.5 Response Session Processing

6.4.5.1 Introduction

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

6.4.5.2 ComputeRpHash()

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

```
static TPM2B_DIGEST *
ComputeRpHash(
    COMMAND         *command,       // IN: command structure
    TPM_ALG_ID       hashAlg        // IN: hash algorithm to compute rpHash
    )
{
    TPM2B_DIGEST    *rpHash = GetRpHashPointer(command, hashAlg);
    HASH_STATE       hashState;
//
    if(rpHash->t.size == 0)
    {
    //   rpHash := hash(responseCode || commandCode || parameters)
```



```
1410
1411        // Initiate hash creation.
1412            rpHash->t.size = CryptHashStart(&hashState, hashAlg);
1413
1414            // Add hash constituents.
1415            CryptDigestUpdateInt(&hashState, sizeof(TPM_RC), TPM_RC_SUCCESS);
1416            CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
1417            CryptDigestUpdate(&hashState, command->parameterSize,
1418                              command->parameterBuffer);
1419            // Complete hash computation.
1420            CryptHashEnd2B(&hashState, &rpHash->b);
1421        }
1422        return rpHash;
1423    }
```

6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```
1424    static void
1425    InitAuditSession(
1426        SESSION         *session        // session to be initialized
1427        )
1428    {
1429        // Mark session as an audit session.
1430        session->attributes.isAudit = SET;
1431
1432        // Audit session can not be bound.
1433        session->attributes.isBound = CLEAR;
1434
1435        // Size of the audit log is the size of session hash algorithm digest.
1436        session->u2.auditDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
1437
1438        // Set the original digest value to be 0.
1439        MemorySet(&session->u2.auditDigest.t.buffer,
1440                  0,
1441                  session->u2.auditDigest.t.size);
1442        return;
1443    }
```

6.4.5.4 UpdateAuditDigest

Function to update an audit digest

```
1444    static void
1445    UpdateAuditDigest(
1446        COMMAND         *command,
1447        TPMI_ALG_HASH    hashAlg,
1448        TPM2B_DIGEST    *digest
1449        )
1450    {
1451        HASH_STATE       hashState;
1452        TPM2B_DIGEST    *cpHash = GetCpHash(command, hashAlg);
1453        TPM2B_DIGEST    *rpHash = ComputeRpHash(command, hashAlg);
1454    //
1455        pAssert(cpHash != NULL);
1456
1457        // digestNew :=  hash (digestOld || cpHash || rpHash)
1458            // Start hash computation.
1459        digest->t.size = CryptHashStart(&hashState, hashAlg);
1460            // Add old digest.
1461        CryptDigestUpdate2B(&hashState, &digest->b);
1462            // Add cpHash
```



```
1463        CryptDigestUpdate2B(&hashState, &cpHash->b);
1464            // Add rpHash
1465        CryptDigestUpdate2B(&hashState, &rpHash->b);
1466            // Finalize the hash.
1467        CryptHashEnd2B(&hashState, &digest->b);
1468    }
```

6.4.5.5 Audit()

This function updates the audit digest in an audit session.

```
1469    static void
1470    Audit(
1471        COMMAND         *command,       // IN: primary control structure
1472        SESSION         *auditSession   // IN: loaded audit session
1473        )
1474    {
1475        UpdateAuditDigest(command, auditSession->authHashAlg,
1476                          &auditSession->u2.auditDigest);
1477        return;
1478    }
1479    #ifdef  TPM_CC_GetCommandAuditDigest
```

6.4.5.6 CommandAudit()

This function updates the command audit digest.

```
1480    static void
1481    CommandAudit(
1482        COMMAND         *command        // IN:
1483        )
1484    {
1485        // If the digest.size is one, it indicates the special case of changing
1486        // the audit hash algorithm. For this case, no audit is done on exit.
1487        // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1488        // force an update to the NV on exit so that the change in digest will
1489        // be recorded. So, it is safe to exit here without setting any flags
1490        // because the digest change will be written to NV when this code exits.
1491        if(gr.commandAuditDigest.t.size == 1)
1492        {
1493            gr.commandAuditDigest.t.size = 0;
1494            return;
1495        }
1496        // If the digest size is zero, need to start a new digest and increment
1497        // the audit counter.
1498        if(gr.commandAuditDigest.t.size == 0)
1499        {
1500            gr.commandAuditDigest.t.size = CryptHashGetDigestSize(gp.auditHashAlg);
1501            MemorySet(gr.commandAuditDigest.t.buffer,
1502                      0,
1503                      gr.commandAuditDigest.t.size);
1504
1505            // Bump the counter and save its value to NV.
1506            gp.auditCounter++;
1507            NV_SYNC_PERSISTENT(auditCounter);
1508        }
1509        UpdateAuditDigest(command, gp.auditHashAlg, &gr.commandAuditDigest);
1510        return;
1511    }
1512    #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.

```
1513 static void
1514 UpdateAuditSessionStatus(
1515     COMMAND         *command        // IN: primary control structure
1516     )
1517 {
1518     UINT32           i;
1519     TPM_HANDLE       auditSession = TPM_RH_UNASSIGNED;
1520 //
1521     // Iterate through sessions
1522     for(i = 0; i < command->sessionNum; i++)
1523     {
1524         SESSION     *session;
1525 //
1526         // PW session do not have a loaded session and can not be an audit
1527         // session either.  Skip it.
1528         if(s_sessionHandles[i] == TPM_RS_PW)
1529             continue;
1530         session = SessionGet(s_sessionHandles[i]);
1531
1532         // If a session is used for audit
1533         if(IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, audit))
1534         {
1535             // An audit session has been found
1536             auditSession = s_sessionHandles[i];
1537
1538             // If the session has not been an audit session yet, or
1539             // the auditSetting bits indicate a reset, initialize it and set
1540             // it to be the exclusive session
1541             if(session->attributes.isAudit == CLEAR
1542                || IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditReset))
1543             {
1544                 InitAuditSession(session);
1545                 g_exclusiveAuditSession = auditSession;
1546             }
1547             else
1548             {
1549                 // Check if the audit session is the current exclusive audit
1550                 // session and, if not, clear previous exclusive audit session.
1551                 if(g_exclusiveAuditSession != auditSession)
1552                     g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1553             }
1554             // Report audit session exclusivity.
1555             if(g_exclusiveAuditSession == auditSession)
1556             {
1557                 SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1558             }
1559             else
1560             {
1561                 CLEAR_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1562             }
1563             // Extend audit log.
1564             Audit(command, session);
1565         }
```

```
1566        }
1567        // If no audit session is found in the command, and the command allows
1568        // a session then, clear the current exclusive
1569        // audit session.
1570        if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(command->index))
1571        {
1572            g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1573        }
1574        return;
1575    }
```

6.4.5.8 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```
1576    static void
1577    ComputeResponseHMAC(
1578        COMMAND         *command,       // IN: command structure
1579        UINT32           sessionIndex,  // IN: session index to be processed
1580        SESSION         *session,       // IN: loaded session
1581        TPM2B_DIGEST    *hmac           // OUT: authHMAC
1582        )
1583    {
1584        TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1585        TPM2B_KEY        key;       // HMAC key
1586        BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
1587        BYTE            *buffer;
1588        UINT32           marshalSize;
1589        HMAC_STATE       hmacState;
1590        TPM2B_DIGEST    *rpHash = ComputeRpHash(command, session->authHashAlg);
1591    //
1592        // Generate HMAC key
1593        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1594
1595        // Add the object authValue if required
1596        if(session->attributes.includeAuth == SET)
1597        {
1598            // Note: includeAuth may be SET for a policy that is used in
1599            // UndefineSpaceSpecial(). At this point, the Index has been deleted
1600            // so the includeAuth will have no meaning. However, the
1601            // s_associatedHandles[] value for the session is now set to TPM_RH_NULL so
1602            // this will return the authValue associated with TPM_RH_NULL and that is
1603            // and empty buffer.
1604            TPM2B_AUTH          authValue;
1605    //
1606            // Get the authValue with trailing zeros removed
1607            EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
1608
1609            // Add it to the key
1610            MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
1611        }
1612
1613        // if the HMAC key size is 0, the response HMAC is computed according to the
1614        // input HMAC
1615        if(key.t.size == 0
1616           && s_inputAuthValues[sessionIndex].t.size == 0)
1617        {
1618            hmac->t.size = 0;
1619            return;
1620        }
1621        // Start HMAC computation.
1622        hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
1623
1624        // Add hash components.
```



```
1625        CryptDigestUpdate2B(&hmacState.hashState, &rpHash->b);
1626        CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
1627        CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
1628
1629        // Add session attributes.
1630        buffer = marshalBuffer;
1631        marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1632        CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
1633
1634        // Finalize HMAC.
1635        CryptHmacEnd2B(&hmacState, &hmac->b);
1636
1637        return;
1638    }
```

6.4.5.9 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time.

b) Clears a policy session since nonce is rolling.

```
1639    static void
1640    UpdateInternalSession(
1641        SESSION         *session,       // IN: the session structure
1642        UINT32           i              // IN: session number
1643        )
1644    {
1645        // If nonce is rolling in a policy session, the policy related data
1646        // will be re-initialized.
1647        if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION
1648           && IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1649        {
1650            // When the nonce rolls it starts a new timing interval for the
1651            // policy session.
1652            SessionResetPolicyData(session);
1653            SessionSetStartTime(session);
1654        }
1655        return;
1656    }
```

6.4.5.10 BuildSingleResponseAuth()

Function to compute response HMAC value for a policy or HMAC session.

```
1657    static TPM2B_NONCE *
1658    BuildSingleResponseAuth(
1659        COMMAND         *command,       // IN: command structure
1660        UINT32           sessionIndex,  // IN: session index to be processed
1661        TPM2B_AUTH      *auth           // OUT: authHMAC
1662        )
1663    {
1664        // Fill in policy/HMAC based session response.
1665        SESSION     *session = SessionGet(s_sessionHandles[sessionIndex]);
1666    //
1667        // If the session is a policy session with isPasswordNeeded SET, the
1668        // authorization field is empty.
1669        if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1670           && session->attributes.isPasswordNeeded == SET)
1671            auth->t.size = 0;
1672        else
1673            // Compute response HMAC.
```

```
1674            ComputeResponseHMAC(command, sessionIndex, session, auth);
1675
1676        UpdateInternalSession(session, sessionIndex);
1677        return &session->nonceTPM;
1678    }
```

6.4.5.11 UpdateAllNonceTPM()

Updates TPM nonce for all sessions in command.

```
1679    static void
1680    UpdateAllNonceTPM(
1681        COMMAND         *command        // IN: controlling structure
1682        )
1683    {
1684        UINT32      i;
1685        SESSION     *session;
1686    //
1687        for(i = 0; i < command->sessionNum; i++)
1688        {
1689            // If not a PW session, compute the new nonceTPM.
1690            if(s_sessionHandles[i] != TPM_RS_PW)
1691            {
1692                session = SessionGet(s_sessionHandles[i]);
1693                // Update nonceTPM in both internal session and response.
1694                CryptRandomGenerate(session->nonceTPM.t.size,
1695                                    session->nonceTPM.t.buffer);
1696            }
1697        }
1698        return;
1699    }
```

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.

```
1700    void
1701    BuildResponseSession(
1702        COMMAND         *command        // IN: structure that has relevant command
1703                                        //     information
1704        )
1705    {
1706        pAssert(command->authSize == 0);
1707
1708        // Reset the parameter buffer to point to the start of the parameters so that
1709        // there is a starting point for any rpHash that might be generated and so there
1710        // is a place where parameter encryption would start
1711        command->parameterBuffer = command->responseBuffer - command->parameterSize;
1712
1713        // Session nonces should be updated before parameter encryption
1714        if(command->tag == TPM_ST_SESSIONS)
1715        {
1716            UpdateAllNonceTPM(command);
1717
1718            // Encrypt first parameter if applicable. Parameter encryption should
1719            // happen after nonce update and before any rpHash is computed.
1720            // If the encrypt session is associated with a handle, the authValue of
1721            // this handle will be concatenated with sessionKey to generate
```



```
1722            // encryption key, no matter if the handle is the session bound entity
1723            // or not. The authValue is added to sessionKey only when the authValue
1724            // is available.
1725            if(s_encryptSessionIndex != UNDEFINED_INDEX)
1726            {
1727                UINT32          size;
1728                TPM2B_AUTH      extraKey;
1729    //
1730                extraKey.b.size = 0;
1731                // If this is an authorization session, include the authValue in the
1732                // generation of the encryption key
1733                if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED)
1734                {
1735                    EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1736                                       &extraKey);
1737                }
1738                size = EncryptSize(command->index);
1739                CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1740                                         &s_nonceCaller[s_encryptSessionIndex].b,
1741                                         (UINT16)size,
1742                                         &extraKey,
1743                                         command->parameterBuffer);
1744            }
1745        }
1746        // Audit sessions should be processed regardless of the tag because
1747        // a command with no session may cause a change of the exclusivity state.
1748        UpdateAuditSessionStatus(command);
1749    #if CC_GetCommandAuditDigest
1750        // Command Audit
1751        if(CommandAuditIsRequired(command->index))
1752            CommandAudit(command);
1753    #endif
1754        // Process command with sessions.
1755        if(command->tag == TPM_ST_SESSIONS)
1756        {
1757            UINT32          i;
1758    //
1759            pAssert(command->sessionNum > 0);
1760
1761            // Iterate over each session in the command session area, and create
1762            // corresponding sessions for response.
1763            for(i = 0; i < command->sessionNum; i++)
1764            {
1765                TPM2B_NONCE     *nonceTPM;
1766                TPM2B_DIGEST    responseAuth;
1767                // Make sure that continueSession is SET on any Password session.
1768                // This makes it marginally easier for the management software
1769                // to keep track of the closed sessions.
1770                if(s_sessionHandles[i] == TPM_RS_PW)
1771                {
1772                    SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession);
1773                    responseAuth.t.size = 0;
1774                    nonceTPM = (TPM2B_NONCE *)&responseAuth;
1775                }
1776                else
1777                {
1778                    // Compute the response HMAC and get a pointer to the nonce used.
1779                    // This function will also update the values if needed. Note, the
1780                    nonceTPM = BuildSingleResponseAuth(command, i, &responseAuth);
1781                }
1782                command->authSize += TPM2B_NONCE_Marshal(nonceTPM,
1783                                                         &command->responseBuffer,
1784                                                         NULL);
1785                command->authSize += TPMA_SESSION_Marshal(&s_attributes[i],
1786                                                          &command->responseBuffer,
1787                                                          NULL);
```

```
1788                command->authSize += TPM2B_DIGEST_Marshal(&responseAuth,
1789                                                          &command->responseBuffer,
1790                                                          NULL);
1791                if(!IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1792                    SessionFlush(s_sessionHandles[i]);
1793            }
1794        }
1795        return;
1796    }
```

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().

```
1797    void
1798    SessionRemoveAssociationToHandle(
1799        TPM_HANDLE       handle
1800        )
1801    {
1802        UINT32               i;
1803    //
1804        for(i = 0; i < MAX_SESSION_NUM; i++)
1805        {
1806            if(s_associatedHandles[i] == handle)
1807            {
1808                s_associatedHandles[i] = TPM_RH_NULL;
1809            }
1810        }
1811    }
```

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;

```

```
42      // Check the hierarchy of sign object.  For NULL sign handle, the hierarchy
43      // will be TPM_RH_NULL
44      if((signObject == NULL)
45         || (!signObject->attributes.epsHierarchy
46             && !signObject->attributes.ppsHierarchy))
47      {
48          // For signing key that is not in platform or endorsement hierarchy,
49          // obfuscate the reset, restart and firmware version information
50          UINT64          obfuscation[2];
51          CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gp.shProof.b, OBFUSCATE_STRING,
52                    &attest->qualifiedSigner.b, NULL, 128,
53                    (BYTE *)&obfuscation[0], NULL, FALSE);
54          // Obfuscate data
55          attest->firmwareVersion += obfuscation[0];
56          attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
57          attest->clockInfo.restartCount += (UINT32)obfuscation[1];
58      }
59      // External data
60      if(CryptIsSchemeAnonymous(scheme->scheme))
61          attest->extraData.t.size = 0;
62      else
63      {
64          // If we move the data to the attestation structure, then it is not
65          // used in the signing operation except as part of the signed data
66          attest->extraData = *data;
67          data->t.size = 0;
68      }
69  }
```

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)

```
70  TPM_RC
71  SignAttestInfo(
72      OBJECT              *signKey,           // IN: sign object
73      TPMT_SIG_SCHEME     *scheme,            // IN: sign scheme
74      TPMS_ATTEST         *certifyInfo,       // IN: the data to be signed
75      TPM2B_DATA          *qualifyingData,    // IN: extra data for the signing
76                                              //     process
77      TPM2B_ATTEST        *attest,            // OUT: marshaled attest blob to be
78                                              //     signed
79      TPMT_SIGNATURE      *signature          // OUT: signature
80      )
81  {
82      BYTE                    *buffer;
83      HASH_STATE              hashState;
84      TPM2B_DIGEST            digest;
85      TPM_RC                  result;
86
87      // Marshal TPMS_ATTEST structure for hash
88      buffer = attest->t.attestationData;
89      attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);
90
91      if(signKey == NULL)
```



```
 92         {
 93             signature->sigAlg = TPM_ALG_NULL;
 94             result = TPM_RC_SUCCESS;
 95         }
 96         else
 97         {
 98             TPMI_ALG_HASH           hashAlg;
 99             // Compute hash
100             hashAlg = scheme->details.any.hashAlg;
101             // need to set the receive buffer to get something put in it
102             digest.t.size = sizeof(digest.t.buffer);
103             digest.t.size = CryptHashBlock(hashAlg, attest->t.size,
104                                            attest->t.attestationData,
105                                            digest.t.size, digest.t.buffer);
106             // If there is qualifying data, need to rehash the data
107             // hash(qualifyingData || hash(attestationData))
108             if(qualifyingData->t.size != 0)
109             {
110                 CryptHashStart(&hashState, hashAlg);
111                 CryptDigestUpdate2B(&hashState, &qualifyingData->b);
112                 CryptDigestUpdate2B(&hashState, &digest.b);
113                 CryptHashEnd2B(&hashState, &digest.b);
114             }
115             // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
116             // TPM_RC_ATTRIBUTES error may be returned at this point
117             result = CryptSign(signKey, scheme, &digest, signature);
118
119             // Since the clock is used in an attestation, the state in NV is no longer
120             // "orderly" with respect to the data in RAM if the signature is valid
121             if(result == TPM_RC_SUCCESS)
122             {
123                 // Command uses the clock so need to clear the orderly state if it is
124                 // set.
125                 result = NvClearOrderly();
126             }
127         }
128         return result;
129     }
```

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(1)	object may sign
FALSE(0)	object may not sign

```
130 BOOL
131 IsSigningObject(
132     OBJECT          *object         // IN:
133     )
134 {
135     return ((object == NULL)
136             || ((IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, sign)
137                && object->publicArea.type != TPM_ALG_SYMCIPHER)));
138 }
```

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_PROOF     *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);

```



```
50      // Copy the rest as the IV
51      pAssert(iv->t.size <= sizeof(iv->t.buffer));
52      MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
53
54      return;
55  }
```

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

```
56  void
57  ComputeContextIntegrity(
58      TPMS_CONTEXT    *contextBlob,   // IN: context blob
59      TPM2B_DIGEST    *integrity      // OUT: integrity
60      )
61  {
62      HMAC_STATE          hmacState;
63      TPM2B_PROOF        *proof;
64      UINT16              integritySize;
65
66      // Get proof value
67      proof = HierarchyGetProof(contextBlob->hierarchy);
68
69      // Start HMAC
70      integrity->t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
71                                           &proof->b);
72
73      // Compute integrity size at the beginning of context blob
74      integritySize = sizeof(integrity->t.size) + integrity->t.size;
75
76      // Adding total reset counter so that the context cannot be
77      // used after a TPM Reset
78      CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
79                           gp.totalResetCount);
80
81      // If this is a ST_CLEAR object, add the clear count
82      // so that this contest cannot be loaded after a TPM Restart
83      if(contextBlob->savedHandle == 0x80000002)
84          CryptDigestUpdateInt(&hmacState.hashState, sizeof(gr.clearCount),
85                               gr.clearCount);
86
87      // Adding sequence number to the HMAC to make sure that it doesn't
88      // get changed
89      CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->sequence),
90                           contextBlob->sequence);
91
92      // Protect the handle
93      CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->savedHandle),
94                           contextBlob->savedHandle);
95
96      // Adding sensitive contextData, skip the leading integrity area
97      CryptDigestUpdate(&hmacState.hashState,
98                        contextBlob->contextBlob.t.size - integritySize,
99                        contextBlob->contextBlob.t.buffer + integritySize);
100
101     // Complete HMAC
102     CryptHmacEnd2B(&hmacState, &integrity->b);
103
104     return;
105 }
```


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.

```
106  void
107  SequenceDataExport(
108      HASH_OBJECT          *object,        // IN: an internal hash object
109      HASH_OBJECT_BUFFER   *exportObject   // OUT: a sequence context in a buffer
110      )
111  {
112      // If the hash object is not an event, then only one hash context is needed
113      int                    count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
114
115      for(count--; count >= 0; count--)
116      {
117          HASH_STATE           *hash = &object->state.hashState[count];
118          size_t                offset = (BYTE *)hash - (BYTE *)object;
119          BYTE                 *exportHash = &((BYTE *)exportObject)[offset];
120
121          CryptHashExportState(hash, (EXPORT_HASH_STATE *)exportHash);
122      }
123  }
```

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.

```
124  void
125  SequenceDataImport(
126      HASH_OBJECT          *object,        // IN/OUT: an internal hash object
127      HASH_OBJECT_BUFFER   *exportObject   // IN/OUT: a sequence context in a buffer
128      )
129  {
130      // If the hash object is not an event, then only one hash context is needed
131      int                    count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
132
133      for(count--; count >= 0; count--)
134      {
135          HASH_STATE           *hash = &object->state.hashState[count];
136          size_t                offset = (BYTE *)hash - (BYTE *)object;
137          BYTE                 *importHash = &((BYTE *)exportObject)[offset];
138  //
139          CryptHashImportState(hash, (EXPORT_HASH_STATE *)importHash);
140      }
141  }
```


7.4 Policy Command Support (Policy_spt.c)

7.4.1 Includes

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

7.4.2 Functions

7.4.2.1 PolicyParameterChecks()

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

```
6  TPM_RC
7  PolicyParameterChecks(
8      SESSION         *session,
9      UINT64           authTimeout,
10     TPM2B_DIGEST    *cpHashA,
11     TPM2B_NONCE     *nonce,
12     TPM_RC           blameNonce,
13     TPM_RC           blameCpHash,
14     TPM_RC           blameExpiration
15     )
16 {
17     // Validate that input nonceTPM is correct if present
18     if(nonce != NULL && nonce->t.size != 0)
19     {
20         if(!MemoryEqual2B(&nonce->b, &session->nonceTPM.b))
21             return TPM_RCS_NONCE + blameNonce;
22     }
23     // If authTimeout is set (expiration != 0...
24     if(authTimeout != 0)
25     {
26         // Validate input expiration.
27         // Cannot compare time if clock stop advancing.  A TPM_RC_NV_UNAVAILABLE
28         // or TPM_RC_NV_RATE error may be returned here.
29         RETURN_IF_NV_IS_NOT_AVAILABLE;
30
31         // if the time has already passed or the time epoch has changed then the
32         // time value is no longer good.
33         if((authTimeout < g_time)
34            || (session->epoch != g_timeEpoch))
35             return TPM_RCS_EXPIRED + blameExpiration;
36     }
37     // If the cpHash is present, then check it
38     if(cpHashA != NULL && cpHashA->t.size != 0)
39     {
40         // The cpHash input has to have the correct size
41         if(cpHashA->t.size != session->u2.policyDigest.t.size)
42             return TPM_RCS_SIZE + blameCpHash;
43
44         // If the cpHash has already been set, then this input value
45         // must match the current value.
46         if(session->u1.cpHash.b.size != 0
47            && !MemoryEqual2B(&cpHashA->b, &session->u1.cpHash.b))
48             return TPM_RC_CPHASH;
49     }
```



```
50      return TPM_RC_SUCCESS;
51  }
```

7.4.2.2 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.

```
52  void
53  PolicyContextUpdate(
54      TPM_CC           commandCode,   // IN: command code
55      TPM2B_NAME      *name,          // IN: name of entity
56      TPM2B_NONCE     *ref,           // IN: the reference data
57      TPM2B_DIGEST    *cpHash,        // IN: the cpHash (optional)
58      UINT64           policyTimeout, // IN: the timeout value for the policy
59      SESSION         *session        // IN/OUT: policy session to be updated
60      )
61  {
62      HASH_STATE           hashState;
63
64      // Start hash
65     CryptHashStart(&hashState, session->authHashAlg);
66
67      // policyDigest size should always be the digest size of session hash algorithm.
68      pAssert(session->u2.policyDigest.t.size
69              == CryptHashGetDigestSize(session->authHashAlg));
70
71      // add old digest
72      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
73
74      // add commandCode
75      CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
76
77      // add name if applicable
78      if(name != NULL)
79          CryptDigestUpdate2B(&hashState, &name->b);
80
81      // Complete the digest and get the results
82      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
83
84      // If the policy reference is not null, do a second update to the digest.
85      if(ref != NULL)
86      {
87
88          // Start second hash computation
89          CryptHashStart(&hashState, session->authHashAlg);
90
91          // add policyDigest
92          CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
93
94          // add policyRef
95          CryptDigestUpdate2B(&hashState, &ref->b);
96
97          // Complete second digest
98          CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
99      }
100     // Deal with the cpHash. If the cpHash value is present
101     // then it would have already been checked to make sure that
102     // it is compatible with the current value so all we need
103     // to do here is copy it and set the isCpHashDefined attribute
104     if(cpHash != NULL && cpHash->t.size != 0)
105     {
106         session->u1.cpHash = *cpHash;
107         session->attributes.isCpHashDefined = SET;
```



```
108         }
109
110         // update the timeout if it is specified
111         if(policyTimeout != 0)
112         {
113             // If the timeout has not been set, then set it to the new value
114             // than the current timeout then set it to the new value
115             if(session->timeout == 0 || session->timeout > policyTimeout)
116                 session->timeout = policyTimeout;
117         }
118         return;
119     }
```

7.4.2.3 ComputeAuthTimeout()

This function is used to determine what the authorization timeout value for the session should be.

```
120     UINT64
121     ComputeAuthTimeout(
122         SESSION         *session,               // IN: the session containing the time
123                                                 //     values
124         INT32            expiration,            // IN: either the number of seconds from
125                                                 //     the start of the session or the
126                                                 //     time in g_timer;
127         TPM2B_NONCE     *nonce                  // IN: indicator of the time base
128         )
129     {
130         UINT64           policyTime;
131         // If no expiration, policy time is 0
132         if(expiration == 0)
133             policyTime = 0;
134         else
135         {
136             if(expiration < 0)
137                 expiration = -expiration;
138             if(nonce->t.size == 0)
139                 // The input time is absolute Time (not Clock), but it is expressed
140                 // in seconds. To make sure that we don't time out too early, take the
141                 // current value of milliseconds in g_time and add that to the input
142                 // seconds value.
143                 policyTime = (((UINT64)expiration) * 1000) + g_time % 1000;
144             else
145                 // The policy timeout is the absolute value of the expiration in seconds
146                 // added to the start time of the policy.
147                 policyTime = session->startTime + (((UINT64)expiration) * 1000);
148
149         }
150         return policyTime;
151     }
```

7.4.2.4 PolicyDigestClear()

Function to reset the *policyDigest* of a session

```
152     void
153     PolicyDigestClear(
154         SESSION         *session
155         )
156     {
157         session->u2.policyDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
158         MemorySet(session->u2.policyDigest.t.buffer, 0,
159                   session->u2.policyDigest.t.size);
160     }
```

```
161 BOOL
162 PolicySptCheckCondition(
163     TPM_EO          operation,
164     BYTE            *opA,
165     BYTE            *opB,
166     UINT16           size
167     )
168 {
169     // Arithmetic Comparison
170     switch(operation)
171     {
172         case TPM_EO_EQ:
173             // compare A = B
174             return (UnsignedCompareB(size, opA, size, opB) == 0);
175             break;
176         case TPM_EO_NEQ:
177             // compare A != B
178             return (UnsignedCompareB(size, opA, size, opB) != 0);
179             break;
180         case TPM_EO_SIGNED_GT:
181             // compare A > B signed
182             return (SignedCompareB(size, opA, size, opB) > 0);
183             break;
184         case TPM_EO_UNSIGNED_GT:
185             // compare A > B unsigned
186             return (UnsignedCompareB(size, opA, size, opB) > 0);
187             break;
188         case TPM_EO_SIGNED_LT:
189             // compare A < B signed
190             return (SignedCompareB(size, opA, size, opB) < 0);
191             break;
192         case TPM_EO_UNSIGNED_LT:
193             // compare A < B unsigned
194             return (UnsignedCompareB(size, opA, size, opB) < 0);
195             break;
196         case TPM_EO_SIGNED_GE:
197             // compare A >= B signed
198             return (SignedCompareB(size, opA, size, opB) >= 0);
199             break;
200         case TPM_EO_UNSIGNED_GE:
201             // compare A >= B unsigned
202             return (UnsignedCompareB(size, opA, size, opB) >= 0);
203             break;
204         case TPM_EO_SIGNED_LE:
205             // compare A <= B signed
206             return (SignedCompareB(size, opA, size, opB) <= 0);
207             break;
208         case TPM_EO_UNSIGNED_LE:
209             // compare A <= B unsigned
210             return (UnsignedCompareB(size, opA, size, opB) <= 0);
211             break;
212         case TPM_EO_BITSET:
213             // All bits SET in B are SET in A. ((A&B)=B)
214         {
215             UINT32 i;
216             for(i = 0; i < size; i++)
217                 if((opA[i] & opB[i]) != opB[i])
218                     return FALSE;
219         }
220         break;
221         case TPM_EO_BITCLEAR:
222             // All bits SET in B are CLEAR in A. ((A&B)=0)
223         {
224             UINT32 i;
225             for(i = 0; i < size; i++)
226                 if((opA[i] & opB[i]) != 0)
```



```
227                         return FALSE;
228             }
229             break;
230             default:
231                 FAIL(FATAL_ERROR_INTERNAL);
232                 break;
233         }
234         return TRUE;
235     }
```

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

```
TPM_RC
NvReadAccessChecks(
    TPM_HANDLE        authHandle,     // IN: the handle that provided the
                                      //     authorization
    TPM_HANDLE        nvHandle,       // IN: the handle of the NV index to be read
    TPMA_NV           attributes      // IN: the attributes of 'nvHandle'
    )
{
    // If data is read locked, returns an error
   if(IS_ATTRIBUTE(attributes, TPMA_NV, READLOCKED))
       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(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERREAD))
            return TPM_RC_NV_AUTHORIZATION;
    }
    else if(authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided authorization then PPWRITE must be SET
        if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPREAD))
            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(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN))
        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

```
43  TPM_RC
44  NvWriteAccessChecks(
45      TPM_HANDLE        authHandle,     // IN: the handle that provided the
46                                        //     authorization
47      TPM_HANDLE        nvHandle,       // IN: the handle of the NV index to be written
48      TPMA_NV           attributes      // IN: the attributes of 'nvHandle'
49      )
50  {
51      // If data is write locked, returns an error
52      if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED))
53          return TPM_RC_NV_LOCKED;
54          // If the authorization was provided by the owner or platform, then check
55      // that the attributes allow the write.  If the authorization handle
56      // is the same as the index, then the checks were made when the authorization
57      // was checked..
58      if(authHandle == TPM_RH_OWNER)
59      {
60          // If Owner provided authorization then ONWERWRITE must be SET
61          if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERWRITE))
62              return TPM_RC_NV_AUTHORIZATION;
63      }
64      else if(authHandle == TPM_RH_PLATFORM)
65      {
66          // If Platform provided authorization then PPWRITE must be SET
67          if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPWRITE))
68              return TPM_RC_NV_AUTHORIZATION;
69      }
70      // If neither Owner nor Platform provided authorization, make sure that it was
71      // provided by this index.
72      else if(authHandle != nvHandle)
73          return TPM_RC_NV_AUTHORIZATION;
74      return TPM_RC_SUCCESS;
75  }
```

7.5.2.3 NvClearOrderly()

This function is used to cause gp.*orderlyState* to be cleared to the non-orderly state.

```
76  TPM_RC
77  NvClearOrderly(
78      void
79      )
80  {
81      if(gp.orderlyState < SU_DA_USED_VALUE)
82          RETURN_IF_NV_IS_NOT_AVAILABLE;
83      g_clearOrderly = TRUE;
84      return TPM_RC_SUCCESS;
85  }
```


7.5.2.4 NvIsPinPassIndex()

Function to check to see if an NV index is a PIN Pass Index

Return Value	Meaning
TRUE(1)	is pin pass
FALSE(0)	is not pin pass

```
BOOL
NvIsPinPassIndex(
    TPM_HANDLE          index       // IN: Handle to check
    )
{
    if(HandleGetType(index) == TPM_HT_NV_INDEX)
    {
        NV_INDEX                *nvIndex = NvGetIndexInfo(index, NULL);

        return IsNvPinPassIndex(nvIndex->publicArea.attributes);
    }
    return FALSE;
}
```

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

```
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
    )
```

```
40  {
41      const TPM2B         *seed = seedIn;
42
43      // Determine the algorithms for the KDF and the encryption/decryption
44      // For TPM_RH_NULL, using context settings
45      if(protector == NULL)
46      {
47          // Use the context encryption algorithm and key size
48          *symAlg = CONTEXT_ENCRYPT_ALG;
49          symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
50          *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
51      }
52      else
53      {
54          TPMT_SYM_DEF_OBJECT *symDef;
55          symDef = &protector->publicArea.parameters.asymDetail.symmetric;
56          *symAlg = symDef->algorithm;
57          *keyBits = symDef->keyBits.sym;
58          symKey->t.size = (*keyBits + 7) / 8;
59      }
60      // Get seed for KDF
61      if(seed == NULL)
62          seed = GetSeedForKDF(protector);
63      // KDFa to generate symmetric key and IV value
64      CryptKDFa(hashAlg, seed, STORAGE_KEY, name, NULL,
65                symKey->t.size * 8, symKey->t.buffer, NULL, FALSE);
66      return;
67  }
```

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.

```
68  static void
69  ComputeOuterIntegrity(
70      TPM2B           *name,              // IN: the name of the object
71      OBJECT          *protector,         // IN: the object that
72                                          //     provides protection. For an object,
73                                          //     it is a parent. For a credential, it
74                                          //     is the encrypt object. For
75                                          //     a Temporary Object, it is NULL
76      TPMI_ALG_HASH    hashAlg,           // IN: algorithm to use for integrity
77      TPM2B           *seedIn,            // IN: an external seed may be provided for
78                                          //     duplication blob. For non duplication
79                                          //     blob, this parameter should be NULL
80      UINT32           sensitiveSize,     // IN: size of the marshaled sensitive data
81      BYTE            *sensitiveData,     // IN: sensitive area
82      TPM2B_DIGEST    *integrity          // OUT: integrity
83      )
84  {
85      HMAC_STATE       hmacState;
86      TPM2B_DIGEST     hmacKey;
87      const TPM2B     *seed = seedIn;
88  //
89      // Get seed for KDF
90      if(seed == NULL)
91          seed = GetSeedForKDF(protector);
92      // Determine the HMAC key bits
93      hmacKey.t.size = CryptHashGetDigestSize(hashAlg);
94
95      // KDFa to generate HMAC key
```



```
96      CryptKDFa(hashAlg, seed, INTEGRITY_KEY, NULL, NULL,
97                hmacKey.t.size * 8, hmacKey.t.buffer, NULL, FALSE);
98      // Start HMAC and get the size of the digest which will become the integrity
99      integrity->t.size = CryptHmacStart2B(&hmacState, hashAlg, &hmacKey.b);
100
101     // Adding the marshaled sensitive area to the integrity value
102     CryptDigestUpdate(&hmacState.hashState, sensitiveSize, sensitiveData);
103
104     // Adding name
105     CryptDigestUpdate2B(&hmacState.hashState, name);
106
107     // Compute HMAC
108     CryptHmacEnd2B(&hmacState, &integrity->b);
109
110     return;
111 }
```

7.6.2.4 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap

```
112 static void
113 ComputeInnerIntegrity(
114     TPM_ALG_ID       hashAlg,        // IN: hash algorithm for inner wrap
115     TPM2B           *name,           // IN: the name of the object
116     UINT16           dataSize,       // IN: the size of sensitive data
117     BYTE            *sensitiveData,  // IN: sensitive data
118     TPM2B_DIGEST    *integrity       // OUT: inner integrity
119     )
120 {
121     HASH_STATE      hashState;
122 //
123     // Start hash and get the size of the digest which will become the integrity
124     integrity->t.size = CryptHashStart(&hashState, hashAlg);
125
126     // Adding the marshaled sensitive area to the integrity value
127     CryptDigestUpdate(&hashState, dataSize, sensitiveData);
128
129     // Adding name
130     CryptDigestUpdate2B(&hashState, name);
131
132     // Compute hash
133     CryptHashEnd2B(&hashState, &integrity->b);
134
135     return;
136 }
```

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

```
137 static UINT16
138 ProduceInnerIntegrity(
139     TPM2B           *name,           // IN: the name of the object
140     TPM_ALG_ID       hashAlg,        // IN: hash algorithm for inner wrap
141     UINT16           dataSize,       // IN: the size of sensitive data, excluding the
142                                      //     leading integrity buffer size
143     BYTE            *innerBuffer     // IN/OUT: inner buffer with sensitive data in
144                                      //     it.  At input, the leading bytes of this
```

```
145                                         //     buffer is reserved for integrity
146         )
147     {
148         BYTE            *sensitiveData; // pointer to the sensitive data
149         TPM2B_DIGEST     integrity;
150         UINT16           integritySize;
151         BYTE            *buffer;        // Auxiliary buffer pointer
152     //
153         // sensitiveData points to the beginning of sensitive data in innerBuffer
154         integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
155         sensitiveData = innerBuffer + integritySize;
156
157         ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);
158
159         // Add integrity at the beginning of inner buffer
160         buffer = innerBuffer;
161         TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
162
163         return dataSize + integritySize;
164     }
```

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

```
165     static TPM_RC
166     CheckInnerIntegrity(
167         TPM2B           *name,          // IN: the name of the object
168         TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
169         UINT16           dataSize,      // IN: the size of sensitive data, including the
170                                         //     leading integrity buffer size
171         BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
172                                         //     it
173         )
174     {
175         TPM_RC          result;
176         TPM2B_DIGEST    integrity;
177         TPM2B_DIGEST    integrityToCompare;
178         BYTE            *buffer;                // Auxiliary buffer pointer
179         INT32           size;
180     //
181         // Unmarshal integrity
182         buffer = innerBuffer;
183         size = (INT32)dataSize;
184         result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
185         if(result == TPM_RC_SUCCESS)
186         {
187             // Compute integrity to compare
188             ComputeInnerIntegrity(hashAlg, name, (UINT16)size, buffer,
189                                   &integrityToCompare);
190             // Compare outer blob integrity
191             if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
192                 result = TPM_RC_INTEGRITY;
193         }
194         return result;
195     }
```


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.

```
196 BOOL
197 AdjustAuthSize(
198     TPM2B_AUTH          *auth,          // IN/OUT: value to adjust
199     TPMI_ALG_HASH        nameAlg        // IN:
200     )
201 {
202     UINT16               digestSize;
203 //
204     // If there is no nameAlg, then this is a LoadExternal and the authVale can
205     // be any size up to the maximum allowed by the
206     digestSize = (nameAlg == TPM_ALG_NULL) ? sizeof(TPMU_HA)
207         : CryptHashGetDigestSize(nameAlg);
208     if(digestSize < MemoryRemoveTrailingZeros(auth))
209         return FALSE;
210     else if(digestSize > auth->t.size)
211         MemoryPad2B(&auth->b, digestSize);
212     auth->t.size = digestSize;
213
214     return TRUE;
215 }
```

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(1)	properties are those of a parent
FALSE(0)	properties are not those of a parent

```
216 BOOL
217 ObjectIsParent(
218     OBJECT          *parentObject   // IN: parent handle
219     )
220 {
221     return parentObject->attributes.isParent;
222 }
```

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()

```
223 TPM_RC
224 CreateChecks(
```



```
225     OBJECT              *parentObject,
226     TPMT_PUBLIC         *publicArea,
227     UINT16               sensitiveDataSize
228     )
229 {
230     TPMA_OBJECT          attributes = publicArea->objectAttributes;
231     TPM_RC               result = TPM_RC_SUCCESS;
232 //
233     // If the caller indicates that they have provided the data, then make sure that
234     // they have provided some data.
235     if((!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
236        && (sensitiveDataSize == 0))
237         return TPM_RCS_ATTRIBUTES;
238     // For an ordinary object, data can only be provided when sensitiveDataOrigin
239     // is CLEAR
240     if((parentObject != NULL)
241        && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
242        && (sensitiveDataSize != 0))
243         return TPM_RCS_ATTRIBUTES;
244     switch(publicArea->type)
245     {
246         case ALG_KEYEDHASH_VALUE:
247             // if this is a data object (sign == decrypt == CLEAR) then the
248             // TPM cannot be the data source.
249             if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
250                && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
251                && IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
252                 result = TPM_RC_ATTRIBUTES;
253             // comment out the next line in order to prevent a fixedTPM derivation
254             // parent
255 //            break;
256         case ALG_SYMCIPHER_VALUE:
257             // A restricted key symmetric key (SYMCIPHER and KEYEDHASH)
258             // must have sensitiveDataOrigin SET unless it has fixedParent and
259             // fixedTPM CLEAR.
260             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
261                 if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
262                     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
263                        || IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
264                         result = TPM_RCS_ATTRIBUTES;
265             break;
266         default: // Asymmetric keys cannot have the sensitive portion provided
267             if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
268                 result = TPM_RCS_ATTRIBUTES;
269             break;
270     }
271     if(TPM_RC_SUCCESS == result)
272     {
273         result = PublicAttributesValidation(parentObject, publicArea);
274     }
275     return result;
276 }
```

7.6.3.4 SchemeChecks

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

Error Returns	Meaning
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 ALG_NULL

```
277 TPM_RC
278 SchemeChecks(
279     OBJECT          *parentObject,  // IN: parent (null if primary seed)
280     TPMT_PUBLIC     *publicArea     // IN: public area of the object
281     )
282 {
283     TPMT_SYM_DEF_OBJECT     *symAlgs = NULL;
284     TPM_ALG_ID               scheme = TPM_ALG_NULL;
285     TPMA_OBJECT              attributes = publicArea->objectAttributes;
286     TPMU_PUBLIC_PARMS        *parms = &publicArea->parameters;
287 //
288     switch(publicArea->type)
289     {
290         case ALG_SYMCIPHER_VALUE:
291             symAlgs = &parms->symDetail.sym;
292             // If this is a decrypt key, then only the block cipher modes (not
293             // SMAC) are valid. TPM_ALG_NULL is OK too. If this is a 'sign' key,
294             // then any mode that got through the unmarshaling is OK.
295             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
296                && !CryptSymModeIsValid(symAlgs->mode.sym, TRUE))
297                 return TPM_RCS_SCHEME;
298             break;
299         case ALG_KEYEDHASH_VALUE:
300             scheme = parms->keyedHashDetail.scheme.scheme;
301             // if both sign and decrypt
302             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
303                == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
304             {
305                 // if both sign and decrypt are set or clear, then need
306                 // ALG_NULL as scheme
307                 if(scheme != TPM_ALG_NULL)
308                     return TPM_RCS_SCHEME;
309             }
310             else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
311                     && scheme != TPM_ALG_HMAC)
312                 return TPM_RCS_SCHEME;
313             else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
314             {
315                 if(scheme != TPM_ALG_XOR)
316                     return TPM_RCS_SCHEME;
317                 // If this is a derivation parent, then the KDF needs to be
318                 // SP800-108 for this implementation. This is the only derivation
319                 // supported by this implementation. Other implementations could
320                 // support additional schemes. There is no default.
321                 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
322                 {
323                     if(parms->keyedHashDetail.scheme.details.xor.kdf
324                        != TPM_ALG_KDF1_SP800_108)
325                         return TPM_RCS_SCHEME;
326                     // Must select a digest.
```

```
327                     if(CryptHashGetDigestSize(
328                         parms->keyedHashDetail.scheme.details.xor.hashAlg) == 0)
329                         return TPM_RCS_HASH;
330                 }
331             }
332             break;
333         default: // handling for asymmetric
334             scheme = parms->asymDetail.scheme.scheme;
335             symAlgs = &parms->asymDetail.symmetric;
336             // if the key is both sign and decrypt, then the scheme must be
337             // ALG_NULL because there is no way to specify both a sign and a
338             // decrypt scheme in the key.
339             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
340                == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
341             {
342                 // scheme must be TPM_ALG_NULL
343                 if(scheme != TPM_ALG_NULL)
344                     return TPM_RCS_SCHEME;
345             }
346             else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
347             {
348                 // If this is a signing key, see if it has a signing scheme
349                 if(CryptIsAsymSignScheme(publicArea->type, scheme))
350                 {
351                     // if proper signing scheme then it needs a proper hash
352                     if(parms->asymDetail.scheme.details.anySig.hashAlg
353                        == TPM_ALG_NULL)
354                         return TPM_RCS_SCHEME;
355                 }
356                 else
357                 {
358                     // signing key that does not have a proper signing scheme.
359                     // This is OK if the key is not restricted and its scheme
360                     // is TPM_ALG_NULL
361                     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
362                        || scheme != TPM_ALG_NULL)
363                         return TPM_RCS_SCHEME;
364                 }
365             }
366             else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
367             {
368                 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
369                 {
370                     // for a restricted decryption key (a parent), scheme
371                     // is required to be TPM_ALG_NULL
372                     if(scheme != TPM_ALG_NULL)
373                         return TPM_RCS_SCHEME;
374                 }
375                 else
376                 {
377                     // For an unrestricted decryption key, the scheme has to
378                     // be a valid scheme or TPM_ALG_NULL
379                     if(scheme != TPM_ALG_NULL &&
380                        !CryptIsAsymDecryptScheme(publicArea->type, scheme))
381                         return TPM_RCS_SCHEME;
382                 }
383             }
384             if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
385                || !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
386             {
387                 // For an asymmetric key that is not a parent, the symmetric
388                 // algorithms must be TPM_ALG_NULL
389                 if(symAlgs->algorithm != TPM_ALG_NULL)
390                     return TPM_RCS_SYMMETRIC;
391             }
392             // Special checks for an ECC key
```



```
393 #if     ALG_ECC
394             if(publicArea->type == TPM_ALG_ECC)
395             {
396                 TPM_ECC_CURVE            curveID;
397                 const TPMT_ECC_SCHEME   *curveScheme;
398
399                 curveID = publicArea->parameters.eccDetail.curveID;
400                 curveScheme = CryptGetCurveSignScheme(curveID);
401                 // The curveId must be valid or the unmarshaling is busted.
402                 pAssert(curveScheme != NULL);
403
404                 // If the curveID requires a specific scheme, then the key must
405                 // select the same scheme
406                 if(curveScheme->scheme != TPM_ALG_NULL)
407                 {
408                     TPMS_ECC_PARMS      *ecc = &publicArea->parameters.eccDetail;
409                     if(scheme != curveScheme->scheme)
410                         return TPM_RCS_SCHEME;
411                     // The scheme can allow any hash, or not...
412                     if(curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
413                        && (ecc->scheme.details.anySig.hashAlg
414                            != curveScheme->details.anySig.hashAlg))
415                         return TPM_RCS_SCHEME;
416                 }
417                 // For now, the KDF must be TPM_ALG_NULL
418                 if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
419                     return TPM_RCS_KDF;
420             }
421 #endif
422             break;
423     }
424     // If this is a restricted decryption key with symmetric algorithms, then it
425     // is an ordinary parent (not a derivation parent). It needs to specific
426     // symmetric algorithms other than TPM_ALG_NULL
427     if(symAlgs != NULL
428        && IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
429        && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
430     {
431         if(symAlgs->algorithm == TPM_ALG_NULL)
432             return TPM_RCS_SYMMETRIC;
433 #if 0       //??
434 // This next check is under investigation. Need to see if it will break Windows
435 // before it is enabled. If it does not, then it should be default because a
436 // the mode used with a parent is always CFB and Part 2 indicates as much.
437         if(symAlgs->mode.sym != TPM_ALG_CFB)
438             return TPM_RCS_MODE;
439 #endif
440         // If this parent is not duplicable, then the symmetric algorithms
441         // (encryption and hash) must match those of its parent
442         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
443            && (parentObject != NULL))
444         {
445             if(publicArea->nameAlg != parentObject->publicArea.nameAlg)
446                 return TPM_RCS_HASH;
447             if(!MemoryEqual(symAlgs, &parentObject->publicArea.parameters,
448                             sizeof(TPMT_SYM_DEF_OBJECT)))
449                 return TPM_RCS_SYMMETRIC;
450         }
451     }
452     return TPM_RC_SUCCESS;
453 }
```

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()

```
454  TPM_RC
455  PublicAttributesValidation(
456      OBJECT          *parentObject,  // IN: input parent object
457      TPMT_PUBLIC     *publicArea     // IN: public area of the object
458      )
459  {
460      TPMA_OBJECT      attributes = publicArea->objectAttributes;
461      TPMA_OBJECT      parentAttributes = {0};
462  //
463      if(parentObject != NULL)
464          parentAttributes = parentObject->publicArea.objectAttributes;
465      if(publicArea->nameAlg == TPM_ALG_NULL)
466          return TPM_RCS_HASH;
467      // If there is an authPolicy, it needs to be the size of the digest produced
468      // by the nameAlg of the object
469      if((publicArea->authPolicy.t.size != 0
470          && (publicArea->authPolicy.t.size
471              != CryptHashGetDigestSize(publicArea->nameAlg))))
472          return TPM_RCS_SIZE;
473      // If the parent is fixedTPM (including a Primary Object) the object must have
474      // the same value for fixedTPM and fixedParent
475      if(parentObject == NULL
476         || IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
477      {
478          if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
479             != IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
480              return TPM_RCS_ATTRIBUTES;
481      }
482      else
483      {
484          // The parent is not fixedTPM so the object can't be fixedTPM
485          if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
486              return  TPM_RCS_ATTRIBUTES;
487      }
488      // See if sign and decrypt are the same
489      if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
490         == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
491      {
492          // a restricted key cannot have both SET or both CLEAR
493          if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
494              return TPM_RC_ATTRIBUTES;
495          // only a data object may have both sign and decrypt CLEAR
496          // BTW, since we know that decrypt==sign, no need to check both
```



```
497         if(publicArea->type != TPM_ALG_KEYEDHASH
498            && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
499             return TPM_RC_ATTRIBUTES;
500     }
501     // If the object can't be duplicated (directly or indirectly) then there
502     // is no justification for having encryptedDuplication SET
503     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
504        && IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
505         return TPM_RCS_ATTRIBUTES;
506     // If a parent object has fixedTPM CLEAR, the child must have the
507     // same encryptedDuplication value as its parent.
508     // Primary objects are considered to have a fixedTPM parent (the seeds).
509     if(parentObject != NULL
510        && !IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
511     {
512         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication)
513            != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, encryptedDuplication))
514             return TPM_RCS_ATTRIBUTES;
515     }
516     // Special checks for derived objects
517     if((parentObject != NULL) && (parentObject->attributes.derivation == SET))
518     {
519         // A derived object has the same settings for fixedTPM as its parent
520         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
521            != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
522             return TPM_RCS_ATTRIBUTES;
523         // A derived object is required to be fixedParent
524         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))
525             return TPM_RCS_ATTRIBUTES;
526     }
527     return SchemeChecks(parentObject, publicArea);
528 }
```

7.6.3.6 FillInCreationData()

Fill in creation data for an object.

```
529 void
530 FillInCreationData(
531     TPMI_DH_OBJECT          parentHandle,  // IN: handle of parent
532     TPMI_ALG_HASH           nameHashAlg,   // IN: name hash algorithm
533     TPML_PCR_SELECTION     *creationPCR,   // IN: PCR selection
534     TPM2B_DATA             *outsideData,   // IN: outside data
535     TPM2B_CREATION_DATA    *outCreation,   // OUT: creation data for output
536     TPM2B_DIGEST           *creationDigest // OUT: creation digest
537     )
538 {
539     BYTE                creationBuffer[sizeof(TPMS_CREATION_DATA)];
540     BYTE               *buffer;
541     HASH_STATE          hashState;
542 //
543     // Fill in TPMS_CREATION_DATA in outCreation
544
545     // Compute PCR digest
546     PCRComputeCurrentDigest(nameHashAlg, creationPCR,
547                             &outCreation->creationData.pcrDigest);
548
549     // Put back PCR selection list
550     outCreation->creationData.pcrSelect = *creationPCR;
551
552     // Get locality
553     outCreation->creationData.locality
554         = LocalityGetAttributes(_plat__LocalityGet());
555     outCreation->creationData.parentNameAlg = TPM_ALG_NULL;
```

```
556
557         // If the parent is either a primary seed or TPM_ALG_NULL, then  the Name
558         // and QN of the parent are the parent's handle.
559         if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
560         {
561             buffer = &outCreation->creationData.parentName.t.name[0];
562             outCreation->creationData.parentName.t.size =
563                 TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
564             // For a primary or temporary object, the parent name (a handle) and the
565             // parent's QN are the same
566             outCreation->creationData.parentQualifiedName
567                 = outCreation->creationData.parentName;
568         }
569         else          // Regular object
570         {
571             OBJECT          *parentObject = HandleToObject(parentHandle);
572     //
573             // Set name algorithm
574             outCreation->creationData.parentNameAlg =
575                 parentObject->publicArea.nameAlg;
576             // Copy parent name
577             outCreation->creationData.parentName = parentObject->name;
578
579             // Copy parent qualified name
580             outCreation->creationData.parentQualifiedName =
581                 parentObject->qualifiedName;
582         }
583         // Copy outside information
584         outCreation->creationData.outsideInfo = *outsideData;
585
586         // Marshal creation data to canonical form
587         buffer = creationBuffer;
588         outCreation->size = TPMS_CREATION_DATA_Marshal(&outCreation->creationData,
589                                                        &buffer, NULL);
590         // Compute hash for creation field in public template
591         creationDigest->t.size = CryptHashStart(&hashState, nameHashAlg);
592         CryptDigestUpdate(&hashState, outCreation->size, creationBuffer);
593         CryptHashEnd2B(&hashState, &creationDigest->b);
594
595         return;
596     }
```

7.6.3.7 GetSeedForKDF()

Get a seed for KDF. The KDF for encryption and HMAC key use the same seed.

```
597     const TPM2B *
598     GetSeedForKDF(
599         OBJECT          *protector         // IN: the protector handle
600         )
601     {
602         // Get seed for encryption key.  Use input seed if provided.
603         // Otherwise, using protector object's seedValue.  TPM_RH_NULL is the only
604         // exception that we may not have a loaded object as protector.  In such a
605         // case, use nullProof as seed.
606         if(protector == NULL)
607             return &gr.nullProof.b;
608         else
609             return &protector->sensitive.seedValue.b;
610     }
```


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

```
611 UINT16
612 ProduceOuterWrap(
613     OBJECT          *protector,     // IN: The handle of the object that provides
614                                     //     protection.  For object, it is parent
615                                     //     handle. For credential, it is the handle
616                                     //     of encrypt object.
617     TPM2B           *name,          // IN: the name of the object
618     TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
619     TPM2B           *seed,          // IN: an external seed may be provided for
620                                     //     duplication blob. For non duplication
621                                     //     blob, this parameter should be NULL
622     BOOL             useIV,         // IN: indicate if an IV is used
623     UINT16           dataSize,      // IN: the size of sensitive data, excluding the
624                                     //     leading integrity buffer size or the
625                                     //     optional iv size
626     BYTE            *outerBuffer    // IN/OUT: outer buffer with sensitive data in
627                                     //     it
628     )
629 {
630     TPM_ALG_ID      symAlg;
631     UINT16          keyBits;
632     TPM2B_SYM_KEY   symKey;
633     TPM2B_IV        ivRNG;          // IV from RNG
634     TPM2B_IV        *iv = NULL;
635     UINT16          ivSize = 0;     // size of iv area, including the size field
636     BYTE            *sensitiveData; // pointer to the sensitive data
637     TPM2B_DIGEST    integrity;
638     UINT16          integritySize;
639     BYTE            *buffer;        // Auxiliary buffer pointer
640 //
641     // Compute the beginning of sensitive data.  The outer integrity should
642     // always exist if this function is called to make an outer wrap
643     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
644     sensitiveData = outerBuffer + integritySize;
645
646     // If iv is used, adjust the pointer of sensitive data and add iv before it
647     if(useIV)
648     {
649         ivSize = GetIV2BSize(protector);
650
651         // Generate IV from RNG.  The iv data size should be the total IV area
652         // size minus the size of size field
653         ivRNG.t.size = ivSize - sizeof(UINT16);
654         CryptRandomGenerate(ivRNG.t.size, ivRNG.t.buffer);
655
656         // Marshal IV to buffer
657         buffer = sensitiveData;
658         TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
659
660         // adjust sensitive data starting after IV area
661         sensitiveData += ivSize;
662
```

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

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

```
684 TPM_RC
685 UnwrapOuter(
686     OBJECT          *protector,     // IN: The object that provides
687                                     //     protection.  For object, it is parent
688                                     //     handle. For credential, it is the
689                                     //     encrypt object.
690     TPM2B           *name,          // IN: the name of the object
691     TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
692     TPM2B           *seed,          // IN: an external seed may be provided for
693                                     //     duplication blob. For non duplication
694                                     //     blob, this parameter should be NULL.
695     BOOL             useIV,         // IN: indicates if an IV is used
696     UINT16           dataSize,      // IN: size of sensitive data in outerBuffer,
697                                     //     including the leading integrity buffer
698                                     //     size, and an optional iv area
699     BYTE            *outerBuffer    // IN/OUT: sensitive data
700     )
701 {
702     TPM_RC          result;
703     TPM_ALG_ID      symAlg = TPM_ALG_NULL;
704     TPM2B_SYM_KEY   symKey;
705     UINT16          keyBits = 0;
706     TPM2B_IV        ivIn;               // input IV retrieved from input buffer
707     TPM2B_IV        *iv = NULL;
708     BYTE            *sensitiveData;     // pointer to the sensitive data
```



```
709     TPM2B_DIGEST    integrityToCompare;
710     TPM2B_DIGEST    integrity;
711     INT32           size;
712 //
713     // Unmarshal integrity
714     sensitiveData = outerBuffer;
715     size = (INT32)dataSize;
716     result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
717     if(result == TPM_RC_SUCCESS)
718     {
719         // Compute integrity to compare
720         ComputeOuterIntegrity(name, protector, hashAlg, seed,
721                               (UINT16)size, sensitiveData,
722                               &integrityToCompare);
723         // Compare outer blob integrity
724         if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
725             return TPM_RCS_INTEGRITY;
726         // Get the symmetric algorithm parameters used for encryption
727         ComputeProtectionKeyParms(protector, hashAlg, name, seed,
728                                   &symAlg, &keyBits, &symKey);
729         // Retrieve IV if it is used
730         if(useIV)
731         {
732             result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
733             if(result == TPM_RC_SUCCESS)
734             {
735                 // The input iv size for CFB must match the encryption algorithm
736                 // block size
737                 if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
738                     result = TPM_RC_VALUE;
739                 else
740                     iv = &ivIn;
741             }
742         }
743     }
744     // If no errors, decrypt private in place. Since this function uses CFB,
745     // CryptSymmetricDecrypt() will not return any errors. It may fail but it will
746     // not return an error.
747     if(result == TPM_RC_SUCCESS)
748         CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
749                               symKey.t.buffer, iv, TPM_ALG_CFB,
750                               (UINT16)size, sensitiveData);
751     return result;
752 }
```

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.

```
753 static UINT16
754 MarshalSensitive(
755     BYTE                *buffer,            // OUT: receiving buffer
756     TPMT_SENSITIVE      *sensitive,         // IN: the sensitive area to marshal
757     TPMI_ALG_HASH        nameAlg            // IN:
758     )
759 {
760     BYTE                *sizeField = buffer;    // saved so that size can be
761                                                 // marshaled after it is known
762     UINT16               retVal;
763 //
764     // Pad the authValue if needed
765     MemoryPad2B(&sensitive->authValue.b, CryptHashGetDigestSize(nameAlg));
766     buffer += 2;
```

```
767
768     // Marshal the structure
769     retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
770
771     // Marshal the size
772     retVal = (UINT16)(retVal + UINT16_Marshal(&retVal, &sizeField, NULL));
773
774     return retVal;
775 }
```

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.

```
776 void
777 SensitiveToPrivate(
778     TPMT_SENSITIVE  *sensitive,     // IN: sensitive structure
779     TPM2B           *name,          // IN: the name of the object
780     OBJECT          *parent,        // IN: The parent object
781     TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  This
782                                     //     parameter is used when parentHandle is
783                                     //     NULL, in which case the object is
784                                     //     temporary.
785     TPM2B_PRIVATE   *outPrivate     // OUT: output private structure
786     )
787 {
788     BYTE                *sensitiveData;     // pointer to the sensitive data
789     UINT16              dataSize;           // data blob size
790     TPMI_ALG_HASH       hashAlg;            // hash algorithm for integrity
791     UINT16              integritySize;
792     UINT16              ivSize;
793 //
794     pAssert(name != NULL && name->size != 0);
795
796     // Find the hash algorithm for integrity computation
797     if(parent == NULL)
798     {
799         // For Temporary Object, using self name algorithm
800         hashAlg = nameAlg;
801     }
802     else
803     {
804         // Otherwise, using parent's name algorithm
805         hashAlg = ObjectGetNameAlg(parent);
806     }
807     // Starting of sensitive data without wrappers
808     sensitiveData = outPrivate->t.buffer;
809
810     // Compute the integrity size
811     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
812
813     // Reserve space for integrity
814     sensitiveData += integritySize;
815
816     // Get iv size
817     ivSize = GetIV2BSize(parent);
818
819     // Reserve space for iv
820     sensitiveData += ivSize;
```



```
821
822     // Marshal the sensitive area including authValue size adjustments.
823     dataSize = MarshalSensitive(sensitiveData, sensitive, nameAlg);
824
825     //Produce outer wrap, including encryption and HMAC
826     outPrivate->t.size = ProduceOuterWrap(parent, name, hashAlg, NULL,
827                                           TRUE, dataSize, outPrivate->t.buffer);
828     return;
829 }
```

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

```
830 TPM_RC
831 PrivateToSensitive(
832     TPM2B           *inPrivate,     // IN: input private structure
833     TPM2B           *name,          // IN: the name of the object
834     OBJECT          *parent,        // IN: parent object
835     TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  It is
836                                     //     passed separately because we only pass
837                                     //     name, rather than the whole public area
838                                     //     of the object.  This parameter is used in
839                                     //     the following two cases: 1. primary
840                                     //     objects. 2. duplication blob with inner
841                                     //     wrap.  In other cases, this parameter
842                                     //     will be ignored
843     TPMT_SENSITIVE  *sensitive      // OUT: sensitive structure
844     )
845 {
846     TPM_RC          result;
847     BYTE            *buffer;
848     INT32           size;
849     BYTE            *sensitiveData; // pointer to the sensitive data
850     UINT16          dataSize;
851     UINT16          dataSizeInput;
852     TPMI_ALG_HASH   hashAlg;        // hash algorithm for integrity
853     UINT16          integritySize;
854     UINT16          ivSize;
855 //
856     // Make sure that name is provided
857     pAssert(name != NULL && name->size != 0);
858
859     // Find the hash algorithm for integrity computation
860     if(parent == NULL)
861     {
862         // For Temporary Object, using self name algorithm
863         hashAlg = nameAlg;
```

```
    }
    else
    {
        // Otherwise, using parent's name algorithm
        hashAlg = ObjectGetNameAlg(parent);
    }
    // unwrap outer
    result = UnwrapOuter(parent, name, hashAlg, NULL, TRUE,
                         inPrivate->size, inPrivate->buffer);
    if(result != TPM_RC_SUCCESS)
        return result;
    // Compute the inner integrity size.
    integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);

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

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

    // Unmarshal input data size
    buffer = sensitiveData;
    size = (INT32)dataSize;
    result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
    if(result == TPM_RC_SUCCESS)
    {
        if((dataSizeInput + sizeof(UINT16)) != dataSize)
            result = TPM_RC_SENSITIVE;
        else
        {
            // Unmarshal sensitive buffer to sensitive structure
            result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
            if(result != TPM_RC_SUCCESS || size != 0)
            {
                result = TPM_RC_SENSITIVE;
            }
        }
    }
    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.
```



```
918         TPMT_SYM_DEF_OBJECT *symDef,        // IN: Symmetric key definition. If the
919                                             //     symmetric key algorithm is NULL,
920                                             //     no inner wrap should be applied.
921         TPM2B_DATA          *innerSymKey,   // IN/OUT: a symmetric key may be
922                                             //     provided to encrypt the inner
923                                             //     wrap of a duplication blob. May
924                                             //     be generated here if needed.
925         TPM2B_PRIVATE       *outPrivate     // OUT: output private structure
926         )
927     {
928         BYTE            *sensitiveData; // pointer to the sensitive data
929         TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;// The hash algorithm for outer wrap
930         TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;// The hash algorithm for inner wrap
931         UINT16          dataSize;       // data blob size
932         BOOL            doInnerWrap = FALSE;
933         BOOL            doOuterWrap = FALSE;
934     //
935         // Make sure that name is provided
936         pAssert(name != NULL && name->size != 0);
937
938         // Make sure symDef and innerSymKey are not NULL
939         pAssert(symDef != NULL && innerSymKey != NULL);
940
941         // Starting of sensitive data without wrappers
942         sensitiveData = outPrivate->t.buffer;
943
944         // Find out if inner wrap is required
945         if(symDef->algorithm != TPM_ALG_NULL)
946         {
947             doInnerWrap = TRUE;
948
949             // Use self nameAlg as inner hash algorithm
950             innerHash = nameAlg;
951
952             // Adjust sensitive data pointer
953             sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(innerHash);
954         }
955         // Find out if outer wrap is required
956         if(seed->size != 0)
957         {
958             doOuterWrap = TRUE;
959
960             // Use parent nameAlg as outer hash algorithm
961             outerHash = ObjectGetNameAlg(parent);
962
963             // Adjust sensitive data pointer
964             sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
965         }
966         // Marshal sensitive area
967         dataSize = MarshalSensitive(sensitiveData, sensitive, nameAlg);
968
969         // Apply inner wrap for duplication blob.  It includes both integrity and
970         // encryption
971         if(doInnerWrap)
972         {
973             BYTE            *innerBuffer = NULL;
974             BOOL            symKeyInput = TRUE;
975             innerBuffer = outPrivate->t.buffer;
976             // Skip outer integrity space
977             if(doOuterWrap)
978                 innerBuffer += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
979             dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
980                                              innerBuffer);
981             // Generate inner encryption key if needed
982             if(innerSymKey->t.size == 0)
983             {
```

```
984                 innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
985                 CryptRandomGenerate(innerSymKey->t.size, innerSymKey->t.buffer);
986
987                 // TPM generates symmetric encryption.  Set the flag to FALSE
988                 symKeyInput = FALSE;
989             }
990             else
991             {
992                 // assume the input key size should matches the symmetric definition
993                 pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
994             }
995
996             // Encrypt inner buffer in place
997             CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
998                                   symDef->keyBits.sym, innerSymKey->t.buffer, NULL,
999                                   TPM_ALG_CFB, dataSize, innerBuffer);
1000
1001            // If the symmetric encryption key is imported, clear the buffer for
1002            // output
1003            if(symKeyInput)
1004                innerSymKey->t.size = 0;
1005        }
1006        // Apply outer wrap for duplication blob.  It includes both integrity and
1007        // encryption
1008        if(doOuterWrap)
1009        {
1010            dataSize = ProduceOuterWrap(parent, name, outerHash, seed, FALSE,
1011                                        dataSize, outPrivate->t.buffer);
1012        }
1013        // Data size for output
1014        outPrivate->t.size = dataSize;
1015
1016        return;
1017    }
```

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

```
1018    TPM_RC
1019    DuplicateToSensitive(
1020        TPM2B                *inPrivate,      // IN: input private structure
1021        TPM2B                *name,           // IN: the name of the object
1022        OBJECT               *parent,         // IN: the parent
1023        TPM_ALG_ID            nameAlg,        // IN: hash algorithm in public area.
1024        TPM2B                *seed,           // IN: an external seed may be provided.
1025                                               //     If external seed is provided with
1026                                               //     size of 0, no outer wrap is
1027                                               //     applied
1028        TPMT_SYM_DEF_OBJECT *symDef,          // IN: Symmetric key definition. If the
```

```
1029                                            //     symmetric key algorithm is NULL,
1030                                            //     no inner wrap is applied
1031        TPM2B               *innerSymKey,   // IN: a symmetric key may be provided
1032                                            //     to decrypt the inner wrap of a
1033                                            //     duplication blob.
1034        TPMT_SENSITIVE      *sensitive      // OUT: sensitive structure
1035        )
1036    {
1037        TPM_RC               result;
1038        BYTE                *buffer;
1039        INT32                size;
1040        BYTE                *sensitiveData; // pointer to the sensitive data
1041        UINT16               dataSize;
1042        UINT16               dataSizeInput;
1043    //
1044        // Make sure that name is provided
1045        pAssert(name != NULL && name->size != 0);
1046
1047        // Make sure symDef and innerSymKey are not NULL
1048        pAssert(symDef != NULL && innerSymKey != NULL);
1049
1050        // Starting of sensitive data
1051        sensitiveData = inPrivate->buffer;
1052        dataSize = inPrivate->size;
1053
1054        // Find out if outer wrap is applied
1055        if(seed->size != 0)
1056        {
1057            // Use parent nameAlg as outer hash algorithm
1058            TPMI_ALG_HASH   outerHash = parent->publicArea.nameAlg;
1059
1060            result = UnwrapOuter(parent, name, outerHash, seed, FALSE,
1061                                 dataSize, sensitiveData);
1062            if(result != TPM_RC_SUCCESS)
1063                return result;
1064            // Adjust sensitive data pointer and size
1065            sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1066            dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1067        }
1068        // Find out if inner wrap is applied
1069        if(symDef->algorithm != TPM_ALG_NULL)
1070        {
1071            // assume the input key size matches the symmetric definition
1072            pAssert(innerSymKey->size == (symDef->keyBits.sym + 7) / 8);
1073
1074            // Decrypt inner buffer in place
1075            CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
1076                                  symDef->keyBits.sym, innerSymKey->buffer, NULL,
1077                                  TPM_ALG_CFB, dataSize, sensitiveData);
1078            // Check inner integrity
1079            result = CheckInnerIntegrity(name, nameAlg, dataSize, sensitiveData);
1080            if(result != TPM_RC_SUCCESS)
1081                return result;
1082            // Adjust sensitive data pointer and size
1083            sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1084            dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1085        }
1086        // Unmarshal input data size
1087        buffer = sensitiveData;
1088        size = (INT32)dataSize;
1089        result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1090        if(result == TPM_RC_SUCCESS)
1091        {
1092            if((dataSizeInput + sizeof(UINT16)) != dataSize)
1093                result = TPM_RC_SIZE;
1094            else
```

```
1095            {
1096                // Unmarshal sensitive buffer to sensitive structure
1097                result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1098
1099                // if the results is OK make sure that all the data was unmarshaled
1100                if(result == TPM_RC_SUCCESS && size != 0)
1101                    result = TPM_RC_SIZE;
1102            }
1103        }
1104        return result;
1105    }
```

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

```
1106    void
1107    SecretToCredential(
1108        TPM2B_DIGEST        *secret,        // IN: secret information
1109        TPM2B               *name,          // IN: the name of the object
1110        TPM2B               *seed,          // IN: an external seed.
1111        OBJECT              *protector,     // IN: the protector
1112        TPM2B_ID_OBJECT     *outIDObject    // OUT: output credential
1113        )
1114    {
1115        BYTE                *buffer;        // Auxiliary buffer pointer
1116        BYTE                *sensitiveData; // pointer to the sensitive data
1117        TPMI_ALG_HASH        outerHash;     // The hash algorithm for outer wrap
1118        UINT16               dataSize;      // data blob size
1119    //
1120        pAssert(secret != NULL && outIDObject != NULL);
1121
1122        // use protector's name algorithm as outer hash
1123        outerHash = ObjectGetNameAlg(protector);
1124
1125        // Marshal secret area to credential buffer, leave space for integrity
1126        sensitiveData = outIDObject->t.credential
1127            + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1128    // Marshal secret area
1129        buffer = sensitiveData;
1130        dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1131
1132        // Apply outer wrap
1133        outIDObject->t.size = ProduceOuterWrap(protector, name, outerHash, seed, FALSE,
1134                                               dataSize, outIDObject->t.credential);
1135        return;
1136    }
```

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

```
TPM_RC
CredentialToSecret(
    TPM2B               *inIDObject,     // IN: input credential blob
    TPM2B               *name,           // IN: the name of the object
    TPM2B               *seed,           // IN: an external seed.
    OBJECT              *protector,      // IN: the protector
    TPM2B_DIGEST        *secret          // OUT: secret information
    )
{
    TPM_RC                   result;
    BYTE                    *buffer;
    INT32                    size;
    TPMI_ALG_HASH            outerHash;     // The hash algorithm for outer wrap
    BYTE                    *sensitiveData; // pointer to the sensitive data
    UINT16                   dataSize;
//
    // use protector's name algorithm as outer hash
    outerHash = ObjectGetNameAlg(protector);

    // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
    result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
                         inIDObject->size, inIDObject->buffer);
    if(result == TPM_RC_SUCCESS)
    {
        // Compute the beginning of sensitive data
        sensitiveData = inIDObject->buffer
            + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
        dataSize = inIDObject->size
            - (sizeof(UINT16) + CryptHashGetDigestSize(outerHash));
        // Unmarshal secret buffer to TPM2B_DIGEST structure
        buffer = sensitiveData;
        size = (INT32)dataSize;
        result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);

        // If there were no other unmarshaling errors, make sure that the
        // expected amount of data was recovered
        if(result == TPM_RC_SUCCESS && size != 0)
            return TPM_RC_SIZE;
    }
    return result;
}
```

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.

```
UINT16
```

```
1179 MemoryRemoveTrailingZeros(
1180     TPM2B_AUTH      *auth           // IN/OUT: value to adjust
1181     )
1182 {
1183     while((auth->t.size > 0) && (auth->t.buffer[auth->t.size - 1] == 0))
1184         auth->t.size--;
1185     return auth->t.size;
1186 }
```

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.

```
1187 TPM_RC
1188 SetLabelAndContext(
1189     TPMS_DERIVE             *labelContext,  // IN/OUT: the recovered label and
1190                                             //         context
1191     TPM2B_SENSITIVE_DATA    *sensitive      // IN: the sensitive data
1192     )
1193 {
1194     TPMS_DERIVE              sensitiveValue;
1195     TPM_RC                   result;
1196     INT32                    size;
1197     BYTE                    *buff;
1198 //
1199     // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
1200     // If there is something to unmarshal...
1201     if(sensitive->t.size != 0)
1202     {
1203         size = sensitive->t.size;
1204         buff = sensitive->t.buffer;
1205         result = TPMS_DERIVE_Unmarshal(&sensitiveValue, &buff, &size);
1206         if(result != TPM_RC_SUCCESS)
1207             return result;
1208         // If there was a label in the public area leave it there, otherwise, copy
1209         // the new value
1210         if(labelContext->label.t.size == 0)
1211             MemoryCopy2B(&labelContext->label.b, &sensitiveValue.label.b,
1212                          sizeof(labelContext->label.t.buffer));
1213         // if there was a context string in publicArea, it overrides
1214         if(labelContext->context.t.size == 0)
1215             MemoryCopy2B(&labelContext->context.b, &sensitiveValue.context.b,
1216                          sizeof(labelContext->label.t.buffer));
1217     }
1218     return TPM_RC_SUCCESS;
1219 }
```

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_PUBLIC 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*.

```
1220 TPM_RC
1221 UnmarshalToPublic(
1222     TPMT_PUBLIC         *tOut,       // OUT: output
1223     TPM2B_TEMPLATE      *tIn,        // IN:
1224     BOOL                 derivation, // IN: indicates if this is for a derivation
1225     TPMS_DERIVE         *labelContext// OUT: label and context if derivation
1226     )
```



```
1227 {
1228     BYTE                *buffer = tIn->t.buffer;
1229     INT32                size = tIn->t.size;
1230     TPM_RC               result;
1231 //
1232     // make sure that tOut is zeroed so that there are no remnants from previous
1233     // uses
1234     MemorySet(tOut, 0, sizeof(TPMT_PUBLIC));
1235     // Unmarshal the components of the TPMT_PUBLIC up to the unique field
1236     result = TPMI_ALG_PUBLIC_Unmarshal(&tOut->type, &buffer, &size);
1237     if(result != TPM_RC_SUCCESS)
1238         return result;
1239     result = TPMI_ALG_HASH_Unmarshal(&tOut->nameAlg, &buffer, &size, FALSE);
1240     if(result != TPM_RC_SUCCESS)
1241         return result;
1242     result = TPMA_OBJECT_Unmarshal(&tOut->objectAttributes, &buffer, &size);
1243     if(result != TPM_RC_SUCCESS)
1244         return result;
1245     result = TPM2B_DIGEST_Unmarshal(&tOut->authPolicy, &buffer, &size);
1246     if(result != TPM_RC_SUCCESS)
1247         return result;
1248     result = TPMU_PUBLIC_PARMS_Unmarshal(&tOut->parameters, &buffer, &size,
1249                                          tOut->type);
1250     if(result != TPM_RC_SUCCESS)
1251         return result;
1252     // Now unmarshal a TPMS_DERIVE if this is for derivation
1253     if(derivation)
1254         result = TPMS_DERIVE_Unmarshal(labelContext, &buffer, &size);
1255     else
1256         // otherwise, unmarshal a TPMU_PUBLIC_ID
1257         result = TPMU_PUBLIC_ID_Unmarshal(&tOut->unique, &buffer, &size,
1258                                           tOut->type);
1259     // Make sure the template was used up
1260     if((result == TPM_RC_SUCCESS) && (size != 0))
1261         result = TPM_RC_SIZE;
1262     return result;
1263 }
```

7.6.3.20 ObjectSetExternal()

Set the external attributes for an object.

```
1264 void
1265 ObjectSetExternal(
1266     OBJECT      *object
1267     )
1268 {
1269     object->attributes.external = SET;
1270 }
```

7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c)

```
1  #include "Tpm.h"
2  #include "EncryptDecrypt_fp.h"
3  #include "EncryptDecrypt_spt_fp.h"
4  #if 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

```
5  TPM_RC
6  EncryptDecryptShared(
7      TPMI_DH_OBJECT          keyHandleIn,
8      TPMI_YES_NO             decryptIn,
9      TPMI_ALG_SYM_MODE       modeIn,
10     TPM2B_IV                *ivIn,
11     TPM2B_MAX_BUFFER        *inData,
12     EncryptDecrypt_Out      *out
13     )
14 {
15     OBJECT                 *symKey;
16     UINT16                  keySize;
17     UINT16                  blockSize;
18     BYTE                   *key;
19     TPM_ALG_ID              alg;
20     TPM_ALG_ID              mode;
21     TPM_RC                  result;
22     BOOL                    OK;
23 // Input Validation
24     symKey = HandleToObject(keyHandleIn);
25     mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;
26
27     // The input key should be a symmetric key
28     if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
29         return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
30     // The key must be unrestricted and allow the selected operation
31     OK = !IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
32                        TPMA_OBJECT, restricted);
33     if(YES == decryptIn)
34         OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
35                                 TPMA_OBJECT, decrypt);
36     else
37         OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
38                                 TPMA_OBJECT, sign);
39     if(!OK)
40         return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
41
42     // Make sure that key is an encrypt/decrypt key and not SMAC
43     if(!CryptSymModeIsValid(mode, TRUE))
44        return TPM_RCS_MODE + RC_EncryptDecrypt_keyHandle;
45
46     // If the key mode is not TPM_ALG_NULL...
47     // or TPM_ALG_NULL
48     if(mode != TPM_ALG_NULL)
49     {
50         // then the input mode has to be TPM_ALG_NULL or the same as the key
```



```
51          if((modeIn != TPM_ALG_NULL) && (modeIn != mode))
52              return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
53      }
54      else
55      {
56          // if the key mode is null, then the input can't be null
57          if(modeIn == TPM_ALG_NULL)
58              return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
59          mode = modeIn;
60      }
61      // The input iv for ECB mode should be an Empty Buffer.  All the other modes
62      // should have an iv size same as encryption block size
63      keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
64      alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
65      blockSize = CryptGetSymmetricBlockSize(alg, keySize);
66
67      // reverify the algorithm. This is mainly to keep static analysis tools happy
68      if(blockSize == 0)
69          return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
70
71      // Note: When an algorithm is not supported by a TPM, the TPM_ALG_xxx for that
72      // algorithm is not defined. However, it is assumed that the ALG_xxx_VALUE for
73      // the algorithm is always defined. Both have the same numeric value.
74      // ALG_xxx_VALUE is used here so that the code does not get cluttered with
75      // #ifdef's. Having this check does not mean that the algorithm is supported.
76      // If it was not supported the unmarshaling code would have rejected it before
77      // this function were called. This means that, depending on the implementation,
78      // the check could be redundant but it doesn't hurt.
79      if(((mode == ALG_ECB_VALUE) && (ivIn->t.size != 0))
80         || ((mode != ALG_ECB_VALUE) && (ivIn->t.size != blockSize)))
81          return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
82
83      // The input data size of CBC mode or ECB mode must be an even multiple of
84      // the symmetric algorithm's block size
85      if(((mode == ALG_CBC_VALUE) || (mode == ALG_ECB_VALUE))
86         && ((inData->t.size % blockSize) != 0))
87          return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
88
89      // Copy IV
90      // Note: This is copied here so that the calls to the encrypt/decrypt functions
91      // will modify the output buffer, not the input buffer
92      out->ivOut = *ivIn;
93
94  // Command Output
95      key = symKey->sensitive.sensitive.sym.t.buffer;
96      // For symmetric encryption, the cipher data size is the same as plain data
97      // size.
98      out->outData.t.size = inData->t.size;
99      if(decryptIn == YES)
100     {
101         // Decrypt data to output
102         result = CryptSymmetricDecrypt(out->outData.t.buffer, alg, keySize, key,
103                                        &(out->ivOut), mode, inData->t.size,
104                                        inData->t.buffer);
105     }
106     else
107     {
108         // Encrypt data to output
109         result = CryptSymmetricEncrypt(out->outData.t.buffer, alg, keySize, key,
110                                        &(out->ivOut), mode, inData->t.size,
111                                        inData->t.buffer);
112     }
113     return result;
114 }
115 #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))
```



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

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(1)	command code audit status was changed
FALSE(0)	command code audit status was not changed

```
38  BOOL
39  CommandAuditSet(
40      TPM_CC            commandCode     // IN: command code
41      )
42  {
43      COMMAND_INDEX         commandIndex = CommandCodeToCommandIndex(commandCode);
44
45      // Only SET a bit if the corresponding command is implemented
46      if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
47      {
48          // Can't audit shutdown
49          if(commandCode != TPM_CC_Shutdown)
50          {
51              if(!TEST_BIT(commandIndex, gp.auditCommands))
52              {
53                  // Set bit
54                  SET_BIT(commandIndex, gp.auditCommands);
55                  return TRUE;
56              }
57          }
58      }
59      // No change
60      return FALSE;
61  }
```

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(1)	command code audit status was changed
FALSE(0)	command code audit status was not changed

```
BOOL
CommandAuditClear(
    TPM_CC            commandCode     // IN: command code
    )
{
    COMMAND_INDEX        commandIndex = CommandCodeToCommandIndex(commandCode);

    // Do nothing if the command is not implemented
    if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
    {
        // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
        // cleared
        if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
        {
            if(TEST_BIT(commandIndex, gp.auditCommands))
            {
                // Clear bit
                CLEAR_BIT(commandIndex, gp.auditCommands);
                return TRUE;
            }
        }
    }
    // No change
    return FALSE;
}
```

8.1.3.5 CommandAuditIsRequired()

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

Return Value	Meaning
TRUE(1)	command is audited
FALSE(0)	command is not audited

```
BOOL
CommandAuditIsRequired(
    COMMAND_INDEX     commandIndex    // IN: command index
    )
{
    // Check the bit map.  If the bit is SET, command audit is required
    return(TEST_BIT(commandIndex, gp.auditCommands));
}
```

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

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

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

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

    // Find the implemented command that has a command code that is the same or
    // higher than the input
    // Collect audit commands
    for(commandIndex = GetClosestCommandIndex(commandCode);
    commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
        commandIndex = GetNextCommandIndex(commandIndex))
    {
        if(CommandAuditIsRequired(commandIndex))
        {
            if(commandList->count < count)
            {
                // If we have not filled up the return list, add this command
                // code to its
                TPM_CC      cc = GET_ATTRIBUTE(s_ccAttr[commandIndex],
                                               TPMA_CC, commandIndex);
                if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
                    cc += (1 << 29);
                commandList->commandCodes[commandList->count] = cc;
                commandList->count++;
            }
            else
            {
                // If the return list is full but we still have command
                // available, report this and stop iterating
                more = YES;
                break;
            }
        }
    }

    return more;
}
```

8.1.3.7 CommandAuditGetDigest

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

```
void
CommandAuditGetDigest(
```

```
145         TPM2B_DIGEST    *digest         // OUT: command digest
146         )
147     {
148         TPM_CC                      commandCode;
149         COMMAND_INDEX               commandIndex;
150         HASH_STATE                  hashState;
151
152         // Start hash
153         digest->t.size = CryptHashStart(&hashState, gp.auditHashAlg);
154
155         // Add command code
156         for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
157         {
158             if(CommandAuditIsRequired(commandIndex))
159             {
160                 commandCode = GetCommandCode(commandIndex);
161                 CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
162             }
163         }
164
165         // Complete hash
166         CryptHashEnd2B(&hashState, &digest->b);
167
168         return;
169     }
```

8.2 DA.c

8.2.1 Introduction

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

8.2.2 Includes and Data Definitions

```
1  #define DA_C
2  #include "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().

```
3  void
4  DAPreInstall_Init(
5      void
6      )
7  {
8      gp.failedTries = 0;
9      gp.maxTries = 3;
10     gp.recoveryTime = 1000;          // in seconds (~16.67 minutes)
11     gp.lockoutRecovery = 1000;       // in seconds
12     gp.lockOutAuthEnabled = TRUE;    // Use of lockoutAuth is enabled
13
14     // Record persistent DA parameter changes to NV
15     NV_SYNC_PERSISTENT(failedTries);
16     NV_SYNC_PERSISTENT(maxTries);
17     NV_SYNC_PERSISTENT(recoveryTime);
18     NV_SYNC_PERSISTENT(lockoutRecovery);
19     NV_SYNC_PERSISTENT(lockOutAuthEnabled);
20
21     return;
22 }
```

8.2.3.2 DAStartup()

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

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

```
23 void
24 DAStartup(
25     STARTUP_TYPE     type            // IN: startup type
26     )
27 {
28     NOT_REFERENCED(type);
29 #if !ACCUMULATE_SELF_HEAL_TIMER
30     _plat__TimerWasReset();
```

```
31      s_selfHealTimer = 0;
32      s_lockoutTimer = 0;
33  #else
34      if(_plat__TimerWasReset())
35      {
36          if(!NV_IS_ORDERLY)
37          {
38              // If shutdown was not orderly, then don't really know if go.time has
39              // any useful value so reset the timer to 0. This is what the tick
40              // was reset to
41              s_selfHealTimer = 0;
42              s_lockoutTimer = 0;
43          }
44          else
45          {
46              // If we know how much time was accumulated at the last orderly shutdown
47              // subtract that from the saved timer values so that they effectively
48              // have the accumulated values
49              s_selfHealTimer -= go.time;
50              s_lockoutTimer -= go.time;
51          }
52      }
53  #endif
54
55      // For any Startup(), if lockoutRecovery is 0, enable use of lockoutAuth.
56      if(gp.lockoutRecovery == 0)
57      {
58          gp.lockOutAuthEnabled = TRUE;
59          // Record the changes to NV
60          NV_SYNC_PERSISTENT(lockOutAuthEnabled);
61      }
62
63      // If DA has not been disabled and the previous shutdown is not orderly
64      // failedTries is not already at its maximum then increment 'failedTries'
65      if(gp.recoveryTime != 0
66         && gp.failedTries < gp.maxTries
67         && !IS_ORDERLY(g_prevOrderlyState))
68      {
69  #if USE_DA_USED
70          gp.failedTries += g_daUsed;
71          g_daUsed = FALSE;
72  #else
73          gp.failedTries++;
74  #endif
75          // Record the change to NV
76          NV_SYNC_PERSISTENT(failedTries);
77      }
78      // Before Startup, the TPM will not do clock updates. At startup, need to
79      // do a time update which will do the DA update.
80      TimeUpdate();
81
82      return;
83  }
```

8.2.3.3 DARegisterFailure()

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

```
84  void
85  DARegisterFailure(
86      TPM_HANDLE       handle          // IN: handle for failure
87      )
```



```
88  {
89      // Reset the timer associated with lockout if the handle is the lockoutAuth.
90      if(handle == TPM_RH_LOCKOUT)
91          s_lockoutTimer = g_time;
92      else
93          s_selfHealTimer = g_time;
94      return;
95  }
```

8.2.3.4 DASelfHeal()

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

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

```
96  void
97  DASelfHeal(
98      void
99      )
100 {
101     // Regular authorization self healing logic
102     // If no failed authorization tries, do nothing.  Otherwise, try to
103     // decrease failedTries
104     if(gp.failedTries != 0)
105     {
106         // if recovery time is 0, DA logic has been disabled.  Clear failed tries
107         // immediately
108         if(gp.recoveryTime == 0)
109         {
110             gp.failedTries = 0;
111             // Update NV record
112             NV_SYNC_PERSISTENT(failedTries);
113         }
114         else
115         {
116             UINT64          decreaseCount;
117 #if 0 // Errata eliminates this code
118             // In the unlikely event that failedTries should become larger than
119             // maxTries
120             if(gp.failedTries > gp.maxTries)
121                 gp.failedTries = gp.maxTries;
122 #endif
123             // How much can failedTries be decreased
124
125             // Cast s_selfHealTimer to an int in case it became negative at
126             // startup
127             decreaseCount = ((g_time - (INT64)s_selfHealTimer) / 1000)
128                 / gp.recoveryTime;
129
130             if(gp.failedTries <= (UINT32)decreaseCount)
131                 // should not set failedTries below zero
132                 gp.failedTries = 0;
133             else
134                 gp.failedTries -= (UINT32)decreaseCount;
135
136             // the cast prevents overflow of the product
137             s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
138             if(decreaseCount != 0)
139                 // If there was a change to the failedTries, record the changes
140                 // to NV
141                 NV_SYNC_PERSISTENT(failedTries);
142         }
143     }
```

```
144
145         // LockoutAuth self healing logic
146         // If lockoutAuth is enabled, do nothing.  Otherwise, try to see if we
147         // may enable it
148         if(!gp.lockOutAuthEnabled)
149         {
150             // if lockout authorization recovery time is 0, a reboot is required to
151             // re-enable use of lockout authorization.  Self-healing would not
152             // apply in this case.
153             if(gp.lockoutRecovery != 0)
154             {
155                 if(((g_time - (INT64)s_lockoutTimer) / 1000) >= gp.lockoutRecovery)
156                 {
157                     gp.lockOutAuthEnabled = TRUE;
158                     // Record the changes to NV
159                     NV_SYNC_PERSISTENT(lockOutAuthEnabled);
160                 }
161             }
162         }
163         return;
164     }
```

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 = sizeof(gp.EPSeed.t.buffer);
    gp.SPSeed.t.size = sizeof(gp.SPSeed.t.buffer);
    gp.PPSeed.t.size = sizeof(gp.PPSeed.t.buffer);
#if (defined USE_PLATFORM_EPS) && (USE_PLATFORM_EPS != NO)
    _plat__GetEPS(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
#else
    CryptRandomGenerate(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
#endif
    CryptRandomGenerate(gp.SPSeed.t.size, gp.SPSeed.t.buffer);
    CryptRandomGenerate(gp.PPSeed.t.size, gp.PPSeed.t.buffer);

    // Initialize owner, endorsement and lockout authorization
    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 = sizeof(gp.phProof.t.buffer);
    gp.shProof.t.size = sizeof(gp.shProof.t.buffer);
    gp.ehProof.t.size = sizeof(gp.ehProof.t.buffer);
    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
```

```
44      NV_SYNC_PERSISTENT(disableClear);
45      NV_SYNC_PERSISTENT(EPSeed);
46      NV_SYNC_PERSISTENT(SPSeed);
47      NV_SYNC_PERSISTENT(PPSeed);
48      NV_SYNC_PERSISTENT(ownerAuth);
49      NV_SYNC_PERSISTENT(endorsementAuth);
50      NV_SYNC_PERSISTENT(lockoutAuth);
51      NV_SYNC_PERSISTENT(ownerAlg);
52      NV_SYNC_PERSISTENT(ownerPolicy);
53      NV_SYNC_PERSISTENT(endorsementAlg);
54      NV_SYNC_PERSISTENT(endorsementPolicy);
55      NV_SYNC_PERSISTENT(lockoutAlg);
56      NV_SYNC_PERSISTENT(lockoutPolicy);
57      NV_SYNC_PERSISTENT(phProof);
58      NV_SYNC_PERSISTENT(shProof);
59      NV_SYNC_PERSISTENT(ehProof);
60
61      return;
62  }
```

8.3.3.2 HierarchyStartup()

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

```
63  void
64  HierarchyStartup(
65      STARTUP_TYPE     type           // IN: start up type
66      )
67  {
68      // phEnable is SET on any startup
69      g_phEnable = TRUE;
70
71      // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
72      // TPM_RESTART
73      if(type != SU_RESUME)
74      {
75          gc.platformAuth.t.size = 0;
76          gc.platformPolicy.t.size = 0;
77
78          // enable the storage and endorsement hierarchies and the platformNV
79          gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
80      }
81
82      // nullProof and nullSeed are updated at every TPM_RESET
83      if((type != SU_RESTART) && (type != SU_RESUME))
84      {
85          gr.nullProof.t.size = sizeof(gr.nullProof.t.buffer);
86          CryptRandomGenerate(gr.nullProof.t.size, gr.nullProof.t.buffer);
87          gr.nullSeed.t.size = sizeof(gr.nullSeed.t.buffer);
88          CryptRandomGenerate(gr.nullSeed.t.size, gr.nullSeed.t.buffer);
89      }
90
91      return;
92  }
```

8.3.3.3 HierarchyGetProof()

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

```
93  TPM2B_PROOF *
94  HierarchyGetProof(
95      TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy constant
96      )
```



```
97  {
98      TPM2B_PROOF         *proof = NULL;
99
100     switch(hierarchy)
101     {
102         case TPM_RH_PLATFORM:
103             // phProof for TPM_RH_PLATFORM
104             proof = &gp.phProof;
105             break;
106         case TPM_RH_ENDORSEMENT:
107             // ehProof for TPM_RH_ENDORSEMENT
108             proof = &gp.ehProof;
109             break;
110         case TPM_RH_OWNER:
111             // shProof for TPM_RH_OWNER
112             proof = &gp.shProof;
113             break;
114         case TPM_RH_NULL:
115             // nullProof for TPM_RH_NULL
116             proof = &gr.nullProof;
117             break;
118         default:
119             FAIL(FATAL_ERROR_INTERNAL);
120             break;
121     }
122     return proof;
123 }
```

8.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```
124 TPM2B_SEED *
125 HierarchyGetPrimarySeed(
126     TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy
127     )
128 {
129     TPM2B_SEED          *seed = NULL;
130     switch(hierarchy)
131     {
132         case TPM_RH_PLATFORM:
133             seed = &gp.PPSeed;
134             break;
135         case TPM_RH_OWNER:
136             seed = &gp.SPSeed;
137             break;
138         case TPM_RH_ENDORSEMENT:
139             seed = &gp.EPSeed;
140             break;
141         case TPM_RH_NULL:
142             return &gr.nullSeed;
143         default:
144             FAIL(FATAL_ERROR_INTERNAL);
145             break;
146     }
147     return seed;
148 }
```

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(1)	hierarchy is enabled
FALSE(0)	hierarchy is disabled

```
149 BOOL
150 HierarchyIsEnabled(
151     TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy
152     )
153 {
154     BOOL            enabled = FALSE;
155
156     switch(hierarchy)
157     {
158         case TPM_RH_PLATFORM:
159             enabled = g_phEnable;
160             break;
161         case TPM_RH_OWNER:
162             enabled = gc.shEnable;
163             break;
164         case TPM_RH_ENDORSEMENT:
165             enabled = gc.ehEnable;
166             break;
167         case TPM_RH_NULL:
168             enabled = TRUE;
169             break;
170         default:
171             FAIL(FATAL_ERROR_INTERNAL);
172             break;
173     }
174     return enabled;
175 }
```


8.4 NvDynamic.c

8.4.1 Introduction

The NV memory is divided into two area: dynamic space for user defined NV indexes 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

```
#define NV_C
#include "Tpm.h"
#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.

```
static NV_REF
NvNext(
    NV_REF          *iter,          // IN/OUT: the list iterator
    TPM_HANDLE      *handle         // OUT: the handle of the next item.
    )
{
    NV_REF              currentAddr;
    NV_ENTRY_HEADER     header;
```

```
12  //
13      // If iterator is at the beginning of list
14      if(*iter == NV_REF_INIT)
15      {
16          // Initialize iterator
17          *iter = NV_USER_DYNAMIC;
18      }
19      // Step over the size field and point to the handle
20      currentAddr = *iter + sizeof(UINT32);
21
22      // read the header of the next entry
23      NvRead(&header, *iter, sizeof(NV_ENTRY_HEADER));
24
25      // if the size field is zero, then we have hit the end of the list
26      if(header.size == 0)
27          // leave the *iter pointing at the end of the list
28          return 0;
29      // advance the header by the size of the entry
30      *iter += header.size;
31
32      if(handle != NULL)
33          *handle = header.handle;
34      return currentAddr;
35  }
```

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

```
36  static NV_REF
37  NvNextByType(
38      TPM_HANDLE      *handle,        // OUT: the handle of the found type
39      NV_REF          *iter,          // IN: the iterator
40      TPM_HT           type           // IN: the handle type to look for
41      )
42  {
43      NV_REF           addr;
44      TPM_HANDLE       nvHandle;
45  //
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 reference

```
#define NvNextIndex(handle, iter)                    \
    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.

```
#define NvNextEvict(handle, iter)                    \
    NvNextByType(handle, iter, TPM_HT_PERSISTENT)
```

8.4.3.5 NvGetEnd()

Function to find the end of the NV dynamic data list

```
static NV_REF
NvGetEnd(
    void
    )
{
    NV_REF          iter = NV_REF_INIT;
    NV_REF          currentAddr;
//
    // Scan until the next address is 0
    while((currentAddr = NvNext(&iter, NULL)) != 0);
    return iter;
}
```

8.4.3.6 NvGetFreeBytes

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

```
static UINT32
NvGetFreeBytes(
    void
    )
{
    // This does not have an overflow issue because NvGetEnd() cannot return a value
    // that is larger than s_evictNvEnd. This is because there is always a 'stop'
    // word in the NV memory that terminates the search for the end before the
    // value can go past s_evictNvEnd.
    return s_evictNvEnd - NvGetEnd();
}
```

8.4.3.7 NvTestSpace()

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

Return Value	Meaning
TRUE(1)	space available
FALSE(0)	no enough space

```
84  static BOOL
85  NvTestSpace(
86      UINT32          size,           // IN: size of the entity to be added
87      BOOL            isIndex,        // IN: TRUE if the entity is an index
88      BOOL            isCounter       // IN: TRUE if the index is a counter
89      )
90  {
91      UINT32      remainBytes = NvGetFreeBytes();
92      UINT32      reserved = sizeof(UINT32)       // size of the forward pointer
93          + sizeof(NV_LIST_TERMINATOR);
94  //
95      // Do a compile time sanity check on the setting for NV_MEMORY_SIZE
96  #if NV_MEMORY_SIZE < 1024
97  #error "NV_MEMORY_SIZE probably isn't large enough"
98  #endif
99
100     // For NV Index, need to make sure that we do not allocate an Index if this
101     // would mean that the TPM cannot allocate the minimum number of evict
102     // objects.
103     if(isIndex)
104     {
105         // Get the number of persistent objects allocated
106         UINT32      persistentNum = NvCapGetPersistentNumber();
107
108         // If we have not allocated the requisite number of evict objects, then we
109         // need to reserve space for them.
110         // NOTE: some of this is not written as simply as it might seem because
111         // the values are all unsigned and subtracting needs to be done carefully
112         // so that an underflow doesn't cause problems.
113         if(persistentNum < MIN_EVICT_OBJECTS)
114             reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
115     }
116     // If this is not an index or is not a counter, reserve space for the
117     // required number of counter indexes
118     if(!isIndex || !isCounter)
119     {
120         // Get the number of counters
121         UINT32      counterNum = NvCapGetCounterNumber();
122
123         // If the required number of counters have not been allocated, reserved
124         // space for the extra needed counters
125         if(counterNum < MIN_COUNTER_INDICES)
126             reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
127     }
128     // Check that the requested allocation will fit after making sure that there
129     // will be no chance of overflow
130     return ((reserved < remainBytes)
131             && (size <= remainBytes)
132             && (size + reserved <= remainBytes));
133 }
```

8.4.3.8 NvWriteNvListEnd()

Function to write the list terminator.

```
134 NV_REF
135 NvWriteNvListEnd(
136     NV_REF          end
```



```
137     )
138 {
139     // Marker is initialized with zeros
140     BYTE        listEndMarker[sizeof(NV_LIST_TERMINATOR)] = {0};
141     UINT64      maxCount = NvReadMaxCount();
142 //
143     // This is a constant check that can be resolved at compile time.
144     cAssert(sizeof(UINT64) <= sizeof(NV_LIST_TERMINATOR) - sizeof(UINT32));
145
146     // Copy the maxCount value to the marker buffer
147     MemoryCopy(&listEndMarker[sizeof(UINT32)], &maxCount, sizeof(UINT64));
148     pAssert(end + sizeof(NV_LIST_TERMINATOR) <= s_evictNvEnd);
149
150     // Write it to memory
151     NvWrite(end, sizeof(NV_LIST_TERMINATOR), &listEndMarker);
152     return end + sizeof(NV_LIST_TERMINATOR);
153 }
```

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.

```
154 static TPM_RC
155 NvAdd(
156     UINT32          totalSize,      // IN: total size needed for this entity For
157                                     //     evict object, totalSize is the same as
158                                     //     bufferSize.  For NV Index, totalSize is
159                                     //     bufferSize plus index data size
160     UINT32          bufferSize,     // IN: size of initial buffer
161     TPM_HANDLE      handle,         // IN: optional handle
162     BYTE            *entity         // IN: initial buffer
163     )
164 {
165     NV_REF          newAddr;        // IN: where the new entity will start
166     NV_REF          nextAddr;
167 //
168     RETURN_IF_NV_IS_NOT_AVAILABLE;
169
170     // Get the end of data list
171     newAddr = NvGetEnd();
172
173     // Step over the forward pointer
174     nextAddr = newAddr + sizeof(UINT32);
175
176     // Optionally write the handle. For indexes, the handle is TPM_RH_UNASSIGNED
177     // so that the handle in the nvIndex is used instead of writing this value
178     if(handle != TPM_RH_UNASSIGNED)
179     {
180         NvWrite((UINT32)nextAddr, sizeof(TPM_HANDLE), &handle);
181         nextAddr += sizeof(TPM_HANDLE);
182     }
183     // Write entity data
184     NvWrite((UINT32)nextAddr, bufferSize, entity);
185
186     // Advance the pointer by the amount of the total
187     nextAddr += totalSize;
188
189     // Finish by writing the link value
190
191     // Write the next offset (relative addressing)
```

```
192         totalSize = nextAddr - newAddr;
193
194         // Write link value
195         NvWrite((UINT32)newAddr, sizeof(UINT32), &totalSize);
196
197         // Write the list terminator
198         NvWriteNvListEnd(nextAddr);
199
200         return TPM_RC_SUCCESS;
201     }
```

8.4.3.10 NvDelete()

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

```
202     static TPM_RC
203     NvDelete(
204         NV_REF           entityRef      // IN: reference to entity to be deleted
205         )
206     {
207         UINT32          entrySize;
208         // adjust entityAddr to back up and point to the forward pointer
209         NV_REF          entryRef = entityRef - sizeof(UINT32);
210         NV_REF          endRef = NvGetEnd();
211         NV_REF          nextAddr; // address of the next entry
212     //
213         RETURN_IF_NV_IS_NOT_AVAILABLE;
214
215         // Get the offset of the next entry. That is, back up and point to the size
216         // field of the entry
217         NvRead(&entrySize, entryRef, sizeof(UINT32));
218
219         // The next entry after the one being deleted is at a relative offset
220         // from the current entry
221         nextAddr = entryRef + entrySize;
222
223         // If this is not the last entry, move everything up
224         if(nextAddr < endRef)
225         {
226             pAssert(nextAddr > entryRef);
227             _plat__NvMemoryMove(nextAddr,
228                                 entryRef,
229                                 (endRef - nextAddr));
230         }
231         // The end of the used space is now moved up by the amount of space we just
232         // reclaimed
233         endRef -= entrySize;
234
235         // Write the end marker, and make the new end equal to the first byte after
236         // the just added end value. This will automatically update the NV value for
237         // maxCounter.
238         // NOTE: This is the call that sets flag to cause NV to be updated
239         endRef = NvWriteNvListEnd(endRef);
240
241         // Clear the reclaimed memory
242         _plat__NvMemoryClear(endRef, entrySize);
243
244         return TPM_RC_SUCCESS;
245     }
```


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())

8.4.4.2 NvRamNext()

This function is used to iterate trough the list of Ram Index values. *iter needs to be initialized by calling

```
246 static NV_RAM_REF
247 NvRamNext(
248     NV_RAM_REF      *iter,          // IN/OUT: the list iterator
249     TPM_HANDLE      *handle         // OUT: the handle of the next item.
250     )
251 {
252     NV_RAM_REF          currentAddr;
253     NV_RAM_HEADER       header;
254 //
255     // If iterator is at the beginning of list
256     if(*iter == NV_RAM_REF_INIT)
257     {
258         // Initialize iterator
259         *iter = &s_indexOrderlyRam[0];
260     }
261     // if we are going to return what the iter is currently pointing to...
262     currentAddr = *iter;
263
264     // If iterator reaches the end of NV space, then don't advance and return
265     // that we are at the end of the list. The end of the list occurs when
266     // we don't have space for a size and a handle
267     if(currentAddr + sizeof(NV_RAM_HEADER) > RAM_ORDERLY_END)
268         return NULL;
269     // read the header of the next entry
270     MemoryCopy(&header, currentAddr, sizeof(NV_RAM_HEADER));
271
272     // if the size field is zero, then we have hit the end of the list
273     if(header.size == 0)
274         // leave the *iter pointing at the end of the list
275         return NULL;
276     // advance the header by the size of the entry
277     *iter = currentAddr + header.size;
278
279 //    pAssert(*iter <= RAM_ORDERLY_END);
280     if(handle != NULL)
281         *handle = header.handle;
282     return currentAddr;
283 }
```

8.4.4.3 NvRamGetEnd()

This routine performs the same function as NvGetEnd() but for the RAM data.

```
284 static NV_RAM_REF
285 NvRamGetEnd(
286     void
287     )
```

```
288 {
289     NV_RAM_REF          iter = NV_RAM_REF_INIT;
290     NV_RAM_REF          currentAddr;
291 //
292     // Scan until the next address is 0
293     while((currentAddr = NvRamNext(&iter, NULL)) != 0);
294     return iter;
295 }
```

8.4.4.4 NvRamTestSpaceIndex()

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

Return Value	Meaning
TRUE(1)	space available
FALSE(0)	no enough space

```
296 static BOOL
297 NvRamTestSpaceIndex(
298     UINT32          size            // IN: size of the data to be added to RAM
299     )
300 {
301     UINT32          remaining = RAM_ORDERLY_END - NvRamGetEnd();
302     UINT32          needed = sizeof(NV_RAM_HEADER) + size;
303 //
304     // NvRamGetEnd points to the next available byte.
305     return remaining >= needed;
306 }
```

8.4.4.5 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.

```
307 static NV_RAM_REF
308 NvRamGetIndex(
309     TPMI_RH_NV_INDEX    handle          // IN: NV handle
310     )
311 {
312     NV_RAM_REF          iter = NV_RAM_REF_INIT;
313     NV_RAM_REF          currentAddr;
314     TPM_HANDLE          foundHandle;
315 //
316     while((currentAddr = NvRamNext(&iter, &foundHandle)) != 0)
317     {
318         if(handle == foundHandle)
319             break;
320     }
321     return currentAddr;
322 }
```

8.4.4.6 NvUpdateIndexOrderlyData()

This function is used to cause an update of the orderly data to the NV backing store.

```
323 void
324 NvUpdateIndexOrderlyData(
325     void
326     )
```



```
{
    // Write reserved RAM space to NV
    NvWrite(NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
}
```

8.4.4.7 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.

```
static void
NvAddRAM(
    TPMS_NV_PUBLIC  *index          // IN: the index descriptor
    )
{
    NV_RAM_HEADER       header;
    NV_RAM_REF          end = NvRamGetEnd();
//
    header.size = sizeof(NV_RAM_HEADER) + index->dataSize;
    header.handle = index->nvIndex;
    MemoryCopy(&header.attributes, &index->attributes, sizeof(TPMA_NV));

    pAssert(ORDERLY_RAM_ADDRESS_OK(end, header.size));

    // Copy the header to the memory
    MemoryCopy(end, &header, sizeof(NV_RAM_HEADER));

    // Clear the data area (just in case)
    MemorySet(end + sizeof(NV_RAM_HEADER), 0, index->dataSize);

    // Step over this new entry
    end += header.size;

    // If the end marker will fit, add it
    if(end + sizeof(UINT32) < RAM_ORDERLY_END)
        MemorySet(end, 0, sizeof(UINT32));
    // Write reserved RAM space to NV to reflect the newly added NV Index
    SET_NV_UPDATE(UT_ORDERLY);

    return;
}
```

8.4.4.8 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.

```
static void
NvDeleteRAM(
    TPMI_RH_NV_INDEX     handle         // IN: NV handle
    )
{
    NV_RAM_REF           nodeAddress;
    NV_RAM_REF           nextNode;
```

```
    UINT32                  size;
    NV_RAM_REF              lastUsed = NvRamGetEnd();
//
    nodeAddress = NvRamGetIndex(handle);

    pAssert(nodeAddress != 0);

    // Get node size
    MemoryCopy(&size, nodeAddress, sizeof(size));

    // Get the offset of next node
    nextNode = nodeAddress + size;

    // Copy the data
    MemoryCopy(nodeAddress, nextNode, lastUsed - nextNode);

    // Clear out the reclaimed space
    MemorySet(lastUsed - size, 0, size);

    // Write reserved RAM space to NV to reflect the newly delete NV Index
    SET_NV_UPDATE(UT_ORDERLY);

    return;
}
```

8.4.4.9 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.

```
void
NvReadNvIndexInfo(
    NV_REF           ref,           // IN: points to NV where index is located
    NV_INDEX        *nvIndex        // OUT: place to receive index data
    )
{
    pAssert(nvIndex != NULL);
    NvRead(nvIndex, ref, sizeof(NV_INDEX));
    return;
}
```

8.4.4.10 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.

```
void
NvReadObject(
    NV_REF           ref,           // IN: points to NV where index is located
    OBJECT          *object         // OUT: place to receive the object data
    )
{
    NvRead(object, (ref + sizeof(TPM_HANDLE)), sizeof(OBJECT));
    return;
}
```

8.4.4.11 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

```
412 static NV_REF
413 NvFindEvict(
414     TPM_HANDLE        nvHandle,
415     OBJECT           *object
416     )
417 {
418     NV_REF           found = NvFindHandle(nvHandle);
419 //
420     // If we found the handle and the request included an object pointer, fill it in
421     if(found != 0 && object != NULL)
422         NvReadObject(found, object);
423     return found;
424 }
```

8.4.4.12 NvIndexIsDefined()

See if an index is already defined

```
425 BOOL
426 NvIndexIsDefined(
427     TPM_HANDLE        nvHandle        // IN: Index to look for
428     )
429 {
430     return (NvFindHandle(nvHandle) != 0);
431 }
```

8.4.4.13 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

```
432 static TPM_RC
433 NvConditionallyWrite(
434     NV_REF           entryAddr,      // IN: stating address
435     UINT32           size,           // IN: size of the data to write
436     void            *data            // IN: the data to write
437     )
438 {
439     // If the index data is actually changed, then a write to NV is required
440     if(_plat__NvIsDifferent(entryAddr, size, data))
441     {
442         // Write the data if NV is available
443         if(g_NvStatus == TPM_RC_SUCCESS)
444         {
445             NvWrite(entryAddr, size, data);
446         }
447         return g_NvStatus;
448     }
449     return TPM_RC_SUCCESS;
450 }
```


8.4.4.14 NvReadNvIndexAttributes()

This function returns the attributes of an NV Index.

```
451 static TPMA_NV
452 NvReadNvIndexAttributes(
453     NV_REF           locator        // IN: reference to an NV index
454     )
455 {
456     TPMA_NV                 attributes;
457 //
458     NvRead(&attributes,
459            locator + offsetof(NV_INDEX, publicArea.attributes),
460            sizeof(TPMA_NV));
461     return attributes;
462 }
```

8.4.4.15 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.

```
463 static TPMA_NV
464 NvReadRamIndexAttributes(
465     NV_RAM_REF       ref            // IN: pointer to a NV_RAM_HEADER
466     )
467 {
468     TPMA_NV         attributes;
469 //
470     MemoryCopy(&attributes, ref + offsetof(NV_RAM_HEADER, attributes),
471                sizeof(TPMA_NV));
472     return attributes;
473 }
```

8.4.4.16 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

```
474 static TPM_RC
475 NvWriteNvIndexAttributes(
476     NV_REF           locator,       // IN: location of the index
477     TPMA_NV          attributes     // IN: attributes to write
478     )
479 {
480     return NvConditionallyWrite(
481         locator + offsetof(NV_INDEX, publicArea.attributes),
482         sizeof(TPMA_NV),
483         &attributes);
484 }
```

8.4.4.17 NvWriteRamIndexAttributes()

This function is used to write the index attributes into an unaligned structure


```
485 static void
486 NvWriteRamIndexAttributes(
487     NV_RAM_REF        ref,            // IN: address of the header
488     TPMA_NV           attributes      // IN: the attributes to write
489     )
490 {
491     MemoryCopy(ref + offsetof(NV_RAM_HEADER, attributes), &attributes,
492                sizeof(TPMA_NV));
493     return;
494 }
```

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(1)	handle references a platform persistent object and may reference an owner persistent object either
FALSE(0)	handle does not reference platform persistent object

```
495 BOOL
496 NvIsPlatformPersistentHandle(
497     TPM_HANDLE        handle          // IN: handle
498     )
499 {
500     return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
501 }
```

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(1)	handle is owner persistent handle
FALSE(0)	handle is not owner persistent handle and may not be a persistent handle at all

```
502 BOOL
503 NvIsOwnerPersistentHandle(
504     TPM_HANDLE        handle          // IN: handle
505     )
506 {
507     return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
508 }
```

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

```
TPM_RC
NvIndexIsAccessible(
    TPMI_RH_NV_INDEX     handle        // IN: handle
    )
{
    NV_INDEX            *nvIndex = NvGetIndexInfo(handle, NULL);
//
    if(nvIndex == NULL)
    // If index is not found, return TPM_RC_HANDLE
        return TPM_RC_HANDLE;
    if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
    {
        // if shEnable is CLEAR, an ownerCreate NV Index should not be
        // indicated as present
        if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
        {
            if(gc.shEnable == FALSE)
                return TPM_RC_HANDLE;
        }
        // if phEnableNV is CLEAR, a platform created Index should not
        // be visible
        else if(gc.phEnableNV == FALSE)
            return TPM_RC_HANDLE;
    }
#if 0 // Writelock test for debug
    // If the Index is write locked and this is an NV Write operation...
    if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITELOCKED)
       &&  IsWriteOperation(commandIndex))
    {
        // then return a locked indication unless the command is TPM2_NV_WriteLock
        if(GetCommandCode(commandIndex) != TPM_CC_NV_WriteLock)
            return TPM_RC_NV_LOCKED;
        return TPM_RC_SUCCESS;
    }
#endif
#if 0   // Readlock Test for debug
    // If the Index is read locked and this is an NV Read operation...
    if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, READLOCKED)
       && IsReadOperation(commandIndex))
    {
        // then return a locked indication unless the command is TPM2_NV_ReadLock
        if(GetCommandCode(commandIndex) != TPM_CC_NV_ReadLock)
            return TPM_RC_NV_LOCKED;
    }
#endif
    // NV Index is accessible
    return TPM_RC_SUCCESS;
}
```

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

```
TPM_RC
NvGetEvictObject(
    TPM_HANDLE       handle,         // IN: handle
    OBJECT          *object          // OUT: object data
    )
{
    NV_REF          entityAddr;         // offset points to the entity
//
    // Find the address of evict object and copy to object
    entityAddr = NvFindEvict(handle, object);

    // whether there is an error or not, make sure that the evict
    // status of the object is set so that the slot will get freed on exit
    // Must do this after NvFindEvict loads the object
    object->attributes.evict = SET;

    // If handle is not found, return an error
    if(entityAddr == 0)
        return TPM_RC_HANDLE;
    return TPM_RC_SUCCESS;
}
```

8.4.5.5 NvIndexCacheInit()

Function to initialize the Index cache

```
void
NvIndexCacheInit(
    void
    )
{
    s_cachedNvRef = NV_REF_INIT;
    s_cachedNvRamRef = NV_RAM_REF_INIT;
    s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
    return;
}
```

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.

```
void
NvGetIndexData(
    NV_INDEX        *nvIndex,       // IN: the in RAM index descriptor
    NV_REF           locator,       // IN: where the data is located
    UINT32           offset,        // IN: offset of NV data
    UINT16           size,          // IN: size of NV data
    void            *data           // OUT: data buffer
    )
{
    TPMA_NV             nvAttributes;
//
    pAssert(nvIndex != NULL);

    nvAttributes = nvIndex->publicArea.attributes;
```

```
    pAssert(IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN));

    if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, ORDERLY))
    {
        // Get data from RAM buffer
        NV_RAM_REF           ramAddr = NvRamGetIndex(nvIndex->publicArea.nvIndex);
        pAssert(ramAddr != 0 && (size <=
                ((NV_RAM_HEADER *)ramAddr)->size - sizeof(NV_RAM_HEADER) - offset));
        MemoryCopy(data, ramAddr + sizeof(NV_RAM_HEADER) + offset, size);
    }
    else
    {
        // Validate that read falls within range of the index
        pAssert(offset <= nvIndex->publicArea.dataSize
                &&  size <= (nvIndex->publicArea.dataSize - offset));
        NvRead(data, locator + sizeof(NV_INDEX) + offset, size);
    }
    return;
}
```

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.

```
UINT64
NvGetUINT64Data(
    NV_INDEX        *nvIndex,       // IN: the in RAM index descriptor
    NV_REF           locator        // IN: where index exists in NV
    )
{
    UINT64              intVal;
//
    // Read the value and convert it to internal format
    NvGetIndexData(nvIndex, locator, 0, 8, &intVal);
    return BYTE_ARRAY_TO_UINT64(((BYTE *)&intVal));
}
```

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

```
TPM_RC
NvWriteIndexAttributes(
    TPM_HANDLE       handle,
    NV_REF           locator,       // IN: location of the index
    TPMA_NV          attributes     // IN: attributes to write
    )
{
    TPM_RC              result;
//
    if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
    {
        NV_RAM_REF      ram = NvRamGetIndex(handle);
        NvWriteRamIndexAttributes(ram, attributes);
```



```
647             result = TPM_RC_SUCCESS;
648         }
649         else
650         {
651             result = NvWriteNvIndexAttributes(locator, attributes);
652         }
653         return result;
654     }
```

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

```
655     TPM_RC
656     NvWriteIndexAuth(
657         NV_REF           locator,       // IN: location of the index
658         TPM2B_AUTH      *authValue      // IN: the authValue to write
659         )
660     {
661         TPM_RC              result;
662     //
663         // If the locator is pointing to the cached index value...
664         if(locator == s_cachedNvRef)
665         {
666             // copy the authValue to the cached index so it will be there if we
667             // look for it. This is a safety thing.
668             MemoryCopy2B(&s_cachedNvIndex.authValue.b, &authValue->b,
669                          sizeof(s_cachedNvIndex.authValue.t.buffer));
670         }
671         result = NvConditionallyWrite(
672             locator + offsetof(NV_INDEX, authValue),
673             sizeof(UINT16) + authValue->t.size,
674             authValue);
675         return result;
676     }
```

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

```
677     NV_INDEX *
678     NvGetIndexInfo(
679         TPM_HANDLE       nvHandle,      // IN: the index handle
680         NV_REF          *locator        // OUT: location of the index
681         )
682     {
683         if(s_cachedNvIndex.publicArea.nvIndex != nvHandle)
684         {
685             s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
686             s_cachedNvRamRef = 0;
687             s_cachedNvRef = NvFindHandle(nvHandle);
688             if(s_cachedNvRef == 0)
```

```
689                 return NULL;
690             NvReadNvIndexInfo(s_cachedNvRef, &s_cachedNvIndex);
691             if(IS_ATTRIBUTE(s_cachedNvIndex.publicArea.attributes, TPMA_NV, ORDERLY))
692             {
693                 s_cachedNvRamRef = NvRamGetIndex(nvHandle);
694                 s_cachedNvIndex.publicArea.attributes =
695                     NvReadRamIndexAttributes(s_cachedNvRamRef);
696             }
697         }
698         if(locator != NULL)
699             *locator = s_cachedNvRef;
700         return &s_cachedNvIndex;
701     }
```

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 indexes 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

```
702 TPM_RC
703 NvWriteIndexData(
704     NV_INDEX        *nvIndex,       // IN: the description of the index
705     UINT32           offset,        // IN: offset of NV data
706     UINT32           size,          // IN: size of NV data
707     void            *data           // IN: data buffer
708     )
709 {
710     TPM_RC              result = TPM_RC_SUCCESS;
711 //
712     pAssert(nvIndex != NULL);
713     // Make sure that this is dealing with the 'default' index.
714     // Note: it is tempting to change the calling sequence so that the 'default' is
715     // presumed.
716     pAssert(nvIndex->publicArea.nvIndex == s_cachedNvIndex.publicArea.nvIndex);
717
718     // Validate that write falls within range of the index
719     pAssert(offset <= nvIndex->publicArea.dataSize
720             &&  size <= (nvIndex->publicArea.dataSize - offset));
721
722     // Update TPMA_NV_WRITTEN bit if necessary
723     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
724     {
725         // Update the in memory version of the attributes
726         SET_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN);
727
728         // If this is not orderly, then update the NV version of
729         // the attributes
730         if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
731         {
732             result = NvWriteNvIndexAttributes(s_cachedNvRef,
```



```
733                                                 nvIndex->publicArea.attributes);
734             if(result != TPM_RC_SUCCESS)
735                 return result;
736             // If this is a partial write of an ordinary index, clear the whole
737             // index.
738             if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes)
739                && (nvIndex->publicArea.dataSize > size))
740                 _plat__NvMemoryClear(s_cachedNvRef + sizeof(NV_INDEX),
741                                      nvIndex->publicArea.dataSize);
742         }
743         else
744         {
745             // This is orderly so update the RAM version
746             MemoryCopy(s_cachedNvRamRef + offsetof(NV_RAM_HEADER, attributes),
747                        &nvIndex->publicArea.attributes, sizeof(TPMA_NV));
748             // If setting WRITTEN for an orderly counter, make sure that the
749             // state saved version of the counter is saved
750             if(IsNvCounterIndex(nvIndex->publicArea.attributes))
751                 SET_NV_UPDATE(UT_ORDERLY);
752             // If setting the written attribute on an ordinary index, make sure that
753             // the data is all cleared out in case there is a partial write. This
754             // is only necessary for ordinary indexes because all of the other types
755             // are always written in total.
756             else if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes))
757                 MemorySet(s_cachedNvRamRef + sizeof(NV_RAM_HEADER),
758                           0, nvIndex->publicArea.dataSize);
759         }
760     }
761     // If this is orderly data, write it to RAM
762     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
763     {
764         // Note: if this is the first write to a counter, the code above will queue
765         // the write to NV of the RAM data in order to update TPMA_NV_WRITTEN. In
766         // process of doing that write, it will also write the initial counter value
767
768         // Update RAM
769         MemoryCopy(s_cachedNvRamRef + sizeof(NV_RAM_HEADER) + offset, data, size);
770
771         // And indicate that the TPM is no longer orderly
772         g_clearOrderly = TRUE;
773     }
774     else
775     {
776         // Offset into the index to the first byte of the data to be written to NV
777         result = NvConditionallyWrite(s_cachedNvRef + sizeof(NV_INDEX) + offset,
778                                       size, data);
779     }
780     return result;
781 }
```

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)

```
782 TPM_RC
783 NvWriteUINT64Data(
784     NV_INDEX        *nvIndex,       // IN: the description of the index
785     UINT64           intValue       // IN: the value to write
786     )
787 {
```

```
    BYTE            bytes[8];
    UINT64_TO_BYTE_ARRAY(intValue, bytes);
//
    return NvWriteIndexData(nvIndex, 0, 8, &bytes);
}
```

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.

```
TPM2B_NAME *
NvGetIndexName(
    NV_INDEX        *nvIndex,       // IN: the index over which the name is to be
                                    //     computed
    TPM2B_NAME      *name           // OUT: name of the index
    )
{
    UINT16              dataSize, digestSize;
    BYTE                marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
    BYTE               *buffer;
    HASH_STATE          hashState;
//
    // Marshal public area
    buffer = marshalBuffer;
    dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex->publicArea, &buffer, NULL);

    // hash public area
    digestSize = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
    CryptDigestUpdate(&hashState, dataSize, marshalBuffer);

    // Complete digest leaving room for the nameAlg
    CryptHashEnd(&hashState, digestSize, &name->b.buffer[2]);

    // Include the nameAlg
    UINT16_TO_BYTE_ARRAY(nvIndex->publicArea.nameAlg, name->b.buffer);
    name->t.size = digestSize + 2;
    return name;
}
```

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.

```
TPM2B_NAME *
NvGetNameByIndexHandle(
    TPMI_RH_NV_INDEX     handle,        // IN: handle of the index
    TPM2B_NAME          *name           // OUT: name of the index
    )
{
    NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
//
    return NvGetIndexName(nvIndex, name);
}
```


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

```
831  TPM_RC
832  NvDefineIndex(
833      TPMS_NV_PUBLIC  *publicArea,    // IN: A template for an area to create.
834      TPM2B_AUTH      *authValue      // IN: The initial authorization value
835      )
836  {
837      // The buffer to be written to NV memory
838      NV_INDEX        nvIndex;            // the index data
839      UINT16          entrySize;          // size of entry
840      TPM_RC          result;
841  //
842      entrySize = sizeof(NV_INDEX);
843
844      // only allocate data space for indexes that are going to be written to NV.
845      // Orderly indexes don't need space.
846      if(!IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
847          entrySize += publicArea->dataSize;
848      // Check if we have enough space to create the NV Index
849      // In this implementation, the only resource limitation is the available NV
850      // space (and possibly RAM space.)  Other implementation may have other
851      // limitation on counter or on NV slots
852      if(!NvTestSpace(entrySize, TRUE, IsNvCounterIndex(publicArea->attributes)))
853          return TPM_RC_NV_SPACE;
854
855      // if the index to be defined is RAM backed, check RAM space availability
856      // as well
857      if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY)
858         &&  !NvRamTestSpaceIndex(publicArea->dataSize))
859          return TPM_RC_NV_SPACE;
860      // Copy input value to nvBuffer
861      nvIndex.publicArea = *publicArea;
862
863      // Copy the authValue
864      nvIndex.authValue = *authValue;
865
866      // Add index to NV memory
867      result = NvAdd(entrySize, sizeof(NV_INDEX), TPM_RH_UNASSIGNED,
868                     (BYTE *)&nvIndex);
869      if(result == TPM_RC_SUCCESS)
870      {
871      // If the data of NV Index is RAM backed, add the data area in RAM as well
872          if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
873              NvAddRAM(publicArea);
874      }
875      return result;
876  }
```

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

```
877  TPM_RC
878  NvAddEvictObject(
879      TPMI_DH_OBJECT   evictHandle,   // IN: new evict handle
880      OBJECT          *object         // IN: object to be added
881      )
882  {
883      TPM_HANDLE       temp = object->evictHandle;
884      TPM_RC           result;
885  //
886      // Check if we have enough space to add the evict object
887      // An evict object needs 8 bytes in index table + sizeof OBJECT
888      // In this implementation, the only resource limitation is the available NV
889      // space.  Other implementation may have other limitation on evict object
890      // handle space
891      if(!NvTestSpace(sizeof(OBJECT) + sizeof(TPM_HANDLE), FALSE, FALSE))
892          return TPM_RC_NV_SPACE;
893
894      // Set evict attribute and handle
895      object->attributes.evict = SET;
896      object->evictHandle = evictHandle;
897
898      // Now put this in NV
899      result = NvAdd(sizeof(OBJECT), sizeof(OBJECT), evictHandle, (BYTE *)object);
900
901      // Put things back the way they were
902      object->attributes.evict = CLEAR;
903      object->evictHandle = temp;
904
905      return result;
906  }
```

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

```
907  TPM_RC
908  NvDeleteIndex(
909      NV_INDEX        *nvIndex,       // IN: an in RAM index descriptor
910      NV_REF           entityAddr     // IN: location in NV
911      )
912  {
913      TPM_RC           result;
914  //
915      if(nvIndex != NULL)
916      {
917          // Whenever a counter is deleted, make sure that the MaxCounter value is
918          // updated to reflect the value
919          if(IsNvCounterIndex(nvIndex->publicArea.attributes)
920             && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
921              NvUpdateMaxCount(NvGetUINT64Data(nvIndex, entityAddr));
922          result = NvDelete(entityAddr);
923          if(result != TPM_RC_SUCCESS)
```



```
924                 return result;
925             // If the NV Index is RAM backed, delete the RAM data as well
926             if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
927                 NvDeleteRAM(nvIndex->publicArea.nvIndex);
928             NvIndexCacheInit();
929         }
930         return TPM_RC_SUCCESS;
931     }
```

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

```
932     TPM_RC
933     NvDeleteEvict(
934         TPM_HANDLE        handle          // IN: handle of entity to be deleted
935         )
936     {
937         NV_REF       entityAddr = NvFindEvict(handle, NULL);     // pointer to entity
938         TPM_RC       result = TPM_RC_SUCCESS;
939     //
940         if(entityAddr != 0)
941             result = NvDelete(entityAddr);
942         return result;
943     }
```

8.4.5.19 NvFlushHierarchy()

This function will delete persistent objects belonging to the indicated hierarchy. 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

```
944     TPM_RC
945     NvFlushHierarchy(
946         TPMI_RH_HIERARCHY    hierarchy       // IN: hierarchy to be flushed.
947         )
948     {
949         NV_REF           iter = NV_REF_INIT;
950         NV_REF           currentAddr;
951         TPM_HANDLE       entityHandle;
952         TPM_RC           result = TPM_RC_SUCCESS;
953     //
954         while((currentAddr = NvNext(&iter, &entityHandle)) != 0)
955         {
956             if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
957             {
958                 NV_INDEX        nvIndex;
959     //
960                 // If flush endorsement or platform hierarchy, no NV Index would be
961                 // flushed
962                 if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
963                     continue;
964                 // Get the index information
965                 NvReadNvIndexInfo(currentAddr, &nvIndex);
966
967                 // For storage hierarchy, flush OwnerCreated index
968                 if(!IS_ATTRIBUTE(nvIndex.publicArea.attributes, TPMA_NV,
969                                  PLATFORMCREATE))
```

```
970                 {
971                     // Delete the index (including RAM for orderly)
972                     result = NvDeleteIndex(&nvIndex, currentAddr);
973                     if(result != TPM_RC_SUCCESS)
974                         break;
975                     // Re-iterate from beginning after a delete
976                     iter = NV_REF_INIT;
977                 }
978             }
979             else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
980             {
981                 OBJECT_ATTRIBUTES           attributes;
982     //
983                 NvRead(&attributes,
984                        (UINT32)(currentAddr
985                                 + sizeof(TPM_HANDLE)
986                                 + offsetof(OBJECT, attributes)),
987                        sizeof(OBJECT_ATTRIBUTES));
988                 // If the evict object belongs to the hierarchy to be flushed...
989                 if((hierarchy == TPM_RH_PLATFORM && attributes.ppsHierarchy == SET)
990                    || (hierarchy == TPM_RH_OWNER && attributes.spsHierarchy == SET)
991                    || (hierarchy == TPM_RH_ENDORSEMENT
992                        &&  attributes.epsHierarchy == SET))
993                 {
994                     // ...then delete the evict object
995                     result = NvDelete(currentAddr);
996                     if(result != TPM_RC_SUCCESS)
997                         break;
998                     // Re-iterate from beginning after a delete
999                     iter = NV_REF_INIT;
1000                }
1001            }
1002            else
1003            {
1004                FAIL(FATAL_ERROR_INTERNAL);
1005            }
1006        }
1007        return result;
1008    }
```

8.4.5.20 NvSetGlobalLock()

This function is used to SET the TPMA_NV_WRITELOCKED attribute for all NV indexes 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

```
1009    TPM_RC
1010    NvSetGlobalLock(
1011        void
1012        )
1013    {
1014        NV_REF          iter = NV_REF_INIT;
1015        NV_RAM_REF      ramIter = NV_RAM_REF_INIT;
1016        NV_REF          currentAddr;
1017        NV_RAM_REF      currentRamAddr;
1018        TPM_RC          result = TPM_RC_SUCCESS;
1019    //
1020        // Check all normal indexes
1021        while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
```



```
1022        {
1023            TPMA_NV         attributes = NvReadNvIndexAttributes(currentAddr);
1024    //
1025            // See if it should be locked
1026            if(!IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1027               &&  IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1028            {
1029                SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1030                result = NvWriteNvIndexAttributes(currentAddr, attributes);
1031                if(result != TPM_RC_SUCCESS)
1032                    return result;
1033            }
1034        }
1035        // Now search all the orderly attributes
1036        while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1037        {
1038            // See if it should be locked
1039            TPMA_NV         attributes = NvReadRamIndexAttributes(currentRamAddr);
1040            if(IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1041            {
1042                SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1043                NvWriteRamIndexAttributes(currentRamAddr, attributes);
1044            }
1045        }
1046        return result;
1047    }
```

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

```
1048    static void
1049    InsertSort(
1050        TPML_HANDLE     *handleList,    // IN/OUT: sorted handle list
1051        UINT32           count,         // IN: maximum count in the handle list
1052        TPM_HANDLE       entityHandle   // IN: handle to be inserted
1053        )
1054    {
1055        UINT32          i, j;
1056        UINT32          originalCount;
1057    //
1058        // For a corner case that the maximum count is 0, do nothing
1059        if(count == 0)
1060            return;
1061        // For empty list, add the handle at the beginning and return
1062        if(handleList->count == 0)
1063        {
1064            handleList->handle[0] = entityHandle;
1065            handleList->count++;
1066            return;
1067        }
1068        // Check if the maximum of the list has been reached
1069        originalCount = handleList->count;
1070        if(originalCount < count)
1071            handleList->count++;
1072        // Insert the handle to the list
1073        for(i = 0; i < originalCount; i++)
1074        {
1075            if(handleList->handle[i] > entityHandle)
1076            {
1077                for(j = handleList->count - 1; j > i; j--)
1078                {
1079                    handleList->handle[j] = handleList->handle[j - 1];
```

```
1080                }
1081                break;
1082            }
1083        }
1084        // If a slot was found, insert the handle in this position
1085        if(i < originalCount || handleList->count > originalCount)
1086            handleList->handle[i] = entityHandle;
1087        return;
1088    }
```

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

```
1089    TPMI_YES_NO
1090    NvCapGetPersistent(
1091        TPMI_DH_OBJECT   handle,        // IN: start handle
1092        UINT32           count,         // IN: maximum number of returned handles
1093        TPML_HANDLE     *handleList     // OUT: list of handle
1094        )
1095    {
1096        TPMI_YES_NO             more = NO;
1097        NV_REF                  iter = NV_REF_INIT;
1098        NV_REF                  currentAddr;
1099        TPM_HANDLE              entityHandle;
1100    //
1101        pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1102
1103        // Initialize output handle list
1104        handleList->count = 0;
1105
1106        // The maximum count of handles we may return is MAX_CAP_HANDLES
1107        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1108
1109        while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
1110        {
1111            // Ignore persistent handles that have values less than the input handle
1112            if(entityHandle < handle)
1113                continue;
1114            // if the handles in the list have reached the requested count, and there
1115            // are still handles need to be inserted, indicate that there are more.
1116            if(handleList->count == count)
1117                more = YES;
1118            // A handle with a value larger than start handle is a candidate
1119            // for return. Insert sort it to the return list.  Insert sort algorithm
1120            // is chosen here for simplicity based on the assumption that the total
1121            // number of NV indexes is small.  For an implementation that may allow
1122            // large number of NV indexes, a more efficient sorting algorithm may be
1123            // used here.
1124            InsertSort(handleList, count, entityHandle);
1125        }
1126        return more;
1127    }
```


8.4.5.23 NvCapGetIndex()

This function returns a list of handles of NV indexes, starting from *handle*. *Handle* must be in the range of NV indexes, 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

```
TPMI_YES_NO
NvCapGetIndex(
    TPMI_DH_OBJECT   handle,        // IN: start handle
    UINT32           count,         // IN: max number of returned handles
    TPML_HANDLE     *handleList     // OUT: list of handle
    )
{
    TPMI_YES_NO              more = NO;
    NV_REF                   iter = NV_REF_INIT;
    NV_REF                   currentAddr;
    TPM_HANDLE               nvHandle;
//
    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);

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

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

    while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
    {
        // Ignore index handles that have values less than the 'handle'
        if(nvHandle < handle)
            continue;
        // if the count of handles in the list has reached the requested count,
        // and there are still handles to report, set more.
        if(handleList->count == count)
            more = YES;
        // A handle with a value larger than start handle is a candidate
        // for return. Insert sort it to the return list.  Insert sort algorithm
        // is chosen here for simplicity based on the assumption that the total
        // number of NV indexes is small.  For an implementation that may allow
        // large number of NV indexes, a more efficient sorting algorithm may be
        // used here.
        InsertSort(handleList, count, nvHandle);
    }
    return more;
}
```

8.4.5.24 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```
UINT32
NvCapGetIndexNumber(
    void
    )
{
    UINT32           num = 0;
    NV_REF           iter = NV_REF_INIT;
//
```

```
1175        while(NvNextIndex(NULL, &iter) != 0)
1176            num++;
1177        return num;
1178    }
```

8.4.5.25 NvCapGetPersistentNumber()

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

```
1179    UINT32
1180    NvCapGetPersistentNumber(
1181        void
1182        )
1183    {
1184        UINT32          num = 0;
1185        NV_REF         iter = NV_REF_INIT;
1186        TPM_HANDLE      handle;
1187    //
1188        while(NvNextEvict(&handle, &iter) != 0)
1189            num++;
1190        return num;
1191    }
```

8.4.5.26 NvCapGetPersistentAvail()

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

```
1192    UINT32
1193    NvCapGetPersistentAvail(
1194        void
1195        )
1196    {
1197        UINT32          availNVSpace;
1198        UINT32          counterNum = NvCapGetCounterNumber();
1199        UINT32          reserved = sizeof(NV_LIST_TERMINATOR);
1200    //
1201        // Get the available space in NV storage
1202        availNVSpace = NvGetFreeBytes();
1203
1204        if(counterNum < MIN_COUNTER_INDICES)
1205        {
1206            // Some space has to be reserved for counter objects.
1207            reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
1208            if(reserved > availNVSpace)
1209                availNVSpace = 0;
1210            else
1211                availNVSpace -= reserved;
1212        }
1213        return availNVSpace / NV_EVICT_OBJECT_SIZE;
1214    }
```

8.4.5.27 NvCapGetCounterNumber()

Get the number of defined NV Indexes that are counter indexes.

```
1215    UINT32
1216    NvCapGetCounterNumber(
1217        void
1218        )
1219    {
```



```
1220        NV_REF            iter = NV_REF_INIT;
1221        NV_REF            currentAddr;
1222        UINT32            num = 0;
1223    //
1224        while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1225        {
1226            TPMA_NV             attributes = NvReadNvIndexAttributes(currentAddr);
1227            if(IsNvCounterIndex(attributes))
1228                num++;
1229        }
1230        return num;
1231    }
```

8.4.5.28 NvSetStartupAttributes()

Local function to set the attributes of an Index at TPM Reset and TPM Restart.

```
1232    static TPMA_NV
1233    NvSetStartupAttributes(
1234        TPMA_NV         attributes,         // IN: attributes to change
1235        STARTUP_TYPE     type           // IN: start up type
1236        )
1237    {
1238        // Clear read lock
1239        CLEAR_ATTRIBUTE(attributes, TPMA_NV, READLOCKED);
1240
1241        // Will change a non counter index to the unwritten state if:
1242        // a) TPMA_NV_CLEAR_STCLEAR is SET
1243        // b) orderly and TPM Reset
1244        if(!IsNvCounterIndex(attributes))
1245        {
1246            if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
1247               || (IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1248                   && (type == SU_RESET)))
1249                CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITTEN);
1250        }
1251        // Unlock any index that is not written or that does not have
1252        // TPMA_NV_WRITEDEFINE SET.
1253        if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
1254           || !IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
1255           CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1256        return attributes;
1257    }
```

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.

```
1258    void
1259    NvEntityStartup(
1260        STARTUP_TYPE     type           // IN: start up type
1261        )
1262    {
1263        NV_REF                iter = NV_REF_INIT;
1264        NV_RAM_REF            ramIter = NV_RAM_REF_INIT;
```

```
1265        NV_REF                currentAddr;        // offset points to the current entity
1266        NV_RAM_REF            currentRamAddr;
1267        TPM_HANDLE            nvHandle;
1268        TPMA_NV               attributes;
1269    //
1270        // Restore RAM index data
1271        NvRead(s_indexOrderlyRam, NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam));
1272
1273        // Initialize the max NV counter value
1274        NvSetMaxCount(NvGetMaxCount());
1275
1276        // If recovering from state save, do nothing else
1277        if(type == SU_RESUME)
1278            return;
1279        // Iterate all the NV Index to clear the locks
1280        while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1281        {
1282            attributes = NvReadNvIndexAttributes(currentAddr);
1283
1284            // If this is an orderly index, defer processing until loop below
1285            if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
1286                continue;
1287            // Set the attributes appropriate for this startup type
1288            attributes = NvSetStartupAttributes(attributes, type);
1289            NvWriteNvIndexAttributes(currentAddr, attributes);
1290        }
1291        // Iterate all the orderly indexes to clear the locks and initialize counters
1292        while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1293        {
1294            attributes = NvReadRamIndexAttributes(currentRamAddr);
1295
1296            attributes = NvSetStartupAttributes(attributes, type);
1297
1298            // update attributes in RAM
1299            NvWriteRamIndexAttributes(currentRamAddr, attributes);
1300
1301            // Set the lower bits in an orderly counter to 1 for a non-orderly startup
1302            if(IsNvCounterIndex(attributes)
1303               && (g_prevOrderlyState == SU_NONE_VALUE))
1304            {
1305                UINT64      counter;
1306    //
1307                // Read the counter value last saved to NV.
1308                counter = BYTE_ARRAY_TO_UINT64(currentRamAddr + sizeof(NV_RAM_HEADER));
1309
1310                // Set the lower bits of counter to 1's
1311                counter |= MAX_ORDERLY_COUNT;
1312
1313                // Write back to RAM
1314                // NOTE: Do not want to force a write to NV here. The counter value will
1315                // stay in RAM until the next shutdown or rollover.
1316                UINT64_TO_BYTE_ARRAY(counter, currentRamAddr + sizeof(NV_RAM_HEADER));
1317            }
1318        }
1319        return;
1320    }
```

8.4.5.30 NvCapGetCounterAvail()

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

```
1321    UINT32
1322    NvCapGetCounterAvail(
```

```
1323        void
1324        )
1325    {
1326        UINT32          availNVSpace;
1327        UINT32          availRAMSpace;
1328        UINT32          persistentNum = NvCapGetPersistentNumber();
1329        UINT32          reserved = sizeof(NV_LIST_TERMINATOR);
1330    //
1331        // Get the available space in NV storage
1332        availNVSpace = NvGetFreeBytes();
1333
1334        if(persistentNum < MIN_EVICT_OBJECTS)
1335        {
1336            // Some space has to be reserved for evict object. Adjust availNVSpace.
1337            reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
1338            if(reserved > availNVSpace)
1339                availNVSpace = 0;
1340            else
1341                availNVSpace -= reserved;
1342        }
1343        // Compute the available space in RAM
1344        availRAMSpace = RAM_ORDERLY_END - NvRamGetEnd();
1345
1346        // Return the min of counter number in NV and in RAM
1347        if(availNVSpace / NV_INDEX_COUNTER_SIZE
1348            > availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE)
1349            return availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE;
1350        else
1351            return availNVSpace / NV_INDEX_COUNTER_SIZE;
1352    }
```

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.

```
1353    NV_REF
1354    NvFindHandle(
1355        TPM_HANDLE          handle
1356        )
1357    {
1358        NV_REF              addr;
1359        NV_REF              iter = NV_REF_INIT;
1360        TPM_HANDLE          nextHandle;
1361    //
1362        while((addr = NvNext(&iter, &nextHandle)) != 0)
1363        {
1364            if(nextHandle == handle)
1365                break;
1366        }
1367        return addr;
1368    }
```

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.

```
UINT64
NvReadMaxCount(
    void
    )
{
    return s_maxCounter;
}
```

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.

```
void
NvUpdateMaxCount(
    UINT64          count
    )
{
    if(count > s_maxCounter)
        s_maxCounter = count;
}
```

8.4.6.4 NvSetMaxCount()

This function is used at NV initialization time to set the initial value of the maximum counter.

```
void
NvSetMaxCount(
    UINT64          value
    )
{
    s_maxCounter = value;
}
```

8.4.6.5 NvGetMaxCount()

Function to get the NV max counter value from the end-of-list marker

```
UINT64
NvGetMaxCount(
    void
    )
{
    NV_REF              iter = NV_REF_INIT;
    NV_REF              currentAddr;
    UINT64              maxCount;
//
    // Find the end of list marker and initialize the NV Max Counter value.
    while((currentAddr = NvNext(&iter, NULL )) != 0);
```



```
1402        // 'iter' should be pointing at the end of list marker so read in the current
1403        // value of the s_maxCounter.
1404        NvRead(&maxCount, iter + sizeof(UINT32), sizeof(maxCount));
1405
1406        return maxCount;
1407    }
```

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

```
#define NV_C
#include    "Tpm.h"
```

8.5.3 Functions

8.5.3.1 NvInitStatic()

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

```
static void
NvInitStatic(
    void
    )
{
    // In some implementations, the end of NV is variable and is set at boot time.
    // This value will be the same for each boot, but is not necessarily known
    // at compile time.
    s_evictNvEnd = (NV_REF)NV_MEMORY_SIZE;
    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(1)	all NV was initialized
FALSE(0)	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  #if 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      NvUpdateIndexOrderlyData();
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     pAssert(offset + size <= sizeof(gp));
121     MemorySet((&gp) + offset, 0, size);
122     NvWrite(offset, size, (&gp) + offset);
123 }
```

8.5.3.10 NvReadPersistent()

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

```
124 void
125 NvReadPersistent(
126     void
127     )
128 {
129     NvRead(&gp, NV_PERSISTENT_DATA, sizeof(gp));
130     return;
131 }
```

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

```
#define OBJECT_C
#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.

```
void
ObjectFlush(
    OBJECT          *object
    )
{
    object->attributes.occupied = CLEAR;
}
```

8.6.3.2 ObjectSetInUse()

This access function sets the occupied attribute of an object slot.

```
void
ObjectSetInUse(
    OBJECT          *object
    )
{
    object->attributes.occupied = SET;
}
```

8.6.3.3 ObjectStartup()

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

```
void
ObjectStartup(
    void
    )
{
    UINT32      i;
//
    // object slots initialization
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        //Set the slot to not occupied
        ObjectFlush(&s_objects[i]);
    }
```



```
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  //
39      // This has to be iterated because a command may have two handles
40      // and they may both be persistent.
41      // This could be made to be more efficient so that a search is not needed.
42      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
43      {
44          // If an object is a temporary evict object, flush it from slot
45          OBJECT      *object = &s_objects[i];
46          if(object->attributes.evict == SET)
47              ObjectFlush(object);
48      }
49      return;
50  }
```

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(1)	handle references a loaded object
FALSE(0)	handle is not an object handle, or it does not reference to a loaded object

```
51  BOOL
52  IsObjectPresent(
53      TPMI_DH_OBJECT   handle          // IN: handle to be checked
54      )
55  {
56      UINT32           slotIndex = handle - TRANSIENT_FIRST;
57      // Since the handle is just an index into the array that is zero based, any
58      // handle value outsize of the range of:
59      //    TRANSIENT_FIRST -- (TRANSIENT_FIRST + MAX_LOADED_OBJECT - 1)
60      // will now be greater than or equal to MAX_LOADED_OBJECTS
61      if(slotIndex >= MAX_LOADED_OBJECTS)
62          return FALSE;
63      // Indicate if the slot is occupied
64      return (s_objects[slotIndex].attributes.occupied == TRUE);
65  }
```

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(1)	object is an HMAC, hash, or event sequence object
FALSE(0)	object is not an HMAC, hash, or event sequence object

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

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.

```
OBJECT*
HandleToObject(
    TPMI_DH_OBJECT   handle         // IN: handle of the object
    )
{
    UINT32              index;
//
    // Return NULL if the handle references a permanent handle because there is no
    // associated OBJECT.
    if(HandleGetType(handle) == TPM_HT_PERMANENT)
        return NULL;
    // In this implementation, the handle is determined by the slot occupied by the
    // object.
    index = handle - TRANSIENT_FIRST;
    pAssert(index < MAX_LOADED_OBJECTS);
    pAssert(s_objects[index].attributes.occupied);
    return &s_objects[index];
}
```

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.

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

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.

```
101 void
102 GetQualifiedName(
103     TPMI_DH_OBJECT   handle,        // IN: handle of the object
104     TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
105     )
106 {
107     OBJECT      *object;
108 //
109     switch(HandleGetType(handle))
110     {
111         case TPM_HT_PERMANENT:
112             qualifiedName->t.size = sizeof(TPM_HANDLE);
113             UINT32_TO_BYTE_ARRAY(handle, qualifiedName->t.name);
114             break;
115         case TPM_HT_TRANSIENT:
116             object = HandleToObject(handle);
117             if(object == NULL || object->publicArea.nameAlg == TPM_ALG_NULL)
118                 qualifiedName->t.size = 0;
119             else
120                 // Copy the name
121                 *qualifiedName = object->qualifiedName;
122             break;
123         default:
124             FAIL(FATAL_ERROR_INTERNAL);
125     }
126     return;
127 }
```

8.6.3.10 ObjectGetHierarchy()

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

```
128 TPMI_RH_HIERARCHY
129 ObjectGetHierarchy(
130     OBJECT          *object         // IN :object
131     )
132 {
133     if(object->attributes.spsHierarchy)
134     {
135         return TPM_RH_OWNER;
136     }
137     else if(object->attributes.epsHierarchy)
138     {
139         return TPM_RH_ENDORSEMENT;
140     }
141     else if(object->attributes.ppsHierarchy)
142     {
143         return TPM_RH_PLATFORM;
144     }
145     else
146     {
147         return TPM_RH_NULL;
148     }
149 }
```

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.

```
150 TPMI_RH_HIERARCHY
151 GetHeriarchy(
152     TPMI_DH_OBJECT   handle          // IN :object handle
153     )
154 {
155     OBJECT           *object = HandleToObject(handle);
156 //
157     return ObjectGetHierarchy(object);
158 }
```

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

```
159 OBJECT *
160 FindEmptyObjectSlot(
161     TPMI_DH_OBJECT  *handle          // OUT: (optional)
162     )
163 {
164     UINT32                i;
165     OBJECT               *object;
166 //
167     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
168     {
169         object = &s_objects[i];
170         if(object->attributes.occupied == CLEAR)
171         {
172             if(handle)
173                 *handle = i + TRANSIENT_FIRST;
174             // Initialize the object attributes
175             MemorySet(&object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));
176             return object;
177         }
178     }
179     return NULL;
180 }
```

8.6.3.13 ObjectAllocateSlot()

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

```
181 OBJECT *
182 ObjectAllocateSlot(
183     TPMI_DH_OBJECT  *handle          // OUT: handle of allocated object
184     )
185 {
186     OBJECT           *object = FindEmptyObjectSlot(handle);
```



```
187 //
188     if(object != NULL)
189     {
190         // if found, mark as occupied
191         ObjectSetInUse(object);
192     }
193     return object;
194 }
```

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.

```
195 void
196 ObjectSetLoadedAttributes(
197     OBJECT          *object,        // IN: object attributes to finalize
198     TPM_HANDLE       parentHandle   // IN: the parent handle
199     )
200 {
201     OBJECT              *parent = HandleToObject(parentHandle);
202     TPMA_OBJECT          objectAttributes = object->publicArea.objectAttributes;
203 //
204     // Copy the stClear attribute from the public area. This could be overwritten
205     // if the parent has stClear SET
206     object->attributes.stClear =
207         IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear);
208     // If parent handle is a permanent handle, it is a primary (unless it is NULL
209     if(parent == NULL)
210     {
211         object->attributes.primary = SET;
212         switch(parentHandle)
213         {
214             case TPM_RH_ENDORSEMENT:
215                 object->attributes.epsHierarchy = SET;
216                 break;
217             case TPM_RH_OWNER:
218                 object->attributes.spsHierarchy = SET;
219                 break;
220             case TPM_RH_PLATFORM:
221                 object->attributes.ppsHierarchy = SET;
222                 break;
223             default:
224                 // Treat the temporary attribute as a hierarchy
225                 object->attributes.temporary = SET;
226                 object->attributes.primary = CLEAR;
227                 break;
228         }
229     }
230     else
231     {
232         // is this a stClear object
233         object->attributes.stClear =
234             (IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear)
235              || (parent->attributes.stClear == SET));
236         object->attributes.epsHierarchy = parent->attributes.epsHierarchy;
237         object->attributes.spsHierarchy = parent->attributes.spsHierarchy;
238         object->attributes.ppsHierarchy = parent->attributes.ppsHierarchy;
239         // An object is temporary if its parent is temporary or if the object
240         // is external
241         object->attributes.temporary = parent->attributes.temporary
242             || object->attributes.external;
243     }
244     // If this is an external object, set the QN == name but don't SET other
```

```
245     // key properties ('parent' or 'derived')
246     if(object->attributes.external)
247         object->qualifiedName = object->name;
248     else
249     {
250         // check attributes for different types of parents
251         if(IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, restricted)
252            && !object->attributes.publicOnly
253            && IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, decrypt)
254            && object->publicArea.nameAlg != TPM_ALG_NULL)
255         {
256             // This is a parent. If it is not a KEYEDHASH, it is an ordinary parent.
257             // Otherwise, it is a derivation parent.
258             if(object->publicArea.type == TPM_ALG_KEYEDHASH)
259                 object->attributes.derivation = SET;
260             else
261                 object->attributes.isParent = SET;
262         }
263         ComputeQualifiedName(parentHandle, object->publicArea.nameAlg,
264                              &object->name, &object->qualifiedName);
265     }
266     // Set slot occupied
267     ObjectSetInUse(object);
268     return;
269 }
```

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.

```
270 TPM_RC
271 ObjectLoad(
272     OBJECT          *object,        // IN: pointer to object slot
273                                     //     object
274     OBJECT          *parent,        // IN: (optional) the parent object
275     TPMT_PUBLIC     *publicArea,    // IN: public area to be installed in the object
276     TPMT_SENSITIVE  *sensitive,     // IN: (optional) sensitive area to be
277                                     //      installed in the object
278     TPM_RC           blamePublic,   // IN: parameter number to associate with the
279                                     //     publicArea errors
280     TPM_RC           blameSensitive,// IN: parameter number to associate with the
281                                     //     sensitive area errors
282     TPM2B_NAME      *name           // IN: (optional)
283     )
284 {
285     TPM_RC           result = TPM_RC_SUCCESS;
286     BOOL             doCheck;
287 //
288     // Do validations of public area object descriptions
289
290     // Is this public only or a no-name object?
291     if(sensitive == NULL || publicArea->nameAlg == TPM_ALG_NULL)
292     {
293         // Need to have schemes checked so that we do the right thing with the
294         // public key.
295         result = SchemeChecks(NULL, publicArea);
296     }
297     else
298     {
299         // For any sensitive area, make sure that the seedSize is no larger than the
300         // digest size of nameAlg
```



```
301         if(sensitive->seedValue.t.size
302                 > CryptHashGetDigestSize(publicArea->nameAlg))
303             return TPM_RCS_KEY_SIZE + blameSensitive;
304         // Check attributes and schemes for consistency
305         result = PublicAttributesValidation(parent, publicArea);
306     }
307     if(result != TPM_RC_SUCCESS)
308         return RcSafeAddToResult(result, blamePublic);
309     // If object == NULL, then this is am import. For import, load is not called
310     // unless the parent is fixedTPM.
311     if(object == NULL)
312         doCheck = TRUE;// //
313     // If the parent is not NULL, then this is an ordinary load and we only check
314     // if the parent is not fixedTPM
315     else if(parent != NULL)
316         doCheck = !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
317                                 TPMA_OBJECT, fixedTPM);
318     else
319         // This is a loadExternal. Check everything.
320         // Note: the check functions will filter things based on the name algorithm
321         // and whether or not both parts are loaded.
322         doCheck = TRUE;
323     // Note: the parent will be NULL if this is a load external. CryptValidateKeys()
324     // will only check the parts that need to be checked based on the settings
325     // of publicOnly and nameAlg.
326     // Note: For an RSA key, the keys sizes are checked but the binding is not
327     // checked.
328     if(doCheck)
329     {
330         // Do the cryptographic key validation
331         result = CryptValidateKeys(publicArea, sensitive, blamePublic,
332                                    blameSensitive);
333     }
334     // If this is an import, we are done
335     if(object == NULL || result != TPM_RC_SUCCESS)
336         return result;
337     // Set the name, if one was provided
338     if(name != NULL)
339         object->name = *name;
340     else
341         object->name.t.size = 0;
342     // Initialize public
343     object->publicArea = *publicArea;
344
345     // If there is a sensitive area, load it
346     if(sensitive == NULL)
347         object->attributes.publicOnly = SET;
348     else
349     {
350         object->sensitive = *sensitive;
351 #if     ALG_RSA
352         // If this is an RSA key that is not a parent, complete the load by
353         // computing the private exponent.
354         if(publicArea->type == ALG_RSA_VALUE)
355             result = CryptRsaLoadPrivateExponent(object);
356 #endif
357     }
358     return result;
359 }
```

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.

```
360 static HASH_OBJECT *
361 AllocateSequenceSlot(
362     TPM_HANDLE      *newHandle,     // OUT: receives the allocated handle
363     TPM2B_AUTH      *auth           // IN: the authValue for the slot
364     )
365 {
366     HASH_OBJECT      *object = (HASH_OBJECT *)ObjectAllocateSlot(newHandle);
367 //
368     // Validate that the proper location of the hash state data relative to the
369     // object state data. It would be good if this could have been done at compile
370     // time but it can't so do it in something that can be removed after debug.
371     cAssert(offsetof(HASH_OBJECT, auth) == offsetof(OBJECT, publicArea.authPolicy));
372
373     if(object != NULL)
374     {
375
376     // Set the common values that a sequence object shares with an ordinary object
377         // First, clear all attributes
378         MemorySet(&object->objectAttributes, 0, sizeof(TPMA_OBJECT));
379
380         // The type is TPM_ALG_NULL
381         object->type = TPM_ALG_NULL;
382
383         // This has no name algorithm and the name is the Empty Buffer
384         object->nameAlg = TPM_ALG_NULL;
385
386         // A sequence object is considered to be in the NULL hierarchy so it should
387         // be marked as temporary so that it can't be persisted
388         object->attributes.temporary = SET;
389
390         // A sequence object is DA exempt.
391         SET_ATTRIBUTE(object->objectAttributes, TPMA_OBJECT, noDA);
392
393         // Copy the authorization value
394         if(auth != NULL)
395             object->auth = *auth;
396         else
397             object->auth.t.size = 0;
398     }
399     return object;
400 }
401 #if CC_HMAC_Start || CC_MAC_Start
```

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

```
402 TPM_RC
403 ObjectCreateHMACSequence(
404     TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
405     OBJECT          *keyObject,     // IN: the object containing the HMAC key
406     TPM2B_AUTH      *auth,          // IN: authValue
407     TPMI_DH_OBJECT  *newHandle      // OUT: HMAC sequence object handle
408     )
409 {
410     HASH_OBJECT         *hmacObject;
411 //
412     // Try to allocate a slot for new object
413     hmacObject = AllocateSequenceSlot(newHandle, auth);
414
```



```
415         if(hmacObject == NULL)
416             return TPM_RC_OBJECT_MEMORY;
417         // Set HMAC sequence bit
418         hmacObject->attributes.hmacSeq = SET;
419
420     #if !SMAC_IMPLEMENTED
421         if(CryptHmacStart(&hmacObject->state.hmacState, hashAlg,
422                           keyObject->sensitive.sensitive.bits.b.size,
423                           keyObject->sensitive.sensitive.bits.b.buffer) == 0)
424     #else
425         if(CryptMacStart(&hmacObject->state.hmacState,
426                          &keyObject->publicArea.parameters,
427                          hashAlg, &keyObject->sensitive.sensitive.any.b) == 0)
428     #endif // SMAC_IMPLEMENTED
429             return TPM_RC_FAILURE;
430         return TPM_RC_SUCCESS;
431     }
432     #endif
```

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

```
433     TPM_RC
434     ObjectCreateHashSequence(
435         TPMI_ALG_HASH     hashAlg,        // IN: hash algorithm
436         TPM2B_AUTH       *auth,           // IN: authValue
437         TPMI_DH_OBJECT   *newHandle       // OUT: sequence object handle
438         )
439     {
440         HASH_OBJECT          *hashObject = AllocateSequenceSlot(newHandle, auth);
441     //
442         // See if slot allocated
443         if(hashObject == NULL)
444             return TPM_RC_OBJECT_MEMORY;
445         // Set hash sequence bit
446         hashObject->attributes.hashSeq = SET;
447
448         // Start hash for hash sequence
449         CryptHashStart(&hashObject->state.hashState[0], hashAlg);
450
451         return TPM_RC_SUCCESS;
452     }
```

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

```
453     TPM_RC
454     ObjectCreateEventSequence(
455         TPM2B_AUTH       *auth,           // IN: authValue
456         TPMI_DH_OBJECT   *newHandle       // OUT: sequence object handle
457         )
458     {
```

```
459     HASH_OBJECT         *hashObject = AllocateSequenceSlot(newHandle, auth);
460     UINT32               count;
461     TPM_ALG_ID           hash;
462 //
463     // See if slot allocated
464     if(hashObject == NULL)
465         return TPM_RC_OBJECT_MEMORY;
466     // Set the event sequence attribute
467     hashObject->attributes.eventSeq = SET;
468
469     // Initialize hash states for each implemented PCR algorithms
470     for(count = 0; (hash = CryptHashGetAlgByIndex(count)) != TPM_ALG_NULL; count++)
471         CryptHashStart(&hashObject->state.hashState[count], hash);
472     return TPM_RC_SUCCESS;
473 }
```

8.6.3.20 ObjectTerminateEvent()

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

```
474 void
475 ObjectTerminateEvent(
476     void
477     )
478 {
479     HASH_OBJECT         *hashObject;
480     int                  count;
481     BYTE                 buffer[MAX_DIGEST_SIZE];
482 //
483     hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
484
485     // Don't assume that this is a proper sequence object
486     if(hashObject->attributes.eventSeq)
487     {
488         // If it is, close any open hash contexts. This is done in case
489         // the cryptographic implementation has some context values that need to be
490         // cleaned up (hygiene).
491         //
492         for(count = 0; CryptHashGetAlgByIndex(count) != TPM_ALG_NULL; count++)
493         {
494             CryptHashEnd(&hashObject->state.hashState[count], 0, buffer);
495         }
496         // Flush sequence object
497         FlushObject(g_DRTMHandle);
498     }
499     g_DRTMHandle = TPM_RH_UNASSIGNED;
500 }
```

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
!= NULL	points to the loaded object

```
501 OBJECT *
502 ObjectContextLoad(
503     ANY_OBJECT_BUFFER   *object,        // IN: pointer to object structure in saved
504                                         //     context
505     TPMI_DH_OBJECT      *handle         // OUT: object handle
```



```
506         )
507     {
508         OBJECT      *newObject = ObjectAllocateSlot(handle);
509     //
510         // Try to allocate a slot for new object
511         if(newObject != NULL)
512         {
513             // Copy the first part of the object
514             MemoryCopy(newObject, object, offsetof(HASH_OBJECT, state));
515             // See if this is a sequence object
516             if(ObjectIsSequence(newObject))
517             {
518                 // If this is a sequence object, import the data
519                 SequenceDataImport((HASH_OBJECT *)newObject,
520                                    (HASH_OBJECT_BUFFER *)object);
521             }
522             else
523             {
524                 // Copy input object data to internal structure
525                 MemoryCopy(newObject, object, sizeof(OBJECT));
526             }
527         }
528         return newObject;
529     }
```

8.6.3.22 FlushObject()

This function frees an object slot.

This function requires that the object is loaded.

```
530     void
531     FlushObject(
532         TPMI_DH_OBJECT   handle          // IN: handle to be freed
533         )
534     {
535         UINT32      index = handle - TRANSIENT_FIRST;
536     //
537         pAssert(index < MAX_LOADED_OBJECTS);
538         // Clear all the object attributes
539         MemorySet((BYTE*)&(s_objects[index].attributes),
540                   0, sizeof(OBJECT_ATTRIBUTES));
541         return;
542     }
```

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.

```
543     void
544     ObjectFlushHierarchy(
545         TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flush
546         )
547     {
548         UINT16          i;
549     //
550         // iterate object slots
551         for(i = 0; i < MAX_LOADED_OBJECTS; i++)
552         {
553             if(s_objects[i].attributes.occupied)          // If found an occupied slot
554             {
555                 switch(hierarchy)
```

```
556                 {
557                     case TPM_RH_PLATFORM:
558                         if(s_objects[i].attributes.ppsHierarchy == SET)
559                             s_objects[i].attributes.occupied = FALSE;
560                         break;
561                     case TPM_RH_OWNER:
562                         if(s_objects[i].attributes.spsHierarchy == SET)
563                             s_objects[i].attributes.occupied = FALSE;
564                         break;
565                     case TPM_RH_ENDORSEMENT:
566                         if(s_objects[i].attributes.epsHierarchy == SET)
567                             s_objects[i].attributes.occupied = FALSE;
568                         break;
569                     default:
570                         FAIL(FATAL_ERROR_INTERNAL);
571                         break;
572                 }
573             }
574         }
575
576         return;
577     }
```

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

```
578 TPM_RC
579 ObjectLoadEvict(
580     TPM_HANDLE      *handle,        // IN:OUT: evict object handle.  If success, it
581                                     // will be replace by the loaded object handle
582     COMMAND_INDEX    commandIndex   // IN: the command being processed
583     )
584 {
585     TPM_RC          result;
586     TPM_HANDLE      evictHandle = *handle;   // Save the evict handle
587     OBJECT          *object;
588 //
589     // If this is an index that references a persistent object created by
590     // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
591     if(*handle >= PLATFORM_PERSISTENT)
592     {
593         // belongs to platform
594         if(g_phEnable == CLEAR)
595             return TPM_RC_HANDLE;
596     }
597     // belongs to owner
598     else if(gc.shEnable == CLEAR)
599         return TPM_RC_HANDLE;
600     // Try to allocate a slot for an object
601     object = ObjectAllocateSlot(handle);
602     if(object == NULL)
603         return TPM_RC_OBJECT_MEMORY;
604     // Copy persistent object to transient object slot.  A TPM_RC_HANDLE
605     // may be returned at this point. This will mark the slot as containing
606     // a transient object so that it will be flushed at the end of the
```



```
607     // command
608     result = NvGetEvictObject(evictHandle, object);
609
610     // Bail out if this failed
611     if(result != TPM_RC_SUCCESS)
612         return result;
613     // check the object to see if it is in the endorsement hierarchy
614     // if it is and this is not a TPM2_EvictControl() command, indicate
615     // that the hierarchy is disabled.
616     // If the associated hierarchy is disabled, make it look like the
617     // handle is not defined
618     if(ObjectGetHierarchy(object) == TPM_RH_ENDORSEMENT
619        && gc.ehEnable == CLEAR
620        && GetCommandCode(commandIndex) != TPM_CC_EvictControl)
621         return TPM_RC_HANDLE;
622
623     return result;
624 }
```

8.6.3.25 ObjectComputeName()

This does the name computation from a public area (can be marshaled or not).

```
625 TPM2B_NAME *
626 ObjectComputeName(
627     UINT32           size,          // IN: the size of the area to digest
628     BYTE            *publicArea,    // IN: the public area to digest
629     TPM_ALG_ID       nameAlg,       // IN: the hash algorithm to use
630     TPM2B_NAME      *name           // OUT: Computed name
631     )
632 {
633     // Hash the publicArea into the name buffer leaving room for the nameAlg
634     name->t.size = CryptHashBlock(nameAlg, size, publicArea,
635                                   sizeof(name->t.name) - 2,
636                                   &name->t.name[2]);
637     // set the nameAlg
638     UINT16_TO_BYTE_ARRAY(nameAlg, name->t.name);
639     name->t.size += 2;
640     return name;
641 }
```

8.6.3.26 PublicMarshalAndComputeName()

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

```
642 TPM2B_NAME *
643 PublicMarshalAndComputeName(
644     TPMT_PUBLIC     *publicArea,    // IN: public area of an object
645     TPM2B_NAME      *name           // OUT: name of the object
646     )
647 {
648     // Will marshal a public area into a template. This is because the internal
649     // format for a TPM2B_PUBLIC is a structure and not a simple BYTE buffer.
650     TPM2B_TEMPLATE       marshaled;     // this is big enough to hold a
651                                         //  marshaled TPMT_PUBLIC
652     BYTE                *buffer = (BYTE *)&marshaled.t.buffer;
653 //
654     // if the nameAlg is NULL then there is no name.
655     if(publicArea->nameAlg == TPM_ALG_NULL)
656         name->t.size = 0;
657     else
658     {
659         // Marshal the public area into its canonical form
```

```
660         marshaled.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
661         // and compute the name
662         ObjectComputeName(marshaled.t.size, marshaled.t.buffer,
663                           publicArea->nameAlg, name);
664     }
665     return name;
666 }
```

8.6.3.27 ComputeQualifiedName()

This function computes the qualified name of an object.

```
667 void
668 ComputeQualifiedName(
669     TPM_HANDLE       parentHandle,  // IN: parent's handle
670     TPM_ALG_ID       nameAlg,       // IN: name hash
671     TPM2B_NAME      *name,          // IN: name of the object
672     TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
673     )
674 {
675     HASH_STATE      hashState;   // hash state
676     TPM2B_NAME      parentName;
677 //
678     if(parentHandle == TPM_RH_UNASSIGNED)
679     {
680         MemoryCopy2B(&qualifiedName->b, &name->b, sizeof(qualifiedName->t.name));
681         *qualifiedName = *name;
682     }
683     else
684     {
685         GetQualifiedName(parentHandle, &parentName);
686
687         //      QN_A = hash_A (QN of parent || NAME_A)
688
689         // Start hash
690         qualifiedName->t.size = CryptHashStart(&hashState, nameAlg);
691
692         // Add parent's qualified name
693         CryptDigestUpdate2B(&hashState, &parentName.b);
694
695         // Add self name
696         CryptDigestUpdate2B(&hashState, &name->b);
697
698         // Complete hash leaving room for the name algorithm
699         CryptHashEnd(&hashState, qualifiedName->t.size,
700                      &qualifiedName->t.name[2]);
701         UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
702         qualifiedName->t.size += 2;
703     }
704     return;
705 }
```

8.6.3.28 ObjectIsStorage()

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

Return Value	Meaning
TRUE(1)	object is a storage key
FALSE(0)	object is not a storage key

```
BOOL
ObjectIsStorage(
    TPMI_DH_OBJECT   handle         // IN: object handle
    )
{
    OBJECT           *object = HandleToObject(handle);
    TPMT_PUBLIC      *publicArea = ((object != NULL) ? &object->publicArea : NULL);
//
    return (publicArea != NULL
            && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
            && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
            && !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
            && (object->publicArea.type == ALG_RSA_VALUE
                || object->publicArea.type == ALG_ECC_VALUE));
}
```

8.6.3.29 ObjectCapGetLoaded()

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

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

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

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

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

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

            if(handleList->count < count)
            {
                // If we have not filled up the return list, add this object
                // handle to it
                handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
```

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

    return more;
}
```

8.6.3.30 ObjectCapGetTransientAvail()

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

```
UINT32
ObjectCapGetTransientAvail(
    void
    )
{
    UINT32      i;
    UINT32      num = 0;
//
    // Iterate object slot to get the number of unoccupied slots
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        if(s_objects[i].attributes.occupied == FALSE) num++;
    }

    return num;
}
```

8.6.3.31 ObjectGetPublicAttributes()

Returns the attributes associated with an object handles.

```
TPMA_OBJECT
ObjectGetPublicAttributes(
    TPM_HANDLE          handle
    )
{
    return HandleToObject(handle)->publicArea.objectAttributes;
}
OBJECT_ATTRIBUTES
ObjectGetProperties(
    TPM_HANDLE          handle
    )
{
    return HandleToObject(handle)->attributes;
}
```


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

```
#define PCR_C
#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.

```
static const PCR_Attributes s_initAttributes[] =
{
    // PCR 0 - 15, static RTM
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},

    {0, 0x0F, 0x1F},        // PCR 16, Debug
    {0, 0x10, 0x1C},        // PCR 17, Locality 4
    {0, 0x10, 0x1C},        // PCR 18, Locality 3
    {0, 0x10, 0x0C},        // PCR 19, Locality 2
    {0, 0x14, 0x0E},        // PCR 20, Locality 1
    {0, 0x14, 0x04},        // PCR 21, Dynamic OS
    {0, 0x14, 0x04},        // PCR 22, Dynamic OS
    {0, 0x0F, 0x1F},        // PCR 23, Application specific
    {0, 0x0F, 0x1F}         // PCR 24, testing policy
};
```

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(1)	PCR belongs an authorization group
FALSE(0)	PCR does not belong an authorization group

```
BOOL
PCRBelongsAuthGroup(
    TPMI_DH_PCR      handle,        // IN: handle of PCR
    UINT32          *groupIndex     // OUT: group index if PCR belongs a
```

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

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(1)	PCR belongs a policy group
FALSE(0)	PCR does not belong a policy group

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

8.7.3.3 PCRBelongsTCBGroup()

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

Return Value	Meaning
TRUE(1)	PCR belongs to TCB group
FALSE(0)	PCR does not belong to TCB group

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

8.7.3.4 PCRPolicyIsAvailable()

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

Return Value	Meaning
TRUE(1)	the PCR should be authorized by policy
FALSE(0)	the PCR does not allow policy

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

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.

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



```
107         }
108     }
```

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.

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

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.

```
128     void
129     PCRSimStart(
130         void
131         )
132     {
133         UINT32  i;
134     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
135         for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
136         {
137             gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
138             gp.pcrPolicies.policy[i].t.size = 0;
139         }
140     #endif
141     #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
142         for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
143         {
144             gc.pcrAuthValues.auth[i].t.size = 0;
145         }
146     #endif
147         // We need to give an initial configuration on allocated PCR before
148         // receiving any TPM2_PCR_Allocate command to change this configuration
149         // When the simulation environment starts, we allocate all the PCRs
150         for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
151         gp.pcrAllocated.count++)
152         {
153             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
154                 = CryptHashGetAlgByIndex(gp.pcrAllocated.count);
155
156             gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
```



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

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
!= NULL	pointer to the 0th byte of the 0th PCR

```
169 static BYTE *
170 GetSavedPcrPointer(
171     TPM_ALG_ID        alg,            // IN: algorithm for bank
172     UINT32            pcrIndex        // IN: PCR index in PCR_SAVE
173     )
174 {
175     switch(alg)
176     {
177 #if     ALG_SHA1
178         case ALG_SHA1_VALUE:
179             return gc.pcrSave.sha1[pcrIndex];
180             break;
181 #endif
182 #if     ALG_SHA256
183         case ALG_SHA256_VALUE:
184             return gc.pcrSave.sha256[pcrIndex];
185             break;
186 #endif
187 #if     ALG_SHA384
188         case ALG_SHA384_VALUE:
189             return gc.pcrSave.sha384[pcrIndex];
190             break;
191 #endif
192
193 #if     ALG_SHA512
194         case ALG_SHA512_VALUE:
195             return gc.pcrSave.sha512[pcrIndex];
196             break;
197 #endif
198 #if     ALG_SM3_256
199         case ALG_SM3_256_VALUE:
200             return gc.pcrSave.sm3_256[pcrIndex];
201             break;
202 #endif
203         default:
204             break;
205     }
206     FAIL(FATAL_ERROR_INTERNAL);
207 }
```

8.7.3.9 PcrIsAllocated()

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

Return Value	Meaning
TRUE(1)	PCR is allocated
FALSE(0)	PCR is not allocated

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

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
!= NULL	pointer to the 0th byte of the 0th PCR

```
234  static BYTE *
235  GetPcrPointer(
236      TPM_ALG_ID      alg,            // IN: algorithm for bank
237      UINT32          pcrNumber       // IN: PCR number
238      )
239  {
240      static BYTE     *pcr = NULL;
241
242      if(!PcrIsAllocated(pcrNumber, alg))
243          return NULL;
244
245      switch(alg)
246      {
247  #if     ALG_SHA1
248          case ALG_SHA1_VALUE:
249              pcr = s_pcrs[pcrNumber].sha1Pcr;
250              break;
```



```
251     #endif
252     #if     ALG_SHA256
253             case ALG_SHA256_VALUE:
254                 pcr = s_pcrs[pcrNumber].sha256Pcr;
255                 break;
256     #endif
257     #if     ALG_SHA384
258             case ALG_SHA384_VALUE:
259                 pcr = s_pcrs[pcrNumber].sha384Pcr;
260                 break;
261     #endif
262     #if     ALG_SHA512
263             case ALG_SHA512_VALUE:
264                 pcr = s_pcrs[pcrNumber].sha512Pcr;
265                 break;
266     #endif
267     #if     ALG_SM3_256
268             case ALG_SM3_256_VALUE:
269                 pcr = s_pcrs[pcrNumber].sm3_256Pcr;
270                 break;
271     #endif
272             default:
273                 FAIL(FATAL_ERROR_INTERNAL);
274                 break;
275         }
276
277         return pcr;
278     }
```

8.7.3.11 IsPcrSelected()

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

Return Value	Meaning
TRUE(1)	PCR is selected
FALSE(0)	PCR is not selected

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

8.7.3.12 FilterPcr()

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

```
290     static void
291     FilterPcr(
292         TPMS_PCR_SELECTION  *selection          // IN: input PCR selection
293         )
294     {
295         UINT32      i;
296         TPMS_PCR_SELECTION      *allocated = NULL;
297
```

```
298         // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
299         for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
300             selection->pcrSelect[i] = 0;
301
302         // Find the internal configuration for the bank
303         for(i = 0; i < gp.pcrAllocated.count; i++)
304         {
305             if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
306             {
307                 allocated = &gp.pcrAllocated.pcrSelections[i];
308                 break;
309             }
310         }
311
312         for(i = 0; i < selection->sizeofSelect; i++)
313         {
314             if(allocated == NULL)
315             {
316                 // If the required bank does not exist, clear input selection
317                 selection->pcrSelect[i] = 0;
318             }
319             else
320                 selection->pcrSelect[i] &= allocated->pcrSelect[i];
321         }
322
323         return;
324     }
```

8.7.3.13 PcrDrtm()

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

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

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().

```
349     void
```



```
350 PCR_ClearAuth(
351     void
352     )
353 {
354 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
355     int         j;
356     for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
357     {
358         gc.pcrAuthValues.auth[j].t.size = 0;
359     }
360 #endif
361 }
```

8.7.3.15 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```
362 void
363 PCRStartup(
364     STARTUP_TYPE     type,          // IN: startup type
365     BYTE             locality       // IN: startup locality
366     )
367 {
368     UINT32              pcr, j;
369     UINT32              saveIndex = 0;
370
371     g_pcrReConfig = FALSE;
372
373     // Don't test for SU_RESET because that should be the default when nothing
374     // else is selected
375     if(type != SU_RESUME && type != SU_RESTART)
376     {
377         // PCR generation counter is cleared at TPM_RESET
378         gr.pcrCounter = 0;
379     }
380
381     // Initialize/Restore PCR values
382     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
383     {
384         // On resume, need to know if this PCR had its state saved or not
385         UINT32      stateSaved;
386
387         if(type == SU_RESUME
388            &&  s_initAttributes[pcr].stateSave == SET)
389         {
390             stateSaved = 1;
391         }
392         else
393         {
394             stateSaved = 0;
395             PCRChanged(pcr);
396         }
397
398         // If this is the H-CRTM PCR and we are not doing a resume and we
399         // had an H-CRTM event, then we don't change this PCR
400         if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
401             continue;
402
403         // Iterate each hash algorithm bank
404         for(j = 0; j < gp.pcrAllocated.count; j++)
405         {
406             TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
407             BYTE            *pcrData = GetPcrPointer(hash, pcr);
408             UINT16           pcrSize = CryptHashGetDigestSize(hash);
```

```
409
410             if(pcrData != NULL)
411             {
412                 // if state was saved
413                 if(stateSaved == 1)
414                 {
415                     // Restore saved PCR value
416                     BYTE    *pcrSavedData;
417                     pcrSavedData = GetSavedPcrPointer(
418                         gp.pcrAllocated.pcrSelections[j].hash,
419                         saveIndex);
420                     MemoryCopy(pcrData, pcrSavedData, pcrSize);
421                 }
422                 else
423                     // PCR was not restored by state save
424                 {
425                     // If the reset locality of the PCR is 4, then
426                     // the reset value is all one's, otherwise it is
427                     // all zero.
428                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
429                         MemorySet(pcrData, 0xFF, pcrSize);
430                     else
431                     {
432                         MemorySet(pcrData, 0, pcrSize);
433                         if(pcr == HCRTM_PCR)
434                             pcrData[pcrSize - 1] = locality;
435                     }
436                 }
437             }
438         }
439         saveIndex += stateSaved;
440     }
441     // Reset authValues on TPM2_Startup(CLEAR)
442     if(type != SU_RESUME)
443         PCR_ClearAuth();
444 }
```

8.7.3.16 PCRStateSave()

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

```
445 void
446 PCRStateSave(
447     TPM_SU          type            // IN: startup type
448     )
449 {
450     UINT32          pcr, j;
451     UINT32          saveIndex = 0;
452
453     // if state save CLEAR, nothing to be done.  Return here
454     if(type == TPM_SU_CLEAR)
455         return;
456
457     // Copy PCR values to the structure that should be saved to NV
458     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
459     {
460         UINT32  stateSaved = (s_initAttributes[pcr].stateSave == SET) ? 1 : 0;
461
462         // Iterate each hash algorithm bank
463         for(j = 0; j < gp.pcrAllocated.count; j++)
464         {
465             BYTE    *pcrData;
466             UINT32  pcrSize;
467
```



```
468             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
469
470             if(pcrData != NULL)
471             {
472                 pcrSize
473                     = CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
474
475                 if(stateSaved == 1)
476                 {
477                     // Restore saved PCR value
478                     BYTE    *pcrSavedData;
479                     pcrSavedData
480                         = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
481                                              saveIndex);
482                     MemoryCopy(pcrSavedData, pcrData, pcrSize);
483                 }
484             }
485         }
486         saveIndex += stateSaved;
487     }
488
489     return;
490 }
```

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(1)	PCR is state saved
FALSE(0)	PCR is not state saved

```
491 BOOL
492 PCRIsStateSaved(
493     TPMI_DH_PCR      handle          // IN: PCR handle to be extended
494     )
495 {
496     UINT32              pcr = handle - PCR_FIRST;
497
498     if(s_initAttributes[pcr].stateSave == SET)
499         return TRUE;
500     else
501         return FALSE;
502 }
```

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(1)	TPM2_PCR_Reset() is allowed
FALSE(0)	TPM2_PCR_Reset() is not allowed

```
503 BOOL
504 PCRIsResetAllowed(
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
512     // Check for the locality
513     commandLocality = _plat__LocalityGet();
514
515 #ifdef DRTM_PCR
516     // For a TPM that does DRTM, Reset is not allowed at locality 4
517     if(commandLocality == 4)
518         return FALSE;
519 #endif
520
521     localityBits = localityBits << commandLocality;
522     if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
523         return FALSE;
524     else
525         return TRUE;
526 }
```

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().

```
527 void
528 PCRChanged(
529     TPM_HANDLE      pcrHandle       // IN: the handle of the PCR that changed.
530     )
531 {
532     // For the reference implementation, the only change that does not cause
533     // increment is a change to a PCR in the TCB group.
534     if((pcrHandle == 0) || !PCRBelongsTCBGroup(pcrHandle))
535     {
536         gr.pcrCounter++;
537         if(gr.pcrCounter == 0)
538             FAIL(FATAL_ERROR_COUNTER_OVERFLOW);
539     }
540 }
```

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(1)	extend is allowed
FALSE(0)	extend is not allowed

```
541 BOOL
542 PCRIsExtendAllowed(
543     TPMI_DH_PCR     handle          // IN: PCR handle to be extended
544     )
545 {
546     UINT8               commandLocality;
547     UINT8               localityBits = 1;
548     UINT32              pcr = handle - PCR_FIRST;
```



```
549
550         // Check for the locality
551         commandLocality = _plat__LocalityGet();
552         localityBits = localityBits << commandLocality;
553         if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
554             return FALSE;
555         else
556             return TRUE;
557     }
```

8.7.3.21 PCRExtend()

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

```
558     void
559     PCRExtend(
560         TPMI_DH_PCR      handle,        // IN: PCR handle to be extended
561         TPMI_ALG_HASH    hash,          // IN: hash algorithm of PCR
562         UINT32           size,          // IN: size of data to be extended
563         BYTE            *data           // IN: data to be extended
564         )
565     {
566         BYTE                *pcrData;
567         HASH_STATE           hashState;
568         UINT16               pcrSize;
569
570         pcrData = GetPcrPointer(hash, handle - PCR_FIRST);
571
572         // Extend PCR if it is allocated
573         if(pcrData != NULL)
574         {
575             pcrSize = CryptHashGetDigestSize(hash);
576             CryptHashStart(&hashState, hash);
577             CryptDigestUpdate(&hashState, pcrSize, pcrData);
578             CryptDigestUpdate(&hashState, size, data);
579             CryptHashEnd(&hashState, pcrSize, pcrData);
580
581             // PCR has changed so update the pcrCounter if necessary
582             PCRChanged(handle);
583         }
584
585         return;
586     }
```

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.

```
587     void
588     PCRComputeCurrentDigest(
589         TPMI_ALG_HASH        hashAlg,       // IN: hash algorithm to compute digest
590         TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
591                                             //     output)
592         TPM2B_DIGEST        *digest         // OUT: digest
593         )
594     {
595         HASH_STATE               hashState;
596         TPMS_PCR_SELECTION      *select;
597         BYTE                    *pcrData;   // will point to a digest
598         UINT32                   pcrSize;
599         UINT32                   pcr;
```

```
600     UINT32                  i;
601
602     // Initialize the hash
603     digest->t.size = CryptHashStart(&hashState, hashAlg);
604     pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
605
606     // Iterate through the list of PCR selection structures
607     for(i = 0; i < selection->count; i++)
608     {
609         // Point to the current selection
610         select = &selection->pcrSelections[i]; // Point to the current selection
611         FilterPcr(select);      // Clear out the bits for unimplemented PCR
612
613         // Need the size of each digest
614         pcrSize = CryptHashGetDigestSize(selection->pcrSelections[i].hash);
615
616         // Iterate through the selection
617         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
618         {
619             if(IsPcrSelected(pcr, select))        // Is this PCR selected
620             {
621                 // Get pointer to the digest data for the bank
622                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
623                 pAssert(pcrData != NULL);
624                 CryptDigestUpdate(&hashState, pcrSize, pcrData);  // add to digest
625             }
626         }
627     }
628     // Complete hash stack
629     CryptHashEnd2B(&hashState, &digest->b);
630
631     return;
632 }
```

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.

```
633 void
634 PCRRead(
635     TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
636                                         //     output)
637     TPML_DIGEST         *digest,        // OUT: digest
638     UINT32              *pcrCounter     // OUT: the current value of PCR generation
639                                         //     number
640     )
641 {
642     TPMS_PCR_SELECTION      *select;
643     BYTE                    *pcrData;       // will point to a digest
644     UINT32                   pcr;
645     UINT32                   i;
646
647     digest->count = 0;
648
649     // Iterate through the list of PCR selection structures
650     for(i = 0; i < selection->count; i++)
651     {
652         // Point to the current selection
653         select = &selection->pcrSelections[i]; // Point to the current selection
654         FilterPcr(select);      // Clear out the bits for unimplemented PCR
655
656         // Iterate through the selection
657         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
```



```
658         {
659             if(IsPcrSelected(pcr, select))         // Is this PCR selected
660             {
661                 // Check if number of digest exceed upper bound
662                 if(digest->count > 7)
663                 {
664                     // Clear rest of the current select bitmap
665                     while(pcr < IMPLEMENTATION_PCR
666                             // do not round up!
667                           && (pcr / 8) < select->sizeofSelect)
668                     {
669                         // do not round up!
670                         select->pcrSelect[pcr / 8] &= (BYTE)~(1 << (pcr % 8));
671                         pcr++;
672                     }
673                     // Exit inner loop
674                     break;
675                 }
676                 // Need the size of each digest
677                 digest->digests[digest->count].t.size =
678                     CryptHashGetDigestSize(selection->pcrSelections[i].hash);
679
680                 // Get pointer to the digest data for the bank
681                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
682                 pAssert(pcrData != NULL);
683                 // Add to the data to digest
684                 MemoryCopy(digest->digests[digest->count].t.buffer,
685                            pcrData,
686                            digest->digests[digest->count].t.size);
687                 digest->count++;
688             }
689         }
690         // If we exit inner loop because we have exceed the output upper bound
691         if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
692         {
693             // Clear rest of the selection
694             while(i < selection->count)
695             {
696                 MemorySet(selection->pcrSelections[i].pcrSelect, 0,
697                           selection->pcrSelections[i].sizeofSelect);
698                 i++;
699             }
700             // exit outer loop
701             break;
702         }
703     }
704
705     *pcrCounter = gr.pcrCounter;
706
707     return;
708 }
```

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.

```
709 void
710 PcrWrite(
711     TPMI_DH_PCR      handle,        // IN: PCR handle to be extended
712     TPMI_ALG_HASH    hash,          // IN: hash algorithm of PCR
713     TPM2B_DIGEST    *digest         // IN: the new value
714     )
715 {
716     UINT32               pcr = handle - PCR_FIRST;
```

```
    BYTE                *pcrData;

    // Copy value to the PCR if it is allocated
    pcrData = GetPcrPointer(hash, pcr);
    if(pcrData != NULL)
    {
        MemoryCopy(pcrData, digest->t.buffer, digest->t.size);
    }

    return;
}
```

8.7.3.25 PCRAllocate()

This function is used to change the PCR allocation.

Error Returns	Meaning
TPM_RC_NO_RESULT	allocate failed
TPM_RC_PCR	improper allocation

```
TPM_RC
PCRAllocate(
    TPML_PCR_SELECTION  *allocate,      // IN: required allocation
    UINT32              *maxPCR,        // OUT: Maximum number of PCR
    UINT32              *sizeNeeded,    // OUT: required space
    UINT32              *sizeAvailable  // OUT: available space
    )
{
    UINT32                  i, j, k;
    TPML_PCR_SELECTION      newAllocate;
    // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
    BOOL                    pcrHcrtm = FALSE;
    BOOL                    pcrDrtm = FALSE;

    // Create the expected new PCR allocation based on the existing allocation
    // and the new input:
    //  1. if a PCR bank does not appear in the new allocation, the existing
    //     allocation of this PCR bank will be preserved.
    //  2. if a PCR bank appears multiple times in the new allocation, only the
    //     last one will be in effect.
    newAllocate = gp.pcrAllocated;
    for(i = 0; i < allocate->count; i++)
    {
        for(j = 0; j < newAllocate.count; j++)
        {
            // If hash matches, the new allocation covers the old allocation
            // for this particular bank.
            // The assumption is the initial PCR allocation (from manufacture)
            // has all the supported hash algorithms with an assigned bank
            // (possibly empty).  So there must be a match for any new bank
            // allocation from the input.
            if(newAllocate.pcrSelections[j].hash ==
               allocate->pcrSelections[i].hash)
            {
                newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
                break;
            }
        }
        // The j loop must exit with a match.
        pAssert(j < newAllocate.count);
    }

```



```
770     // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
771     *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
772     if(*maxPCR > IMPLEMENTATION_PCR)
773         *maxPCR = IMPLEMENTATION_PCR;
774
775     // Compute required size for allocation
776     *sizeNeeded = 0;
777     for(i = 0; i < newAllocate.count; i++)
778     {
779         UINT32      digestSize
780             = CryptHashGetDigestSize(newAllocate.pcrSelections[i].hash);
781 #if defined(DRTM_PCR)
782         // Make sure that we end up with at least one DRTM PCR
783         pcrDrtm = pcrDrtm || TestBit(DRTM_PCR,
784                                      newAllocate.pcrSelections[i].pcrSelect,
785                                      newAllocate.pcrSelections[i].sizeofSelect);
786
787 #else   // if DRTM PCR is not required, indicate that the allocation is OK
788         pcrDrtm = TRUE;
789 #endif
790
791 #if defined(HCRTM_PCR)
792         // and one HCRTM PCR (since this is usually PCR 0...)
793         pcrHcrtm = pcrHcrtm || TestBit(HCRTM_PCR,
794                                        newAllocate.pcrSelections[i].pcrSelect,
795                                        newAllocate.pcrSelections[i].sizeofSelect);
796 #else
797         pcrHcrtm = TRUE;
798 #endif
799         for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
800         {
801             BYTE        mask = 1;
802             for(k = 0; k < 8; k++)
803             {
804                 if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
805                     *sizeNeeded += digestSize;
806                 mask = mask << 1;
807             }
808         }
809     }
810
811     if(!pcrDrtm || !pcrHcrtm)
812         return TPM_RC_PCR;
813
814     // In this particular implementation, we always have enough space to
815     // allocate PCR.  Different implementation may return a sizeAvailable less
816     // than the sizeNeed.
817     *sizeAvailable = sizeof(s_pcrs);
818
819     // Save the required allocation to NV.  Note that after NV is written, the
820     // PCR allocation in NV is no longer consistent with the RAM data
821     // gp.pcrAllocated.  The NV version reflect the allocate after next
822     // TPM_RESET, while the RAM version reflects the current allocation
823     NV_WRITE_PERSISTENT(pcrAllocated, newAllocate);
824
825     return TPM_RC_SUCCESS;
826 }
```

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.

```
827 void
```

```
828 PCRSetValue(
829     TPM_HANDLE        handle,        // IN: the handle of the PCR to set
830     INT8              initialValue   // IN: the value to set
831     )
832 {
833     int               i;
834     UINT32            pcr = handle - PCR_FIRST;
835     TPMI_ALG_HASH     hash;
836     UINT16            digestSize;
837     BYTE             *pcrData;
838
839     // Iterate supported PCR bank algorithms to reset
840     for(i = 0; i < HASH_COUNT; i++)
841     {
842         hash = CryptHashGetAlgByIndex(i);
843         // Prevent runaway
844         if(hash == TPM_ALG_NULL)
845             break;
846
847         // Get a pointer to the data
848         pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
849
850         // If the PCR is allocated
851         if(pcrData != NULL)
852         {
853             // And the size of the digest
854             digestSize = CryptHashGetDigestSize(hash);
855
856             // Set the LSO to the input value
857             pcrData[digestSize - 1] = initialValue;
858
859             // Sign extend
860             if(initialValue >= 0)
861                 MemorySet(pcrData, 0, digestSize - 1);
862             else
863                 MemorySet(pcrData, -1, digestSize - 1);
864         }
865     }
866 }
```

8.7.3.27 PCRResetDynamics

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

```
867 void
868 PCRResetDynamics(
869     void
870     )
871 {
872     UINT32              pcr, i;
873
874     // Initialize PCR values
875     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
876     {
877         // Iterate each hash algorithm bank
878         for(i = 0; i < gp.pcrAllocated.count; i++)
879         {
880             BYTE    *pcrData;
881             UINT32  pcrSize;
882
883             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
884
885             if(pcrData != NULL)
886             {
```

```
                    pcrSize =
                        CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[i].hash);

                    // Reset PCR
                    // Any PCR can be reset by locality 4 should be reset to 0
                    if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
                        MemorySet(pcrData, 0, pcrSize);
                }
            }
        }
        return;
    }
```

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

```
TPMI_YES_NO
PCRCapGetAllocation(
    UINT32                 count,          // IN: count of return
    TPML_PCR_SELECTION  *pcrSelection      // OUT: PCR allocation list
    )
{
    if(count == 0)
    {
        pcrSelection->count = 0;
        return YES;
    }
    else
    {
        *pcrSelection = gp.pcrAllocated;
        return NO;
    }
}
```

8.7.3.29 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```
static void
PCRSetSelectBit(
    UINT32            pcr,            // IN: PCR number
    BYTE             *bitmap          // OUT: bit map to be set
    )
{
    bitmap[pcr / 8] |= (1 << (pcr % 8));
    return;
}
```

8.7.3.30 PCRGetProperty()

This function returns the selected PCR property.

Return Value	Meaning
TRUE(1)	the property type is implemented
FALSE(0)	the property type is not implemented

```
925  static BOOL
926  PCRGetProperty(
927      TPM_PT_PCR                property,
928      TPMS_TAGGED_PCR_SELECT   *select
929      )
930  {
931      UINT32               pcr;
932      UINT32               groupIndex;
933
934      select->tag = property;
935      // Always set the bitmap to be the size of all PCR
936      select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
937
938      // Initialize bitmap
939      MemorySet(select->pcrSelect, 0, select->sizeofSelect);
940
941      // Collecting properties
942      for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
943      {
944          switch(property)
945          {
946              case TPM_PT_PCR_SAVE:
947                  if(s_initAttributes[pcr].stateSave == SET)
948                      PCRSetSelectBit(pcr, select->pcrSelect);
949                  break;
950              case TPM_PT_PCR_EXTEND_L0:
951                  if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
952                      PCRSetSelectBit(pcr, select->pcrSelect);
953                  break;
954              case TPM_PT_PCR_RESET_L0:
955                  if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
956                      PCRSetSelectBit(pcr, select->pcrSelect);
957                  break;
958              case TPM_PT_PCR_EXTEND_L1:
959                  if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
960                      PCRSetSelectBit(pcr, select->pcrSelect);
961                  break;
962              case TPM_PT_PCR_RESET_L1:
963                  if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
964                      PCRSetSelectBit(pcr, select->pcrSelect);
965                  break;
966              case TPM_PT_PCR_EXTEND_L2:
967                  if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
968                      PCRSetSelectBit(pcr, select->pcrSelect);
969                  break;
970              case TPM_PT_PCR_RESET_L2:
971                  if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
972                      PCRSetSelectBit(pcr, select->pcrSelect);
973                  break;
974              case TPM_PT_PCR_EXTEND_L3:
975                  if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
976                      PCRSetSelectBit(pcr, select->pcrSelect);
977                  break;
978              case TPM_PT_PCR_RESET_L3:
979                  if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
980                      PCRSetSelectBit(pcr, select->pcrSelect);
981                  break;
982              case TPM_PT_PCR_EXTEND_L4:
983                  if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
```



```
984                         PCRSetSelectBit(pcr, select->pcrSelect);
985                     break;
986                 case TPM_PT_PCR_RESET_L4:
987                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
988                         PCRSetSelectBit(pcr, select->pcrSelect);
989                     break;
990                 case TPM_PT_PCR_DRTM_RESET:
991                     // DRTM reset PCRs are the PCR reset by locality 4
992                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
993                         PCRSetSelectBit(pcr, select->pcrSelect);
994                     break;
995     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
996                 case TPM_PT_PCR_POLICY:
997                     if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
998                         PCRSetSelectBit(pcr, select->pcrSelect);
999                     break;
1000    #endif
1001    #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
1002                case TPM_PT_PCR_AUTH:
1003                    if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
1004                        PCRSetSelectBit(pcr, select->pcrSelect);
1005                    break;
1006    #endif
1007    #if ENABLE_PCR_NO_INCREMENT == YES
1008                case TPM_PT_PCR_NO_INCREMENT:
1009                    if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
1010                        PCRSetSelectBit(pcr, select->pcrSelect);
1011                    break;
1012    #endif
1013                default:
1014                    // If property is not supported, stop scanning PCR attributes
1015                    // and return.
1016                    return FALSE;
1017                    break;
1018            }
1019        }
1020        return TRUE;
1021    }
```

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

```
1022    TPMI_YES_NO
1023    PCRCapGetProperties(
1024        TPM_PT_PCR                  property,       // IN: the starting PCR property
1025        UINT32                      count,          // IN: count of returned properties
1026        TPML_TAGGED_PCR_PROPERTY    *select         // OUT: PCR select
1027        )
1028    {
1029        TPMI_YES_NO     more = NO;
1030        UINT32          i;
1031
1032        // Initialize output property list
1033        select->count = 0;
1034
1035        // The maximum count of properties we may return is MAX_PCR_PROPERTIES
1036        if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
```

```
1037
1038        // TPM_PT_PCR_FIRST is defined as 0 in spec.  It ensures that property
1039        // value would never be less than TPM_PT_PCR_FIRST
1040        cAssert(TPM_PT_PCR_FIRST == 0);
1041
1042        // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
1043        // implemented on the TPM.
1044        for(i = property; i <= TPM_PT_PCR_LAST; i++)
1045        {
1046            if(select->count < count)
1047            {
1048                // If we have not filled up the return list, add more properties to it
1049                if(PCRGetProperty(i, &select->pcrProperty[select->count]))
1050                    // only increment if the property is implemented
1051                    select->count++;
1052            }
1053            else
1054            {
1055                // If the return list is full but we still have properties
1056                // available, report this and stop iterating.
1057                more = YES;
1058                break;
1059            }
1060        }
1061        return more;
1062    }
```

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

```
1063    TPMI_YES_NO
1064    PCRCapGetHandles(
1065        TPMI_DH_PCR      handle,        // IN: start handle
1066        UINT32           count,         // IN: count of returned handles
1067        TPML_HANDLE     *handleList     // OUT: list of handle
1068        )
1069    {
1070        TPMI_YES_NO     more = NO;
1071        UINT32          i;
1072
1073        pAssert(HandleGetType(handle) == TPM_HT_PCR);
1074
1075        // Initialize output handle list
1076        handleList->count = 0;
1077
1078        // The maximum count of handles we may return is MAX_CAP_HANDLES
1079        if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1080
1081        // Iterate PCR handle range
1082        for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1083        {
1084            if(handleList->count < count)
1085            {
1086                // If we have not filled up the return list, add this PCR
1087                // handle to it
1088                handleList->handle[handleList->count] = i + PCR_FIRST;
```



```
1089                handleList->count++;
1090            }
1091            else
1092            {
1093                // If the return list is full but we still have PCR handle
1094                // available, report this and stop iterating
1095                more = YES;
1096                break;
1097            }
1098        }
1099        return more;
1100    }
```

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

```
1   #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.

```
2   void
3   PhysicalPresencePreInstall_Init(
4       void
5       )
6   {
7       COMMAND_INDEX        commandIndex;
8       // Clear all the PP commands
9       MemorySet(&gp.ppList, 0, sizeof(gp.ppList));
10
11      // Any command that is PP_REQUIRED should be SET
12      for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
13      {
14          if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED
15             &&  s_commandAttributes[commandIndex] & PP_REQUIRED)
16              SET_BIT(commandIndex, gp.ppList);
17      }
18      // Write PP list to NV
19      NV_SYNC_PERSISTENT(ppList);
20      return;
21  }
```

8.8.3.2 PhysicalPresenceCommandSet()

This function is used to set the indicator that a command requires PP confirmation.

```
22  void
23  PhysicalPresenceCommandSet(
24      TPM_CC            commandCode     // IN: command code
25      )
26  {
27      COMMAND_INDEX        commandIndex = CommandCodeToCommandIndex(commandCode);
28
29      // if the command isn't implemented, the do nothing
30      if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
31          return;
32
33      // only set the bit if this is a command for which PP is allowed
34      if(s_commandAttributes[commandIndex] & PP_COMMAND)
```



```
35          SET_BIT(commandIndex, gp.ppList);
36      return;
37  }
```

8.8.3.3 PhysicalPresenceCommandClear()

This function is used to clear the indicator that a command requires PP confirmation.

```
38  void
39  PhysicalPresenceCommandClear(
40      TPM_CC              commandCode     // IN: command code
41      )
42  {
43      COMMAND_INDEX       commandIndex = CommandCodeToCommandIndex(commandCode);
44
45      // If the command isn't implemented, then don't do anything
46      if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
47          return;
48
49      // Only clear the bit if the command does not require PP
50      if((s_commandAttributes[commandIndex] & PP_REQUIRED) == 0)
51          CLEAR_BIT(commandIndex, gp.ppList);
52
53      return;
54  }
```

8.8.3.4 PhysicalPresenceIsRequired()

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

Return Value	Meaning
TRUE(1)	physical presence is required
FALSE(0)	physical presence is not required

```
55  BOOL
56  PhysicalPresenceIsRequired(
57      COMMAND_INDEX     commandIndex    // IN: command index
58      )
59  {
60      // Check the bit map.  If the bit is SET, PP authorization is required
61      return (TEST_BIT(commandIndex, gp.ppList));
62  }
```

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

```
63  TPMI_YES_NO
64  PhysicalPresenceCapGetCCList(
65      TPM_CC              commandCode,    // IN: start command code
66      UINT32              count,          // IN: count of returned TPM_CC
67      TPML_CC            *commandList     // OUT: list of TPM_CC
```

```
68      )
69  {
70      TPMI_YES_NO     more = NO;
71      COMMAND_INDEX   commandIndex;
72
73      // Initialize output handle list
74      commandList->count = 0;
75
76      // The maximum count of command we may return is MAX_CAP_CC
77      if(count > MAX_CAP_CC) count = MAX_CAP_CC;
78
79      // Collect PP commands
80      for(commandIndex = GetClosestCommandIndex(commandCode);
81      commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
82          commandIndex = GetNextCommandIndex(commandIndex))
83      {
84          if(PhysicalPresenceIsRequired(commandIndex))
85          {
86              if(commandList->count < count)
87              {
88                  // If we have not filled up the return list, add this command
89                  // code to it
90                  commandList->commandCodes[commandList->count]
91                      = GetCommandCode(commandIndex);
92                  commandList->count++;
93              }
94              else
95              {
96                  // If the return list is full but we still have PP command
97                  // available, report this and stop iterating
98                  more = YES;
99                  break;
100             }
101         }
102     }
103     return more;
104 }
```


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

```
#define SESSION_C
#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.

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

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

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

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


8.9.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

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

8.9.5 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(1)	session is loaded
FALSE(0)	session is not loaded

```
BOOL
SessionIsLoaded(
    TPM_HANDLE        handle          // IN: session handle
    )
{
    pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION);

    handle = handle & HR_HANDLE_MASK;

    // if out of range of possible active session, or not assigned to a loaded
    // session return false
    if(handle >= MAX_ACTIVE_SESSIONS
       || gr.contextArray[handle] == 0
       || gr.contextArray[handle] > MAX_LOADED_SESSIONS)
        return FALSE;

    return TRUE;
}
```

8.9.5.2 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(1)	session is saved
FALSE(0)	session 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 SequenceNumberForSavedContextIsValid()

This function validates that the sequence number and handle value within a saved context are valid.

```
124 BOOL
125 SequenceNumberForSavedContextIsValid(
126     TPMS_CONTEXT    *context        // IN: pointer to a context structure to be
127                                     //     validated
128     )
129 {
130 #define MAX_CONTEXT_GAP ((UINT64)((CONTEXT_SLOT) ~0) + 1)
131
132     TPM_HANDLE           handle = context->savedHandle & HR_HANDLE_MASK;
133
134     if(// Handle must be with the range of active sessions
135        handle >= MAX_ACTIVE_SESSIONS
136        // the array entry must be for a saved context
137        || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
138            // the array entry must agree with the sequence number
139        || gr.contextArray[handle] != (CONTEXT_SLOT)context->sequence
140            // the provided sequence number has to be less than the current counter
141        || context->sequence > gr.contextCounter
142            // but not so much that it could not be a valid sequence number
143        || gr.contextCounter - context->sequence > MAX_CONTEXT_GAP)
144         return FALSE;
145
146     return TRUE;
147 }
```

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(1)	PCR value is current
FALSE(0)	PCR value is not current

```
148 BOOL
149 SessionPCRValueIsCurrent(
150     SESSION         *session        // IN: session structure
151     )
152 {
153     if(session->pcrCounter != 0
154        && session->pcrCounter != gr.pcrCounter
155        )
156         return FALSE;
157     else
158         return TRUE;
159 }
```

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.

```
160 SESSION *
```



```
161 SessionGet(
162     TPM_HANDLE        handle          // IN: session handle
163     )
164 {
165     size_t           slotIndex;
166     CONTEXT_SLOT     sessionIndex;
167
168     pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
169             || HandleGetType(handle) == TPM_HT_HMAC_SESSION
170             );
171
172     slotIndex = handle & HR_HANDLE_MASK;
173
174     pAssert(slotIndex < MAX_ACTIVE_SESSIONS);
175
176     // get the contents of the session array.  Because session is loaded, we
177     // should always get a valid sessionIndex
178     sessionIndex = gr.contextArray[slotIndex] - 1;
179
180     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
181
182     return &s_sessions[sessionIndex].session;
183 }
```

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.

Error Returns	Meaning
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

```
184 static TPM_RC
185 ContextIdSessionCreate(
186     TPM_HANDLE       *handle,        // OUT: receives the assigned handle. This will
187                                      //     be an index that must be adjusted by the
188                                      //     caller according to the type of the
189                                      //     session created
190     UINT32            sessionIndex   // IN: The session context array entry that will
191                                      //     be occupied by the created session
192     )
193 {
194     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
195
196     // check to see if creating the context is safe
197     // Is this going to be an assignment for the last session context
198     // array entry?  If so, then there will be no room to recycle the
199     // oldest context if needed.  If the gap is not at maximum, then
200     // it will be possible to save a context if it becomes necessary.
201     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
202        && s_freeSessionSlots == 1)
203     {
204         // See if the gap is at maximum
```

```
205         // The current value of the contextCounter will be assigned to the next
206         // saved context. If the value to be assigned would make the same as an
207         // existing context, then we can't use it because of the ambiguity it would
208         // create.
209         if((CONTEXT_SLOT)gr.contextCounter
210            == gr.contextArray[s_oldestSavedSession])
211             return TPM_RC_CONTEXT_GAP;
212     }
213
214     // Find an unoccupied entry in the contextArray
215     for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
216     {
217         if(gr.contextArray[*handle] == 0)
218         {
219             // indicate that the session associated with this handle
220             // references a loaded session
221             gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex + 1);
222             return TPM_RC_SUCCESS;
223         }
224     }
225     return TPM_RC_SESSION_HANDLES;
226 }
```

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

```
227 TPM_RC
228 SessionCreate(
229     TPM_SE            sessionType,    // IN: the session type
230     TPMI_ALG_HASH     authHash,       // IN: the hash algorithm
231     TPM2B_NONCE      *nonceCaller,    // IN: initial nonceCaller
232     TPMT_SYM_DEF     *symmetric,      // IN: the symmetric algorithm
233     TPMI_DH_ENTITY    bind,           // IN: the bind object
234     TPM2B_DATA       *seed,           // IN: seed data
235     TPM_HANDLE       *sessionHandle,  // OUT: the session handle
236     TPM2B_NONCE      *nonceTpm        // OUT: the session nonce
237     )
238 {
239     TPM_RC                result = TPM_RC_SUCCESS;
240     CONTEXT_SLOT          slotIndex;
241     SESSION              *session = NULL;
242
243     pAssert(sessionType == TPM_SE_HMAC
244             || sessionType == TPM_SE_POLICY
245             || sessionType == TPM_SE_TRIAL);
246
247     // If there are no open spots in the session array, then no point in searching
248     if(s_freeSessionSlots == 0)
249         return TPM_RC_SESSION_MEMORY;
250
251     // Find a space for loading a session
252     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
```

```
253     {
254         // Is this available?
255         if(s_sessions[slotIndex].occupied == FALSE)
256         {
257             session = &s_sessions[slotIndex].session;
258             break;
259         }
260     }
261     // if no spot found, then this is an internal error
262     if(slotIndex >= MAX_LOADED_SESSIONS)
263         FAIL(FATAL_ERROR_INTERNAL);
264
265     // Call context ID function to get a handle.  TPM_RC_SESSION_HANDLE may be
266     // returned from ContextIdHandelAssign()
267     result = ContextIdSessionCreate(sessionHandle, slotIndex);
268     if(result != TPM_RC_SUCCESS)
269         return result;
270
271     //*** Only return from this point on is TPM_RC_SUCCESS
272
273     // Can now indicate that the session array entry is occupied.
274     s_freeSessionSlots--;
275     s_sessions[slotIndex].occupied = TRUE;
276
277     // Initialize the session data
278     MemorySet(session, 0, sizeof(SESSION));
279
280     // Initialize internal session data
281     session->authHashAlg = authHash;
282     // Initialize session type
283     if(sessionType == TPM_SE_HMAC)
284     {
285         *sessionHandle += HMAC_SESSION_FIRST;
286     }
287     else
288     {
289         *sessionHandle += POLICY_SESSION_FIRST;
290
291         // For TPM_SE_POLICY or TPM_SE_TRIAL
292         session->attributes.isPolicy = SET;
293         if(sessionType == TPM_SE_TRIAL)
294             session->attributes.isTrialPolicy = SET;
295
296         SessionSetStartTime(session);
297
298         // Initialize policyDigest.  policyDigest is initialized with a string of 0
299         // of session algorithm digest size. Since the session is already clear.
300         // Just need to set the size
301         session->u2.policyDigest.t.size =
302             CryptHashGetDigestSize(session->authHashAlg);
303     }
304     // Create initial session nonce
305     session->nonceTPM.t.size = nonceCaller->t.size;
306     CryptRandomGenerate(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
307     MemoryCopy2B(&nonceTpm->b, &session->nonceTPM.b,
308                  sizeof(nonceTpm->t.buffer));
309
310     // Set up session parameter encryption algorithm
311     session->symmetric = *symmetric;
312
313     // If there is a bind object or a session secret, then need to compute
314     // a sessionKey.
315     if(bind != TPM_RH_NULL || seed->t.size != 0)
316     {
317         // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
318         //                       nonceCaller, bits)
```



```
319         // The HMAC key for generating the sessionSecret can be the concatenation
320         // of an authorization value and a seed value
321         TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
322         TPM2B_KEY            key;
323
324         // Get hash size, which is also the length of sessionKey
325         session->sessionKey.t.size = CryptHashGetDigestSize(session->authHashAlg);
326
327         // Get authValue of associated entity
328         EntityGetAuthValue(bind, (TPM2B_AUTH *)&key);
329         pAssert(key.t.size + seed->t.size <= sizeof(key.t.buffer));
330
331         // Concatenate authValue and seed
332         MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));
333
334         // Compute the session key
335         CryptKDFa(session->authHashAlg, &key.b, SESSION_KEY, &session->nonceTPM.b,
336                   &nonceCaller->b,
337                   session->sessionKey.t.size * 8, session->sessionKey.t.buffer,
338                   NULL, FALSE);
339     }
340
341     // Copy the name of the entity that the HMAC session is bound to
342     // Policy session is not bound to an entity
343     if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
344     {
345         session->attributes.isBound = SET;
346         SessionComputeBoundEntity(bind, &session->u1.boundEntity);
347     }
348     // If there is a bind object and it is subject to DA, then use of this session
349     // is subject to DA regardless of how it is used.
350     session->attributes.isDaBound = (bind != TPM_RH_NULL)
351         && (IsDAExempted(bind) == FALSE);
352
353 // If the session is bound, then check to see if it is bound to lockoutAuth
354     session->attributes.isLockoutBound = (session->attributes.isDaBound == SET)
355         && (bind == TPM_RH_LOCKOUT);
356     return TPM_RC_SUCCESS;
357 }
```

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_CONTEXTSthe() counter maxed out

```
358 TPM_RC
359 SessionContextSave(
360     TPM_HANDLE           handle,        // IN: session handle
361     CONTEXT_COUNTER     *contextID      // OUT: assigned contextID
362     )
363 {
364     UINT32                      contextIndex;
365     CONTEXT_SLOT                slotIndex;
366
367     pAssert(SessionIsLoaded(handle));
```

```
368
369     // check to see if the gap is already maxed out
370     // Need to have a saved session
371     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
372           // if the oldest saved session has the same value as the low bits
373           // of the contextCounter, then the GAP is maxed out.
374        && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
375         return TPM_RC_CONTEXT_GAP;
376
377     // if the caller wants the context counter, set it
378     if(contextID != NULL)
379         *contextID = gr.contextCounter;
380
381     contextIndex = handle & HR_HANDLE_MASK;
382     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
383
384     // Extract the session slot number referenced by the contextArray
385     // because we are going to overwrite this with the low order
386     // contextID value.
387     slotIndex = gr.contextArray[contextIndex] - 1;
388
389     // Set the contextID for the contextArray
390     gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
391
392     // Increment the counter
393     gr.contextCounter++;
394
395     // In the unlikely event that the 64-bit context counter rolls over...
396     if(gr.contextCounter == 0)
397     {
398         // back it up
399         gr.contextCounter--;
400         // return an error
401         return TPM_RC_TOO_MANY_CONTEXTS;
402     }
403     // if the low-order bits wrapped, need to advance the value to skip over
404     // the values used to indicate that a session is loaded
405     if(((CONTEXT_SLOT)gr.contextCounter) == 0)
406         gr.contextCounter += MAX_LOADED_SESSIONS + 1;
407
408     // If no other sessions are saved, this is now the oldest.
409     if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
410         s_oldestSavedSession = contextIndex;
411
412     // Mark the session slot as unoccupied
413     s_sessions[slotIndex].occupied = FALSE;
414
415     // and indicate that there is an additional open slot
416     s_freeSessionSlots++;
417
418     return TPM_RC_SUCCESS;
419 }
```

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

```
420  TPM_RC
421  SessionContextLoad(
422      SESSION_BUF     *session,       // IN: session structure from saved context
423      TPM_HANDLE      *handle         // IN/OUT: session handle
424      )
425  {
426      UINT32              contextIndex;
427      CONTEXT_SLOT        slotIndex;
428
429      pAssert(HandleGetType(*handle) == TPM_HT_POLICY_SESSION
430              || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
431
432      // Don't bother looking if no openings
433      if(s_freeSessionSlots == 0)
434          return TPM_RC_SESSION_MEMORY;
435
436      // Find a free session slot to load the session
437      for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
438          if(s_sessions[slotIndex].occupied == FALSE) break;
439
440      // if no spot found, then this is an internal error
441      pAssert(slotIndex < MAX_LOADED_SESSIONS);
442
443      contextIndex = *handle & HR_HANDLE_MASK;   // extract the index
444
445      // If there is only one slot left, and the gap is at maximum, the only session
446      // context that we can safely load is the oldest one.
447      if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
448         && s_freeSessionSlots == 1
449         && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
450         && contextIndex != s_oldestSavedSession)
451          return TPM_RC_CONTEXT_GAP;
452
453      pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
454
455      // set the contextArray value to point to the session slot where
456      // the context is loaded
457      gr.contextArray[contextIndex] = slotIndex + 1;
458
459      // if this was the oldest context, find the new oldest
460      if(contextIndex == s_oldestSavedSession)
461          ContextIdSetOldest();
462
463      // Copy session data to session slot
464      MemoryCopy(&s_sessions[slotIndex].session, session, sizeof(SESSION));
465
466             // Set session slot as occupied
467      s_sessions[slotIndex].occupied = TRUE;
468
469      // Reduce the number of open spots
470      s_freeSessionSlots--;
471
472      return TPM_RC_SUCCESS;
473  }
```


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.

```
474 void
475 SessionFlush(
476     TPM_HANDLE       handle         // IN: loaded or saved session handle
477     )
478 {
479     CONTEXT_SLOT         slotIndex;
480     UINT32               contextIndex;   // Index into contextArray
481
482     pAssert((HandleGetType(handle) == TPM_HT_POLICY_SESSION
483              || HandleGetType(handle) == TPM_HT_HMAC_SESSION
484              )
485             && (SessionIsLoaded(handle) || SessionIsSaved(handle))
486             );
487
488          // Flush context ID of this session
489          // Convert handle to an index into the contextArray
490     contextIndex = handle & HR_HANDLE_MASK;
491
492     pAssert(contextIndex < sizeof(gr.contextArray) / sizeof(gr.contextArray[0]));
493
494     // Get the current contents of the array
495     slotIndex = gr.contextArray[contextIndex];
496
497     // Mark context array entry as available
498     gr.contextArray[contextIndex] = 0;
499
500     // Is this a saved session being flushed
501     if(slotIndex > MAX_LOADED_SESSIONS)
502     {
503         // Flushing the oldest session?
504         if(contextIndex == s_oldestSavedSession)
505             // If so, find a new value for oldest.
506             ContextIdSetOldest();
507     }
508     else
509     {
510         // Adjust slot index to point to session array index
511         slotIndex -= 1;
512
513         // Free session array index
514         s_sessions[slotIndex].occupied = FALSE;
515         s_freeSessionSlots++;
516     }
517
518     return;
519 }
```

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.

```
520 void
521 SessionComputeBoundEntity(
```



```
522         TPMI_DH_ENTITY       entityHandle,  // IN: handle of entity
523         TPM2B_NAME          *bind           // OUT: binding value
524         )
525     {
526         TPM2B_AUTH           auth;
527         BYTE                *pAuth = auth.t.buffer;
528         UINT16               i;
529
530         // Get name
531         EntityGetName(entityHandle, bind);
532
533     //    // The bound value of a reserved handle is the handle itself
534     //    if(bind->t.size == sizeof(TPM_HANDLE)) return;
535
536         // For all the other entities, concatenate the authorization value to the name.
537         // Get a local copy of the authorization value because some overlapping
538         // may be necessary.
539         EntityGetAuthValue(entityHandle, &auth);
540
541         // Make sure that the extra space is zeroed
542         MemorySet(&bind->t.name[bind->t.size], 0, sizeof(bind->t.name) - bind->t.size);
543         // XOR the authValue at the end of the name
544         for(i = sizeof(bind->t.name) - auth.t.size; i < sizeof(bind->t.name); i++)
545             bind->t.name[i] ^= *pAuth++;
546
547         // Set the bind value to the maximum size
548         bind->t.size = sizeof(bind->t.name);
549
550         return;
551     }
```

8.9.6.7 SessionSetStartTime()

This function is used to initialize the session timing

```
552     void
553     SessionSetStartTime(
554         SESSION          *session        // IN: the session to update
555         )
556     {
557         session->startTime = g_time;
558         session->epoch = g_timeEpoch;
559         session->timeout = 0;
560     }
```

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.

```
561     void
562     SessionResetPolicyData(
563         SESSION          *session        // IN: the session to reset
564         )
565     {
566         SESSION_ATTRIBUTES      oldAttributes;
567         pAssert(session != NULL);
568
569         // Will need later
570         oldAttributes = session->attributes;
571
572         // No command
573         session->commandCode = 0;
574
```

```
575         // No locality selected
576         MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
577
578         // The cpHash size to zero
579         session->u1.cpHash.b.size = 0;
580
581         // No timeout
582         session->timeout = 0;
583
584         // Reset the pcrCounter
585         session->pcrCounter = 0;
586
587         // Reset the policy hash
588         MemorySet(&session->u2.policyDigest.t.buffer, 0,
589                   session->u2.policyDigest.t.size);
590
591         // Reset the session attributes
592         MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
593
594         // Restore the policy attributes
595         session->attributes.isPolicy = SET;
596         session->attributes.isTrialPolicy = oldAttributes.isTrialPolicy;
597
598         // Restore the bind attributes
599         session->attributes.isDaBound = oldAttributes.isDaBound;
600         session->attributes.isLockoutBound = oldAttributes.isLockoutBound;
601     }
```

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

```
602     TPMI_YES_NO
603     SessionCapGetLoaded(
604         TPMI_SH_POLICY   handle,         // IN: start handle
605         UINT32           count,          // IN: count of returned handles
606         TPML_HANDLE     *handleList      // OUT: list of handle
607         )
608     {
609         TPMI_YES_NO     more = NO;
610         UINT32          i;
611
612         pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
613
614         // Initialize output handle list
615         handleList->count = 0;
616
617         // The maximum count of handles we may return is MAX_CAP_HANDLES
618         if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
619
620         // Iterate session context ID slots to get loaded session handles
621         for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
622         {
623             // If session is active
624             if(gr.contextArray[i] != 0)
625             {
626                 // If session is loaded
```

```
627                 if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
628                 {
629                     if(handleList->count < count)
630                     {
631                         SESSION         *session;
632
633                         // If we have not filled up the return list, add this
634                         // session handle to it
635                         // assume that this is going to be an HMAC session
636                         handle = i + HMAC_SESSION_FIRST;
637                         session = SessionGet(handle);
638                         if(session->attributes.isPolicy)
639                             handle = i + POLICY_SESSION_FIRST;
640                         handleList->handle[handleList->count] = handle;
641                         handleList->count++;
642                     }
643                     else
644                     {
645                         // If the return list is full but we still have loaded object
646                         // available, report this and stop iterating
647                         more = YES;
648                         break;
649                     }
650                 }
651             }
652         }
653
654         return more;
655     }
```

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

```
656 TPMI_YES_NO
657 SessionCapGetSaved(
658     TPMI_SH_HMAC     handle,        // IN: start handle
659     UINT32           count,         // IN: count of returned handles
660     TPML_HANDLE     *handleList     // OUT: list of handle
661     )
662 {
663     TPMI_YES_NO     more = NO;
664     UINT32          i;
665
666 #ifdef  TPM_HT_SAVED_SESSION
667     pAssert(HandleGetType(handle) == TPM_HT_SAVED_SESSION);
668 #else
669     pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);
670 #endif
671
672     // Initialize output handle list
673     handleList->count = 0;
674
675     // The maximum count of handles we may return is MAX_CAP_HANDLES
676     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
677
678     // Iterate session context ID slots to get loaded session handles
```

```
679     for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
680     {
681         // If session is active
682         if(gr.contextArray[i] != 0)
683         {
684             // If session is saved
685             if(gr.contextArray[i] > MAX_LOADED_SESSIONS)
686             {
687                 if(handleList->count < count)
688                 {
689                     // If we have not filled up the return list, add this
690                     // session handle to it
691                     handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
692                     handleList->count++;
693                 }
694                 else
695                 {
696                     // If the return list is full but we still have loaded object
697                     // available, report this and stop iterating
698                     more = YES;
699                     break;
700                 }
701             }
702         }
703     }
704
705     return more;
706 }
```

8.9.6.11 SessionCapGetLoadedNumber()

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

```
707 UINT32
708 SessionCapGetLoadedNumber(
709     void
710     )
711 {
712     return MAX_LOADED_SESSIONS - s_freeSessionSlots;
713 }
```

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.

```
714 UINT32
715 SessionCapGetLoadedAvail(
716     void
717     )
718 {
719     return s_freeSessionSlots;
720 }
```

8.9.6.13 SessionCapGetActiveNumber()

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


```
UINT32
SessionCapGetActiveNumber(
    void
    )
{
    UINT32              i;
    UINT32              num = 0;

    // Iterate the context array to find the number of non-zero slots
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        if(gr.contextArray[i] != 0) num++;
    }

    return num;
}
```

8.9.6.14 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.

```
UINT32
SessionCapGetActiveAvail(
    void
    )
{
    UINT32              i;
    UINT32              num = 0;

    // Iterate the context array to find the number of zero slots
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        if(gr.contextArray[i] == 0) num++;
    }

    return num;
}
```

8.10 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
    )
{
    g_time = _plat__TimerRead();
}
```

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
    )
{
#if CLOCK_STOPS
    CryptRandomGenerate(sizeof(CLOCK_NONCE), (BYTE *)&g_timeEpoch);
#else
    // if the epoch is kept in NV, update it.
    gp.timeEpoch++;
    NV_SYNC_PERSISTENT(timeEpoch);
#endif
    // Clean out any lingering state
    _plat__TimerWasStopped();
}
```

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.

```
25  void
26  TimeStartup(
27      STARTUP_TYPE     type            // IN: start up type
28      )
29  {
30      NOT_REFERENCED(type);
31      // If the previous cycle is orderly shut down, the value of the safe bit
32      // the same as previously saved.  Otherwise, it is not safe.
33      if(!NV_IS_ORDERLY)
34          go.clockSafe = NO;
35      return;
36  }
```

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.

```
37  void
38  TimeClockUpdate(
39      UINT64           newTime     // IN: New time value in mS.
40      )
41  {
42  #define CLOCK_UPDATE_MASK   ((1ULL << NV_CLOCK_UPDATE_INTERVAL)- 1)
43
44      // Check to see if the update will cause a need for an nvClock update
45      if((newTime | CLOCK_UPDATE_MASK) > (go.clock | CLOCK_UPDATE_MASK))
46      {
47          pAssert(g_NvStatus == TPM_RC_SUCCESS);
48
49          // Going to update the NV time state so SET the safe flag
50          go.clockSafe = YES;
51
52          // update the time
53          go.clock = newTime;
54
55          NvWrite(NV_ORDERLY_DATA, sizeof(go), &go);
56      }
57      else
58          // No NV update needed so just update
59          go.clock = newTime;
60
61  }
```

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.

```
62  void
63  TimeUpdate(
```

```
64      void
65      )
66  {
67      UINT64          elapsed;
68  //
69      // Make sure that we consume the current _plat__TimerWasStopped() state.
70     if(_plat__TimerWasStopped())
71      {
72          TimeNewEpoch();
73      }
74      // Get the difference between this call and the last time we updated the tick
75      // timer.
76      elapsed = _plat__TimerRead() - g_time;
77      // Don't read +
78      g_time += elapsed;
79
80      // Don't need to check the result because it has to be success because have
81      // already checked that NV is available.
82      TimeClockUpdate(go.clock + elapsed);
83
84      // Call self healing logic for dictionary attack parameters
85      DASelfHeal();
86  }
```

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.

```
87  void
88  TimeUpdateToCurrent(
89      void
90  )
91  {
92      // Can't update time during the dark interval or when rate limiting so don't
93      // make any modifications to the internal clock value. Also, defer any clock
94      // processing until TPM has run TPM2_Startup()
95      if(!NV_IS_AVAILABLE || !TPMIsStarted())
96          return;
97
98      TimeUpdate();
99  }
```

8.10.3.7 TimeSetAdjustRate()

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

```
100 void
101 TimeSetAdjustRate(
102     TPM_CLOCK_ADJUST     adjust          // IN: adjust constant
103     )
104 {
105     switch(adjust)
106     {
```



```
107         case TPM_CLOCK_COARSE_SLOWER:
108             _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
109             break;
110         case TPM_CLOCK_COARSE_FASTER:
111             _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
112             break;
113         case TPM_CLOCK_MEDIUM_SLOWER:
114             _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
115             break;
116         case TPM_CLOCK_MEDIUM_FASTER:
117             _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
118             break;
119         case TPM_CLOCK_FINE_SLOWER:
120             _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
121             break;
122         case TPM_CLOCK_FINE_FASTER:
123             _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
124             break;
125         case TPM_CLOCK_NO_CHANGE:
126             break;
127         default:
128             FAIL(FATAL_ERROR_INTERNAL);
129             break;
130     }
131
132     return;
133 }
```

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

```
134 UINT16
135 TimeGetMarshaled(
136     TIME_INFO       *dataBuffer     // OUT: result buffer
137     )
138 {
139     TPMS_TIME_INFO      timeInfo;
140
141     // Fill TPMS_TIME_INFO structure
142     timeInfo.time = g_time;
143     TimeFillInfo(&timeInfo.clockInfo);
144
145     // Marshal TPMS_TIME_INFO to canonical form
146     return TPMS_TIME_INFO_Marshal(&timeInfo, (BYTE **)&dataBuffer, NULL);
147 }
```

8.10.3.9 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```
148 void
149 TimeFillInfo(
150     TPMS_CLOCK_INFO     *clockInfo
151     )
152 {
153     clockInfo->clock = go.clock;
154     clockInfo->resetCount = gp.resetCount;
155     clockInfo->restartCount = gr.restartCount;
156
157     // If NV is not available, clock stopped advancing and the value reported is
158     // not "safe".
```

```
159     if(NV_IS_AVAILABLE)
160         clockInfo->safe = go.clockSafe;
161     else
162         clockInfo->safe = NO;
163
164     return;
165 }
```

9 Support

9.1 AlgorithmCap.c

9.1.1 Description

This file contains the algorithm property definitions for the algorithms and the code for the TPM2_GetCapability() to return the algorithm properties.

9.1.2 Includes and Defines

```
1   #include "Tpm.h"
2   typedef struct
3   {
4       TPM_ALG_ID          algID;
5       TPMA_ALGORITHM      attributes;
6   } ALGORITHM;
7   static const ALGORITHM    s_algorithms[] =
8   {
9   // The entries in this table need to be in ascending order but the table doesn't
10  // need to be full (gaps are allowed). One day, a tool might exist to fill in the
11  // table from the TPM_ALG description
12  #if ALG_RSA
13      {TPM_ALG_RSA,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
14  #endif
15  #if ALG_TDES
16      {TPM_ALG_TDES,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
17  #endif
18  #if ALG_SHA1
19      {TPM_ALG_SHA1,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
20  #endif
21
22      {TPM_ALG_HMAC,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 1, 0, 0, 0)},
23
24  #if ALG_AES
25      {TPM_ALG_AES,           TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
26  #endif
27  #if ALG_MGF1
28      {TPM_ALG_MGF1,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
29  #endif
30
31      {TPM_ALG_KEYEDHASH,     TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 1, 0, 1, 1, 0, 0)},
32
33  #if ALG_XOR
34      {TPM_ALG_XOR,           TPMA_ALGORITHM_INITIALIZER(0, 1, 1, 0, 0, 0, 0, 0, 0)},
35  #endif
36
37  #if ALG_SHA256
38      {TPM_ALG_SHA256,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
39  #endif
40  #if ALG_SHA384
41      {TPM_ALG_SHA384,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
42  #endif
43  #if ALG_SHA512
44      {TPM_ALG_SHA512,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
45  #endif
46  #if ALG_SM3_256
47      {TPM_ALG_SM3_256,       TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
48  #endif
49  #if ALG_SM4
50      {TPM_ALG_SM4,           TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
51  #endif
```

```
52  #if ALG_RSASSA
53      {TPM_ALG_RSASSA,        TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
54  #endif
55  #if ALG_RSAES
56      {TPM_ALG_RSAES,         TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
57  #endif
58  #if ALG_RSAPSS
59      {TPM_ALG_RSAPSS,        TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
60  #endif
61  #if ALG_OAEP
62      {TPM_ALG_OAEP,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
63  #endif
64  #if ALG_ECDSA
65      {TPM_ALG_ECDSA,         TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 1, 0)},
66  #endif
67  #if ALG_ECDH
68      {TPM_ALG_ECDH,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
69  #endif
70  #if ALG_ECDAA
71      {TPM_ALG_ECDAA,         TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
72  #endif
73  #if ALG_SM2
74      {TPM_ALG_SM2,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 1, 0)},
75  #endif
76  #if ALG_ECSCHNORR
77      {TPM_ALG_ECSCHNORR,      TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
78  #endif
79  #if ALG_ECMQV
80      {TPM_ALG_ECMQV,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
81  #endif
82  #if ALG_KDF1_SP800_56A
83      {TPM_ALG_KDF1_SP800_56A, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
84  #endif
85  #if ALG_KDF2
86      {TPM_ALG_KDF2,           TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
87  #endif
88  #if ALG_KDF1_SP800_108
89      {TPM_ALG_KDF1_SP800_108, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
90  #endif
91  #if ALG_ECC
92      {TPM_ALG_ECC,            TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
93  #endif
94
95      {TPM_ALG_SYMCIPHER,      TPMA_ALGORITHM_INITIALIZER(0, 0, 0, 1, 0, 0, 0, 0, 0)},
96
97  #if ALG_CAMELLIA
98      {TPM_ALG_CAMELLIA,       TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
99  #endif
100 #if ALG_CMAC
101     {TPM_ALG_CMAC,           TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 1, 0, 0, 0)},
102 #endif
103 #if ALG_CTR
104     {TPM_ALG_CTR,            TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
105 #endif
106 #if ALG_OFB
107     {TPM_ALG_OFB,            TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
108 #endif
109 #if ALG_CBC
110     {TPM_ALG_CBC,            TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
111 #endif
112 #if ALG_CFB
113     {TPM_ALG_CFB,            TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
114 #endif
115 #if ALG_ECB
116     {TPM_ALG_ECB,            TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
117 #endif
```



```
};
```

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

```
TPMI_YES_NO
AlgorithmCapGetImplemented(
    TPM_ALG_ID                  algID,      // IN: the starting algorithm ID
    UINT32                      count,      // IN: count of returned algorithms
    TPML_ALG_PROPERTY          *algList     // OUT: algorithm list
    )
{
    TPMI_YES_NO     more = NO;
    UINT32          i;
    UINT32          algNum;

    // initialize output algorithm list
    algList->count = 0;

    // The maximum count of algorithms we may return is MAX_CAP_ALGS.
    if(count > MAX_CAP_ALGS)
        count = MAX_CAP_ALGS;

    // Compute how many algorithms are defined in s_algorithms array.
    algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);

    // Scan the implemented algorithm list to see if there is a match to 'algID'.
    for(i = 0; i < algNum; i++)
    {
        // If algID is less than the starting algorithm ID, skip it
        if(s_algorithms[i].algID < algID)
            continue;
        if(algList->count < count)
        {
            // If we have not filled up the return list, add more algorithms
            // to it
            algList->algProperties[algList->count].alg = s_algorithms[i].algID;
            algList->algProperties[algList->count].algProperties =
                s_algorithms[i].attributes;
            algList->count++;
        }
        else
        {
            // If the return list is full but we still have algorithms
            // available, report this and stop scanning.
            more = YES;
            break;
        }
    }

    return more;
}
```

9.1.4 AlgorithmGetImplementedVector()

This function returns the bit vector of the implemented algorithms.

```
166 LIB_EXPORT
167 void
168 AlgorithmGetImplementedVector(
169     ALGORITHM_VECTOR    *implemented    // OUT: the implemented bits are SET
170     )
171 {
172     int                     index;
173
174     // Nothing implemented until we say it is
175     MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));
176
177     for(index = (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
178     index >= 0;
179         index--)
180         SET_BIT(s_algorithms[index].algID, *implemented);
181     return;
182 }
```

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(1)	bit is set
FALSE(0)	bit is not set

```
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);
}
```

9.2.3.2 SetBit()

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

```
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));
}
```

9.2.3.3 ClearBit()

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

```
void
```



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

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.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of the command

```
#if !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

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of a command

```
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
33      // The commandCode is a UINT32 and the search index is UINT16. We are going to
34      // search for a match but need to make sure that the commandCode value is not
35      // out of range. To do this, need to clear the vendor bit of the commandCode
36      // (if set) and compare the result to the 16-bit searchIndex value. If it is
37      // out of range, indicate that the command is not implemented
38      if((commandCode & ~CC_VEND) != searchIndex)
39          return UNIMPLEMENTED_COMMAND_INDEX;
40
41      // if there is at least one vendor command, the last entry in the array will
42      // have the v bit set. If the input commandCode is larger than the last
43      // vendor-command, then it is out of range.
44      if(vendor)
45      {
46  #if VENDOR_COMMAND_ARRAY_SIZE > 0
47          COMMAND_INDEX       commandIndex;
48          COMMAND_INDEX       min;
49          COMMAND_INDEX       max;
50          int                 diff;
51  #if LIBRARY_COMMAND_ARRAY_SIZE == COMMAND_COUNT
52  #error "Constants are not consistent."
53  #endif
54          // Check to see if the value is equal to or below the minimum
55          // entry.
56          // Note: Put this check first so that the typical case of only one vendor-
57          // specific command doesn't waste any more time.
58          if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE], TPMA_CC,
59                           commandIndex) >= searchIndex)
60          {
61              // the vendor array is always assumed to be packed so there is
62              // no need to check to see if the command is implemented
63              return LIBRARY_COMMAND_ARRAY_SIZE;
64          }
65          // See if this is out of range on the top
66          if(GET_ATTRIBUTE(s_ccAttr[COMMAND_COUNT - 1], TPMA_CC, commandIndex)
67             < searchIndex)
68          {
69              return UNIMPLEMENTED_COMMAND_INDEX;
70          }
71          commandIndex = UNIMPLEMENTED_COMMAND_INDEX; // Needs initialization to keep
72                                                      // compiler happy
73          min = LIBRARY_COMMAND_ARRAY_SIZE;       // first vendor command
74          max = COMMAND_COUNT - 1;                // last vendor command
75          diff = 1;                               // needs initialization to keep
76                                                  // compiler happy
77          while(min <= max)
78          {
```



```
79              commandIndex = (min + max + 1) / 2;
80              diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
81                  - searchIndex;
82              if(diff == 0)
83                  return commandIndex;
84              if(diff > 0)
85                  max = commandIndex - 1;
86              else
87                  min = commandIndex + 1;
88          }
89          // didn't find and exact match. commandIndex will be pointing at the last
90          // item tested. If 'diff' is positive, then the last item tested was
91          // larger index of the command code so it is the smallest value
92          // larger than the requested value.
93          if(diff > 0)
94              return commandIndex;
95          // if 'diff' is negative, then the value tested was smaller than
96          // the commandCode index and the next higher value is the correct one.
97          // Note: this will necessarily be in range because of the earlier check
98          // that the index was within range.
99          return commandIndex + 1;
100 #else
101     // If there are no vendor commands so anything with the vendor bit set is out
102     // of range
103         return UNIMPLEMENTED_COMMAND_INDEX;
104 #endif
105     }
106     // Get here if the V-Bit was not set in 'commandCode'
107
108     if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE - 1], TPMA_CC,
109                      commandIndex) < searchIndex)
110     {
111         // requested index is out of the range to the top
112 #if VENDOR_COMMAND_ARRAY_SIZE > 0
113         // If there are vendor commands, then the first vendor command
114         // is the next value greater than the commandCode.
115         // NOTE: we got here if the starting index did not have the V bit but we
116         // reached the end of the array of library commands (non-vendor). Since
117         // there is at least one vendor command, and vendor commands are always
118         // in a compressed list that starts after the library list, the next
119         // index value contains a valid vendor command.
120         return LIBRARY_COMMAND_ARRAY_SIZE;
121 #else
122         // if there are no vendor commands, then this is out of range
123         return UNIMPLEMENTED_COMMAND_INDEX;
124 #endif
125     }
126     // If the request is lower than any value in the array, then return
127     // the lowest value (needs to be an index for an implemented command
128     if(GET_ATTRIBUTE(s_ccAttr[0], TPMA_CC, commandIndex) >= searchIndex)
129     {
130         return NextImplementedIndex(0);
131     }
132     else
133     {
134 #if COMPRESSED_LISTS
135         COMMAND_INDEX       commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
136         COMMAND_INDEX       min = 0;
137         COMMAND_INDEX       max = LIBRARY_COMMAND_ARRAY_SIZE - 1;
138         int                 diff = 1;
139 #if LIBRARY_COMMAND_ARRAY_SIZE == 0
140 #error  "Something is terribly wrong"
141 #endif
142         // The s_ccAttr array contains an extra entry at the end (a zero value).
143         // Don't count this as an array entry. This means that max should start
144         // out pointing to the last valid entry in the array which is - 2
```

```
145         pAssert(max == (sizeof(s_ccAttr) / sizeof(TPMA_CC)
146                         - VENDOR_COMMAND_ARRAY_SIZE - 2));
147         while(min <= max)
148         {
149             commandIndex = (min + max + 1) / 2;
150             diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC,
151                                  commandIndex) - searchIndex;
152             if(diff == 0)
153                 return commandIndex;
154             if(diff > 0)
155                 max = commandIndex - 1;
156             else
157                 min = commandIndex + 1;
158         }
159         // didn't find and exact match. commandIndex will be pointing at the
160         // last item tested. If diff is positive, then the last item tested was
161         // larger index of the command code so it is the smallest value
162         // larger than the requested value.
163         if(diff > 0)
164             return commandIndex;
165         // if diff is negative, then the value tested was smaller than
166         // the commandCode index and the next higher value is the correct one.
167         // Note: this will necessarily be in range because of the earlier check
168         // that the index was within range.
169         return commandIndex + 1;
170 #else
171         // The list is not compressed so offset into the array by the command
172         // code value of the first entry in the list. Then go find the first
173         // implemented command.
174         return NextImplementedIndex(searchIndex
175                                     - (COMMAND_INDEX)s_ccAttr[0].commandIndex);
176 #endif
177     }
178 }
```

9.3.3.3 CommandCodeToComandIndex()

This function returns the index in the various attributes arrays of the command.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of the command

```
179 COMMAND_INDEX
180 CommandCodeToCommandIndex(
181     TPM_CC            commandCode    // IN: the command code to look up
182     )
183 {
184     // Extract the low 16-bits of the command code to get the starting search index
185     COMMAND_INDEX       searchIndex = (COMMAND_INDEX)commandCode;
186     BOOL                vendor = (commandCode & CC_VEND) != 0;
187     COMMAND_INDEX       commandIndex;
188 #if !COMPRESSED_LISTS
189     if(!vendor)
190     {
191         commandIndex = searchIndex - (COMMAND_INDEX)s_ccAttr[0].commandIndex;
192         // Check for out of range or unimplemented.
193         // Note, since a COMMAND_INDEX is unsigned, if searchIndex is smaller than
194         // the lowest value of command, it will become a 'negative' number making
195         // it look like a large unsigned number, this will cause it to fail
196         // the unsigned check below.
197         if(commandIndex >= LIBRARY_COMMAND_ARRAY_SIZE
```



```
198             || (s_commandAttributes[commandIndex] & IS_IMPLEMENTED) == 0)
199              return UNIMPLEMENTED_COMMAND_INDEX;
200         return commandIndex;
201     }
202 #endif
203     // Need this code for any vendor code lookup or for compressed lists
204     commandIndex = GetClosestCommandIndex(commandCode);
205
206     // Look at the returned value from get closest. If it isn't the one that was
207     // requested, then the command is not implemented.
208     if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
209     {
210         if((GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
211             != searchIndex)
212            || (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V)) != vendor)
213             commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
214     }
215     return commandIndex;
216 }
```

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

```
217 COMMAND_INDEX
218 GetNextCommandIndex(
219     COMMAND_INDEX    commandIndex   // IN: the starting index
220     )
221 {
222     while(++commandIndex < COMMAND_COUNT)
223     {
224 #if !COMPRESSED_LISTS
225         if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
226 #endif
227             return commandIndex;
228     }
229     return UNIMPLEMENTED_COMMAND_INDEX;
230 }
```

9.3.3.5 GetCommandCode()

This function returns the *commandCode* associated with the command index

```
231 TPM_CC
232 GetCommandCode(
233     COMMAND_INDEX    commandIndex   // IN: the command index
234     )
235 {
236     TPM_CC           commandCode = GET_ATTRIBUTE(s_ccAttr[commandIndex],
237                                                  TPMA_CC, commandIndex);
238     if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
239         commandCode += CC_VEND;
240     return commandCode;
241 }
```


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

```
242 AUTH_ROLE
243 CommandAuthRole(
244     COMMAND_INDEX    commandIndex,  // IN: command index
245     UINT32           handleIndex    // IN: handle index (zero based)
246     )
247 {
248     if(0 == handleIndex)
249     {
250         // Any authorization role set?
251         COMMAND_ATTRIBUTES  properties = s_commandAttributes[commandIndex];
252
253         if(properties & HANDLE_1_USER)
254             return AUTH_USER;
255         if(properties & HANDLE_1_ADMIN)
256             return AUTH_ADMIN;
257         if(properties & HANDLE_1_DUP)
258             return AUTH_DUP;
259     }
260     else if(1 == handleIndex)
261     {
262         if(s_commandAttributes[commandIndex] & HANDLE_2_USER)
263             return AUTH_USER;
264     }
265     return AUTH_NONE;
266 }
```

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

```
267 int
268 EncryptSize(
269     COMMAND_INDEX    commandIndex   // IN: command index
270     )
271 {
272     return ((s_commandAttributes[commandIndex] & ENCRYPT_2) ? 2 :
273             (s_commandAttributes[commandIndex] & ENCRYPT_4) ? 4 : 0);
274 }
```


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

```
275 int
276 DecryptSize(
277     COMMAND_INDEX    commandIndex    // IN: command index
278     )
279 {
280     return ((s_commandAttributes[commandIndex] & DECRYPT_2) ? 2 :
281             (s_commandAttributes[commandIndex] & DECRYPT_4) ? 4 : 0);
282 }
```

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(1)	session is allowed with this command
FALSE(0)	session is not allowed with this command

```
283 BOOL
284 IsSessionAllowed(
285     COMMAND_INDEX    commandIndex    // IN: the command to be checked
286     )
287 {
288     return ((s_commandAttributes[commandIndex] & NO_SESSIONS) == 0);
289 }
```

9.3.3.10 IsHandleInResponse()

This function determines if a command has a handle in the response

```
290 BOOL
291 IsHandleInResponse(
292     COMMAND_INDEX    commandIndex
293     )
294 {
295     return ((s_commandAttributes[commandIndex] & R_HANDLE) != 0);
296 }
```

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

```
297 BOOL
298 IsWriteOperation(
299     COMMAND_INDEX    commandIndex    // IN: Command to check
300     )
```



```
301 {
302 #ifdef  WRITE_LOCK
303     return ((s_commandAttributes[commandIndex] & WRITE_LOCK) != 0);
304 #else
305     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
306     {
307         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
308         {
309             case TPM_CC_NV_Write:
310 #if CC_NV_Increment
311             case TPM_CC_NV_Increment:
312 #endif
313 #if CC_NV_SetBits
314             case TPM_CC_NV_SetBits:
315 #endif
316 #if CC_NV_Extend
317             case TPM_CC_NV_Extend:
318 #endif
319 #if CC_AC_Send
320             case TPM_CC_AC_Send:
321 #endif
322             // NV write lock counts as a write operation for authorization purposes.
323             // We check to see if the NV is write locked before we do the
324             // authorization. If it is locked, we fail the command early.
325             case TPM_CC_NV_WriteLock:
326                 return TRUE;
327             default:
328                 break;
329         }
330     }
331     return FALSE;
332 #endif
333 }
```

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.

```
334 BOOL
335 IsReadOperation(
336     COMMAND_INDEX    commandIndex   // IN: Command to check
337     )
338 {
339 #ifdef  READ_LOCK
340     return ((s_commandAttributes[commandIndex] & READ_LOCK) != 0);
341 #else
342
343     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
344     {
345         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
346         {
347             case TPM_CC_NV_Read:
348             case TPM_CC_PolicyNV:
349             case TPM_CC_NV_Certify:
350             // NV read lock counts as a read operation for authorization purposes.
351             // We check to see if the NV is read locked before we do the
352             // authorization. If it is locked, we fail the command early.
353             case TPM_CC_NV_ReadLock:
354                 return TRUE;
355             default:
356                 break;
357         }
358     }
359     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;

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

    for(commandIndex = GetClosestCommandIndex(commandCode);
    commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
        commandIndex = GetNextCommandIndex(commandIndex))
    {
#if !COMPRESSED_LISTS
        // this check isn't needed for compressed lists.
        if(!(s_commandAttributes[commandIndex] & IS_IMPLEMENTED))
            continue;
#endif
        if(commandList->count < count)
        {
            // If the list is not full, add the attributes for this command.
            commandList->commandAttributes[commandList->count]
                = s_ccAttr[commandIndex];
            commandList->count++;
        }
        else
        {
            // If the list is full but there are more commands to report,
            // indicate this and return.
            more = YES;
            break;
        }
    }
    return more;
}
```

9.3.3.14 IsVendorCommand()

Function indicates if a command index references a vendor command.

Return Value	Meaning
TRUE(1)	command is a vendor command
FALSE(0)	command is not a vendor command

```
BOOL
IsVendorCommand(
    COMMAND_INDEX    commandIndex   // IN: command index to check
    )
{
    return (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V));
}
```

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;
```

```
36                          // null handle, PW session handle and lockout
37                          // handle are always available
38                      case TPM_RH_NULL:
39                      case TPM_RS_PW:
40                          // Need to be careful for lockout. Lockout is always available
41                          // for policy checks but not always available when authValue
42                          // is being checked.
43                      case TPM_RH_LOCKOUT:
44                          break;
45                      default:
46                          // handling of the manufacture_specific handles
47                          if(((TPM_RH)handle >= TPM_RH_AUTH_00)
48                             && ((TPM_RH)handle <= TPM_RH_AUTH_FF))
49                             // use the value that would have been returned from
50                             // unmarshaling if it did the handle filtering
51                              result = TPM_RC_VALUE;
52                          else
53                              FAIL(FATAL_ERROR_INTERNAL);
54                          break;
55                  }
56                  break;
57              case TPM_HT_TRANSIENT:
58                  // For a transient object, check if the handle is associated
59                  // with a loaded object.
60                  if(!IsObjectPresent(handle))
61                      result = TPM_RC_REFERENCE_H0;
62                  break;
63              case TPM_HT_PERSISTENT:
64                  // Persistent object
65                  // Copy the persistent object to RAM and replace the handle with the
66                  // handle of the assigned slot.  A TPM_RC_OBJECT_MEMORY,
67                  // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
68                  // ObjectLoadEvict()
69                  result = ObjectLoadEvict(&command->handles[i], command->index);
70                  break;
71              case TPM_HT_HMAC_SESSION:
72                  // For an HMAC session, see if the session is loaded
73                  // and if the session in the session slot is actually
74                  // an HMAC session.
75                  if(SessionIsLoaded(handle))
76                  {
77                      SESSION             *session;
78                      session = SessionGet(handle);
79                      // Check if the session is a HMAC session
80                      if(session->attributes.isPolicy == SET)
81                          result = TPM_RC_HANDLE;
82                  }
83                  else
84                      result = TPM_RC_REFERENCE_H0;
85                  break;
86              case TPM_HT_POLICY_SESSION:
87                  // For a policy session, see if the session is loaded
88                  // and if the session in the session slot is actually
89                  // a policy session.
90                  if(SessionIsLoaded(handle))
91                  {
92                      SESSION             *session;
93                      session = SessionGet(handle);
94                      // Check if the session is a policy session
95                      if(session->attributes.isPolicy == CLEAR)
96                          result = TPM_RC_HANDLE;
97                  }
98                  else
99                      result = TPM_RC_REFERENCE_H0;
100                 break;
101             case TPM_HT_NV_INDEX:
```

```
102                     // For an NV Index, use the TPM-specific routine
103                     // to search the IN Index space.
104                     result = NvIndexIsAccessible(handle);
105                     break;
106                 case TPM_HT_PCR:
107                     // Any PCR handle that is unmarshaled successfully referenced
108                     // a PCR that is defined.
109                     break;
110 #if CC_AC_Send
111                 case TPM_HT_AC:
112                     // Use the TPM-specific routine to search for the AC
113                     result = AcIsAccessible(handle);
114                     break;
115 #endif
116                 default:
117                     // Any other handle type is a defect in the unmarshaling code.
118                     FAIL(FATAL_ERROR_INTERNAL);
119                     break;
120             }
121             if(result != TPM_RC_SUCCESS)
122             {
123                 if(result == TPM_RC_REFERENCE_H0)
124                     result = result + i;
125                 else
126                     result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
127                 break;
128             }
129         }
130         return result;
131     }
```

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

```
132 UINT16
133 EntityGetAuthValue(
134     TPMI_DH_ENTITY   handle,        // IN: handle of entity
135     TPM2B_AUTH      *auth           // OUT: authValue of the entity
136     )
137 {
138     TPM2B_AUTH      *pAuth = NULL;
139
140     auth->t.size = 0;
141
142     switch(HandleGetType(handle))
143     {
144         case TPM_HT_PERMANENT:
145         {
146             switch(handle)
147             {
148                 case TPM_RH_OWNER:
149                     // ownerAuth for TPM_RH_OWNER
150                     pAuth = &gp.ownerAuth;
151                     break;
```

```
152                     case TPM_RH_ENDORSEMENT:
153                         // endorsementAuth for TPM_RH_ENDORSEMENT
154                         pAuth = &gp.endorsementAuth;
155                         break;
156                     case TPM_RH_PLATFORM:
157                         // platformAuth for TPM_RH_PLATFORM
158                         pAuth = &gc.platformAuth;
159                         break;
160                     case TPM_RH_LOCKOUT:
161                         // lockoutAuth for TPM_RH_LOCKOUT
162                         pAuth = &gp.lockoutAuth;
163                         break;
164                     case TPM_RH_NULL:
165                         // nullAuth for TPM_RH_NULL. Return 0 directly here
166                         return 0;
167                         break;
168 #ifdef  VENDOR_PERMANENT
169                     case VENDOR_PERMANENT:
170                         // vendor authorization value
171                         pAauth = &g_platformUniqueDetails;
172 #endif
173                     default:
174                         // If any other permanent handle is present it is
175                         // a code defect.
176                         FAIL(FATAL_ERROR_INTERNAL);
177                         break;
178                 }
179                 break;
180             }
181             case TPM_HT_TRANSIENT:
182                 // authValue for an object
183                 // A persistent object would have been copied into RAM
184                 // and would have an transient object handle here.
185             {
186                 OBJECT          *object;
187
188                 object = HandleToObject(handle);
189                 // special handling if this is a sequence object
190                 if(ObjectIsSequence(object))
191                 {
192                     pAuth = &((HASH_OBJECT *)object)->auth;
193                 }
194                 else
195                 {
196                     // Authorization is available only when the private portion of
197                     // the object is loaded.  The check should be made before
198                     // this function is called
199                     pAssert(object->attributes.publicOnly == CLEAR);
200                     pAuth = &object->sensitive.authValue;
201                 }
202             }
203             break;
204             case TPM_HT_NV_INDEX:
205                 // authValue for an NV index
206             {
207                 NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
208                 pAssert(nvIndex != NULL);
209                 pAuth = &nvIndex->authValue;
210             }
211             break;
212             case TPM_HT_PCR:
213                 // authValue for PCR
214                 pAuth = PCRGetAuthValue(handle);
215                 break;
216             default:
217                 // If any other handle type is present here, then there is a defect
```



```
218                 // in the unmarshaling code.
219                 FAIL(FATAL_ERROR_INTERNAL);
220                 break;
221         }
222         // Copy the authValue
223         MemoryCopy2B(&auth->b, &pAuth->b, sizeof(auth->t.buffer));
224         MemoryRemoveTrailingZeros(auth);
225         return auth->t.size;
226     }
```

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.

```
227     TPMI_ALG_HASH
228     EntityGetAuthPolicy(
229         TPMI_DH_ENTITY   handle,        // IN: handle of entity
230         TPM2B_DIGEST    *authPolicy     // OUT: authPolicy of the entity
231         )
232     {
233         TPMI_ALG_HASH       hashAlg = TPM_ALG_NULL;
234         authPolicy->t.size = 0;
235
236         switch(HandleGetType(handle))
237         {
238             case TPM_HT_PERMANENT:
239                 switch(handle)
240                 {
241                     case TPM_RH_OWNER:
242                         // ownerPolicy for TPM_RH_OWNER
243                         *authPolicy = gp.ownerPolicy;
244                         hashAlg = gp.ownerAlg;
245                         break;
246                     case TPM_RH_ENDORSEMENT:
247                         // endorsementPolicy for TPM_RH_ENDORSEMENT
248                         *authPolicy = gp.endorsementPolicy;
249                         hashAlg = gp.endorsementAlg;
250                         break;
251                     case TPM_RH_PLATFORM:
252                         // platformPolicy for TPM_RH_PLATFORM
253                         *authPolicy = gc.platformPolicy;
254                         hashAlg = gc.platformAlg;
255                         break;
256                     case TPM_RH_LOCKOUT:
257                         // lockoutPolicy for TPM_RH_LOCKOUT
258                         *authPolicy = gp.lockoutPolicy;
259                         hashAlg = gp.lockoutAlg;
260                         break;
261                     default:
262                         return TPM_ALG_ERROR;
263                         break;
264                 }
265                 break;
266             case TPM_HT_TRANSIENT:
267                 // authPolicy for an object
268             {
269                 OBJECT *object = HandleToObject(handle);
```

```
270                 *authPolicy = object->publicArea.authPolicy;
271                 hashAlg = object->publicArea.nameAlg;
272             }
273             break;
274             case TPM_HT_NV_INDEX:
275                 // authPolicy for a NV index
276             {
277                 NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
278                 pAssert(nvIndex != 0);
279                 *authPolicy = nvIndex->publicArea.authPolicy;
280                 hashAlg = nvIndex->publicArea.nameAlg;
281             }
282             break;
283             case TPM_HT_PCR:
284                 // authPolicy for a PCR
285                 hashAlg = PCRGetAuthPolicy(handle, authPolicy);
286                 break;
287             default:
288                 // If any other handle type is present it is a code defect.
289                 FAIL(FATAL_ERROR_INTERNAL);
290                 break;
291         }
292         return hashAlg;
293     }
```

9.4.3.4 EntityGetName()

This function returns the Name associated with a handle.

```
294     TPM2B_NAME *
295     EntityGetName(
296         TPMI_DH_ENTITY   handle,        // IN: handle of entity
297         TPM2B_NAME      *name           // OUT: name of entity
298         )
299     {
300         switch(HandleGetType(handle))
301         {
302             case TPM_HT_TRANSIENT:
303             {
304                 // Name for an object
305                 OBJECT      *object = HandleToObject(handle);
306                 // an object with no nameAlg has no name
307                 if(object->publicArea.nameAlg == TPM_ALG_NULL)
308                     name->b.size = 0;
309                 else
310                     *name = object->name;
311                 break;
312             }
313             case TPM_HT_NV_INDEX:
314                 // Name for a NV index
315                 NvGetNameByIndexHandle(handle, name);
316                 break;
317             default:
318                 // For all other types, the handle is the Name
319                 name->t.size = sizeof(TPM_HANDLE);
320                 UINT32_TO_BYTE_ARRAY(handle, name->t.name);
321                 break;
322         }
323         return name;
324     }
```


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.

```
TPMI_RH_HIERARCHY
EntityGetHierarchy(
    TPMI_DH_ENTITY   handle         // IN :handle of entity
    )
{
    TPMI_RH_HIERARCHY       hierarchy = TPM_RH_NULL;

    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            // hierarchy for a permanent handle
            switch(handle)
            {
                case TPM_RH_PLATFORM:
                case TPM_RH_ENDORSEMENT:
                case TPM_RH_NULL:
                    hierarchy = handle;
                    break;
                // all other permanent handles are associated with the owner
                // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
                default:
                    hierarchy = TPM_RH_OWNER;
                    break;
            }
            break;
        case TPM_HT_NV_INDEX:
            // hierarchy for NV index
        {
            NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
            pAssert(nvIndex != NULL);

            // If only the platform can delete the index, then it is
            // considered to be in the platform hierarchy, otherwise it
            // is in the owner hierarchy.
            if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV,
                            PLATFORMCREATE))
                hierarchy = TPM_RH_PLATFORM;
            else
                hierarchy = TPM_RH_OWNER;
        }
        break;
        case TPM_HT_TRANSIENT:
            // hierarchy for an object
        {
            OBJECT          *object;
            object = HandleToObject(handle);
            if(object->attributes.ppsHierarchy)
            {
                hierarchy = TPM_RH_PLATFORM;
            }
            else if(object->attributes.epsHierarchy)
            {
                hierarchy = TPM_RH_ENDORSEMENT;
            }
            else if(object->attributes.spsHierarchy)
```

```
380                 {
381                     hierarchy = TPM_RH_OWNER;
382                 }
383             }
384             break;
385             case TPM_HT_PCR:
386                 hierarchy = TPM_RH_OWNER;
387                 break;
388             default:
389                 FAIL(FATAL_ERROR_INTERNAL);
390                 break;
391         }
392         // this is unreachable but it provides a return value for the default
393         // case which makes the complier happy
394         return hierarchy;
395     }
```

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 are in Global.h.

9.5.2 Includes and Defines

```
1   #define GLOBAL_C
2   #include "Tpm.h"
```

9.5.3 __IGNORE_STATE__

This macro value is used to delimit values that are only used within the scope of a single TPM command. For a virtualized TPM, this would be state that does not need to be preserved between commands. This is not a lot of data and it may not be necessary to exclude this data.

9.5.4 Global Data Values

These values are visible across multiple modules.

```
3   BOOL                    g_phEnable;
4   const UINT16            g_rcIndex[15] = {TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,
5                                            TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,
6                                            TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,
7                                            TPM_RC_D, TPM_RC_E, TPM_RC_F
8   };
9   TPM_HANDLE              g_exclusiveAuditSession;
10  UINT64                  g_time;
11  #if CLOCK_STOPS
12  CLOCK_NONCE             g_timeEpoch;
13  #endif
14  BOOL                    g_pcrReConfig;
15  TPMI_DH_OBJECT          g_DRTMHandle;
16  BOOL                    g_DrtmPreStartup;
17  BOOL                    g_StartupLocality3;
18  #if USE_DA_USED
19  BOOL                    g_daUsed;
20  #endif
21  BOOL                    g_powerWasLost;
22  BOOL                    g_clearOrderly;
23  TPM_SU                  g_prevOrderlyState;
24  UPDATE_TYPE             g_updateNV;
25  BOOL                    g_nvOk;
26  TPM_RC                  g_NvStatus;
27  TPM2B_AUTH              g_platformUniqueDetails;
28  ALGORITHM_VECTOR        g_implementedAlgorithms;
29  ALGORITHM_VECTOR        g_toTest;
30  CRYPTO_SELF_TEST_STATE    g_cryptoSelfTestState;    // This structure contains the
31                                                      // cryptographic self-test
32  #if SIMULATION
33  BOOL                    g_forceFailureMode;
34  #endif
35  BOOL                    g_inFailureMode;
36  STATE_CLEAR_DATA        gc;
37  STATE_RESET_DATA        gr;
38  PERSISTENT_DATA         gp;
39  ORDERLY_DATA            go;
```

9.5.5 Private Values

9.5.5.1 Used in SessionProcess.c

```
#ifndef __IGNORE_STATE__          // DO NOT DEFINE THIS VALUE
```

These values do not need to be retained between commands.

```
TPM_HANDLE            s_sessionHandles[MAX_SESSION_NUM];
TPMA_SESSION          s_attributes[MAX_SESSION_NUM];
TPM_HANDLE            s_associatedHandles[MAX_SESSION_NUM];
TPM2B_NONCE           s_nonceCaller[MAX_SESSION_NUM];
TPM2B_AUTH            s_inputAuthValues[MAX_SESSION_NUM];
SESSION              *s_usedSessions[MAX_SESSION_NUM];
UINT32                s_encryptSessionIndex;
UINT32                s_decryptSessionIndex;
UINT32                s_auditSessionIndex;
#endif  // __IGNORE_STATE__
BOOL                  s_DAPendingOnNV;
#if CC_GetCommandAuditDigest
TPM2B_DIGEST          s_cpHashForCommandAudit;
#endif
```

9.5.5.2 Used in DA.c

```
#if !ACCUMULATE_SELF_HEAL_TIMER
UINT64                s_selfHealTimer;
UINT64                s_lockoutTimer;
#endif // !ACCUMULATE_SELF_HEAL_TIMER
```

9.5.5.3 Used in NV.c

```
UINT64                s_maxCounter;
NV_REF                s_evictNvEnd;
BYTE                  s_indexOrderlyRam[RAM_INDEX_SPACE];
#ifndef __IGNORE_STATE__          // DO NOT DEFINE THIS VALUE
NV_INDEX              s_cachedNvIndex;
NV_REF                s_cachedNvRef;
BYTE                 *s_cachedNvRamRef;
#endif // __IGNORE_STATE__
```

9.5.5.4 Used in Object.c

```
OBJECT               s_objects[MAX_LOADED_OBJECTS];
```

9.5.5.5 Used in PCR.c

```
PCR                   s_pcrs[IMPLEMENTATION_PCR];
```

9.5.5.6 Used in Session.c

```
SESSION_SLOT          s_sessions[MAX_LOADED_SESSIONS];
UINT32                s_oldestSavedSession;
int                   s_freeSessionSlots;
```

9.5.5.7 Used in MemoryLib.c

```
#ifndef __IGNORE_STATE__          // DO NOT DEFINE THIS VALUE
```



```
UINT64    s_actionIoBuffer[768];      // action I/O buffer
UINT32    s_actionIoAllocation;       // number of UIN64 allocated for in
#endif
```

9.5.5.8 used in TpmFail.c

```
UINT32              s_failFunction;
UINT32              s_failLine;
UINT32              s_failCode;
```

9.5.5.9 Used in CryptRand.c

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.

```
RAND_STATE          *s_random;
```

9.5.5.10 Used in 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.

```
BOOL                g_manufactured = FALSE;
```

9.5.5.11 Used in 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.

```
BOOL                g_initialized;
```

9.5.5.12 Purpose-specific String Constants

These string constants are shared across functions to make sure that they are all using consistent sting values.

```
TPM2B_STRING(PRIMARY_OBJECT_CREATION, "Primary Object Creation");
TPM2B_STRING(CFB_KEY, "CFB");
TPM2B_STRING(CONTEXT_KEY, "CONTEXT");
TPM2B_STRING(INTEGRITY_KEY, "INTEGRITY");
TPM2B_STRING(SECRET_KEY, "SECRET");
TPM2B_STRING(SESSION_KEY, "ATH");
TPM2B_STRING(STORAGE_KEY, "STORAGE");
TPM2B_STRING(XOR_KEY, "XOR");
TPM2B_STRING(COMMIT_STRING, "ECDAA Commit");
TPM2B_STRING(DUPLICATE_STRING, "DUPLICATE");
TPM2B_STRING(IDENTITY_STRING, "IDENTITY");
TPM2B_STRING(OBFUSCATE_STRING, "OBFUSCATE");
#if SELF_TEST
TPM2B_STRING(OAEP_TEST_STRING, "OAEP Test Value");
#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:

```
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;
                break;
            default:
                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
54      pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
55
56      // Initialize output handle list
57      handleList->count = 0;
58
59      // The maximum count of handles we may return is MAX_CAP_HANDLES
60      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
61
62      // Iterate permanent handle range
63      for(i = NextPermanentHandle(handle);
64      i != 0; i = NextPermanentHandle(i + 1))
65      {
66          if(handleList->count < count)
67          {
68              // If we have not filled up the return list, add this permanent
69              // handle to it
70              handleList->handle[handleList->count] = i;
71              handleList->count++;
72          }
73          else
74          {
75              // If the return list is full but we still have permanent handle
76              // available, report this and stop iterating
77              more = YES;
78              break;
79          }
80      }
81      return more;
82  }
```

9.6.3.4 PermanentHandleGetPolicy()

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

```
83  TPMI_YES_NO
84  PermanentHandleGetPolicy(
85      TPM_HANDLE           handle,        // IN: start handle
86      UINT32               count,         // IN: max count of returned handles
87      TPML_TAGGED_POLICY  *policyList     // OUT: list of handle
88      )
89  {
90      TPMI_YES_NO     more = NO;
91
92      pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
93
94      // Initialize output handle list
95      policyList->count = 0;
96
97      // The maximum count of policies we may return is MAX_TAGGED_POLICIES
98      if(count > MAX_TAGGED_POLICIES)
99          count = MAX_TAGGED_POLICIES;
100
101     // Iterate permanent handle range
102     for(handle = NextPermanentHandle(handle);
103         handle != 0;
104         handle = NextPermanentHandle(handle + 1))
105     {
106         TPM2B_DIGEST    policyDigest;
107         TPM_ALG_ID      policyAlg;
108         // Check to see if this permanent handle has a policy
109         policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
110         if(policyAlg == TPM_ALG_ERROR)
111            continue;
112         if(policyList->count < count)
113         {
114             // If we have not filled up the return list, add this
115             // policy to the list;
116             policyList->policies[policyList->count].handle = handle;
117             policyList->policies[policyList->count].policyHash.hashAlg = policyAlg;
118             MemoryCopy(&policyList->policies[policyList->count].policyHash.digest,
119                        policyDigest.t.buffer, policyDigest.t.size);
120             policyList->count++;
121         }
122         else
123         {
124             // If the return list is full but we still have permanent handle
125             // available, report this and stop iterating
126             more = YES;
127             break;
128         }
129     }
130     return more;
131 }
```


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.

```
1  #define IO_BUFFER_C
2  #include "Tpm.h"
3  #include "IoBuffers_fp.h"
```

9.7.2 Buffers and Functions

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.

9.7.2.1 MemoryIoBufferAllocationReset()

This function is used to reset the allocation of buffers.

```
4  void
5  MemoryIoBufferAllocationReset(
6      void
7  )
8  {
9      s_actionIoAllocation = 0;
10 }
```

9.7.2.2 MemoryIoBufferZero()

Function zeros the action I/O buffer at the end of a command. Calling this is not mandatory for proper functionality.

```
11 void
12 MemoryIoBufferZero(
13     void
14 )
15 {
16     memset(s_actionIoBuffer, 0, s_actionIoAllocation);
17 }
```

9.7.2.3 MemoryGetInBuffer()

This function returns the address of the buffer into which the command parameters will be unmarshaled in preparation for calling the command actions.

```
18 BYTE *
19 MemoryGetInBuffer(
20     UINT32          size            // Size, in bytes, required for the input
21                                     // unmarshaling
22     )
23 {
24     pAssert(size <= sizeof(s_actionIoBuffer));
25     // In this implementation, a static buffer is set aside for the command action
26     // buffers. The buffer is shared between input and output. This is because
27     // there is no need to allocate for the worst case input and worst case output
```

```
28      // at the same time.
29      // Round size up
30      #define UoM  (sizeof(s_actionIoBuffer[0]))
31      size = (size + (UoM - 1)) & (UINT32_MAX - (UoM - 1));
32      memset(s_actionIoBuffer, 0, size);
33      s_actionIoAllocation = size;
34      return (BYTE *)&s_actionIoBuffer[0];
35  }
```

9.7.2.4 MemoryGetOutBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```
36  BYTE *
37  MemoryGetOutBuffer(
38      UINT32          size            // required size of the buffer
39      )
40  {
41      BYTE        *retVal = (BYTE *)(&s_actionIoBuffer[s_actionIoAllocation / UoM]);
42      pAssert((size + s_actionIoAllocation) < (sizeof(s_actionIoBuffer)));
43      // In this implementation, a static buffer is set aside for the command action
44      // output buffer.
45      memset(retVal, 0, size);
46      s_actionIoAllocation += size;
47      return retVal;
48  }
```

9.7.2.5 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
TRUE(1)	string is null terminated
FALSE(0)	string is not null terminated

```
49  BOOL
50  IsLabelProperlyFormatted(
51      TPM2B           *x
52      )
53  {
54      return (((x)->size == 0) || ((x)->buffer[(x)->size - 1] == 0));
55  }
```


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:
            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ZERO);
            break;
        case 1:
            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ONE);
            break;
        case 2:
            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_TWO);
            break;
        case 3:
            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_THREE);
            break;
        case 4:
            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_FOUR);
            break;
        default:
            pAssert(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

```
1  #define MANUFACTURE_C
2  #include "Tpm.h"
3  #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

```
4  LIB_EXPORT int
5  TPM_Manufacture(
6      int             firstTime       // IN: indicates if this is the first call from
7                                      //     main()
8      )
9  {
10     TPM_SU          orderlyShutdown;
11
12 #if RUNTIME_SIZE_CHECKS
13     // Call the function to verify the sizes of values that result from different
14     // compile options.
15     TpmSizeChecks();
16 #endif
17
18     // If TPM has been manufactured, return indication.
19     if(!firstTime && g_manufactured)
20         return 1;
21
22     // Do power on initializations of the cryptographic libraries.
23     CryptInit();
24
25     s_DAPendingOnNV = FALSE;
26
27     // initialize NV
28     NvManufacture();
29
30     // Clear the magic value in the DRBG state
31     go.drbgState.magic = 0;
32
33     CryptStartup(SU_RESET);
34
35     // default configuration for PCR
36     PCRSimStart();
```



```
37
38      // initialize pre-installed hierarchy data
39      // This should happen after NV is initialized because hierarchy data is
40      // stored in NV.
41      HierarchyPreInstall_Init();
42
43      // initialize dictionary attack parameters
44      DAPreInstall_Init();
45
46      // initialize PP list
47      PhysicalPresencePreInstall_Init();
48
49      // initialize command audit list
50      CommandAuditPreInstall_Init();
51
52      // first start up is required to be Startup(CLEAR)
53      orderlyShutdown = TPM_SU_CLEAR;
54      NV_WRITE_PERSISTENT(orderlyState, orderlyShutdown);
55
56      // initialize the firmware version
57      gp.firmwareV1 = FIRMWARE_V1;
58  #ifdef FIRMWARE_V2
59      gp.firmwareV2 = FIRMWARE_V2;
60  #else
61      gp.firmwareV2 = 0;
62  #endif
63      NV_SYNC_PERSISTENT(firmwareV1);
64      NV_SYNC_PERSISTENT(firmwareV2);
65
66      // initialize the total reset counter to 0
67      gp.totalResetCount = 0;
68      NV_SYNC_PERSISTENT(totalResetCount);
69
70      // initialize the clock stuff
71      go.clock = 0;
72      go.clockSafe = YES;
73
74      NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
75
76      // Commit NV writes.  Manufacture process is an artificial process existing
77      // only in simulator environment and it is not defined in the specification
78      // that what should be the expected behavior if the NV write fails at this
79      // point.  Therefore, it is assumed the NV write here is always success and
80      // no return code of this function is checked.
81      NvCommit();
82
83      g_manufactured = TRUE;
84
85      return 0;
86  }
```

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

```
LIB_EXPORT int
TPM_TearDown(
    void
    )
{
    g_manufactured = FALSE;
    return 0;
}
```

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.

```
LIB_EXPORT void
TpmEndSimulation(
    void
    )
{
#if SIMULATION
    HashLibSimulationEnd();
    SymLibSimulationEnd();
    MathLibSimulationEnd();
#if     ALG_RSA
    RsaSimulationEnd();
#endif
#if     ALG_ECC
    EccSimulationEnd();
#endif
#endif // SIMULATION
}
```

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:

```
TPM_RC
TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
                         BOOL flag)
{
    TPM_RC    result;
    result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    if(*target == TPM_RH_NULL)
    {
        if(flag)
            return TPM_RC_SUCCESS;
        else
            return TPM_RC_VALUE;
    }
    if(((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
      &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
            return TPM_RC_VALUE;
    return TPM_RC_SUCCESS;
}
```

and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```
UINT16
TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
```

```
{
    return UINT32_Marshal((UINT32 *)source, buffer, size);
}
```

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
    }
```



```
27      return TPM_RC_SELECTOR;
28  }
```

NOTE The `#ifdef/#endif` directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

```
1   UINT16
2   TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
3                             UINT32 selector)
4   {
5       switch(selector) {
6   #ifdef TPM_ALG_KEYEDHASH
7           case TPM_ALG_KEYEDHASH:
8               return TPMS_KEYEDHASH_PARMS_Marshal(
9                           (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
10  #endif
11  #ifdef TPM_ALG_SYMCIPHER
12          case TPM_ALG_SYMCIPHER:
13              return TPMT_SYM_DEF_OBJECT_Marshal(
14                            (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
15  #endif
16  #ifdef TPM_ALG_RSA
17          case TPM_ALG_RSA:
18              return TPMS_RSA_PARMS_Marshal(
19                             (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
20  #endif
21  #ifdef TPM_ALG_ECC
22          case TPM_ALG_ECC:
23              return TPMS_ECC_PARMS_Marshal(
24                               (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
25  #endif
26      }
27      assert(1);
28      return 0;
29  }
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

9.10.4 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:

```
TPM_RC
TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
{
    TPM_RC    result;
    result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
                                       buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
                                     buffer, size, flag);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
                                   buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
                                    buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
                                         buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;

    result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
                                      buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```


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

```
TPM_RC
TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    // if size equal to 0, the rest of the structure is a zero buffer
    //  so stop processing
    if(target->t.size == 0)
        return TPM_RC_SUCCESS;
    if((target->t.size) > 1024)    // This check is triggered by the {:1024}
                                   // notation on 'buffer'
        return TPM_RC_SIZE;
    result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
                                  (INT32)(target->t.size));
    if(result != TPM_RC_SUCCESS)
        return result;
    return TPM_RC_SUCCESS;
}
```

using these structure definitions:

```
typedef union {
    struct {
        UINT16        size;
        BYTE          buffer[1024];
    }            t;
    TPM2B        b;
} 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.

```
#include "Tpm.h"
```

9.11.2 Functions

9.11.2.1 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)

```
LIB_EXPORT int
UnsignedCompareB(
    UINT32            aSize,          // IN: size of a
    const BYTE       *a,              // IN: a
    UINT32            bSize,          // IN: size of b
    const BYTE       *b               // IN: b
    )
{
    UINT32             i;
    if(aSize > bSize)
        return 1;
    else if(aSize < bSize)
        return -1;
    else
    {
        for(i = 0; i < aSize; i++)
        {
            if(a[i] != b[i])
                return (a[i] > b[i]) ? 1 : -1;
        }
    }
    return 0;
}
```

9.11.2.2 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
35      // For positive or 0, sign_a is 1
36      // for negative, sign_a is 0
37      signA = ((a[0] & 0x80) == 0) ? 1 : 0;
38
39      // For positive or 0, sign_b is 1
40      // for negative, sign_b is 0
41      signB = ((b[0] & 0x80) == 0) ? 1 : 0;
42
43      if(signA != signB)
44      {
45          return signA - signB;
46      }
47      if(signA == 1)
48          // do unsigned compare function
49          return UnsignedCompareB(aSize, a, bSize, b);
50      else
51          // do unsigned compare the other way
52          return 0 - UnsignedCompareB(aSize, a, bSize, b);
53  }
```

9.11.2.3 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

```
54  TPM_RC
55  ModExpB(
56      UINT32           cSize,          // IN: the size of the output buffer. It will
57                                       //     need to be the same size as the modulus
58      BYTE            *c,              // OUT: the buffer to receive the results
59                                       //     (c->size must be set to the maximum size
60                                       //     for the returned value)
61      const UINT32     mSize,
62      const BYTE      *m,              // IN: number to exponentiate
```

```
63      const UINT32     eSize,
64      const BYTE      *e,             // IN: power
65      const UINT32     nSize,
66      const BYTE      *n              // IN: modulus
67      )
68  {
69      BN_MAX(bnC);
70      BN_MAX(bnM);
71      BN_MAX(bnE);
72      BN_MAX(bnN);
73      NUMBYTES         tSize = (NUMBYTES)nSize;
74      TPM_RC           retVal = TPM_RC_SUCCESS;
75
76      // Convert input parameters
77      BnFromBytes(bnM, m, (NUMBYTES)mSize);
78      BnFromBytes(bnE, e, (NUMBYTES)eSize);
79      BnFromBytes(bnN, n, (NUMBYTES)nSize);
80
81      // Make sure that the output is big enough to hold the result
82      // and that 'm' is less than 'n' (the modulus)
83      if(cSize < nSize)
84          ERROR_RETURN(TPM_RC_NO_RESULT);
85      if(BnUnsignedCmp(bnM, bnN) >= 0)
86          ERROR_RETURN(TPM_RC_SIZE);
87      BnModExp(bnC, bnM, bnE, bnN);
88      BnToBytes(bnC, c, &tSize);
89  Exit:
90      return retVal;
91  }
```

9.11.2.4 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_NO_RESULT	*q* or *r* is too small to receive the result

```
92  LIB_EXPORT TPM_RC
93  DivideB(
94      const TPM2B      *n,             // IN: numerator
95      const TPM2B      *d,             // IN: denominator
96      TPM2B            *q,             // OUT: quotient
97      TPM2B            *r              // OUT: remainder
98      )
99  {
100     BN_MAX_INITIALIZED(bnN, n);
101     BN_MAX_INITIALIZED(bnD, d);
102     BN_MAX(bnQ);
103     BN_MAX(bnR);
104 //
105     // Do divide with converted values
106     BnDiv(bnQ, bnR, bnN, bnD);
107
108     // Convert the BIGNUM result back to 2B format using the size of the original
109     // number
110     if(q != NULL)
111         if(!BnTo2B(bnQ, q, q->size))
112             return TPM_RC_NO_RESULT;
113     if(r != NULL)
114         if(!BnTo2B(bnR, r, r->size))
115             return TPM_RC_NO_RESULT;
116     return TPM_RC_SUCCESS;
```



```
117 }
```

9.11.2.5 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.

```
118 UINT16
119 AdjustNumberB(
120     TPM2B           *num,
121     UINT16           requestedSize
122     )
123 {
124     BYTE            *from;
125     UINT16           i;
126     // See if number is already the requested size
127     if(num->size == requestedSize)
128         return requestedSize;
129     from = num->buffer;
130     if (num->size > requestedSize)
131     {
132     // This is a request to shift the number to the left (remove leading zeros)
133         // Find the first non-zero byte. Don't look past the point where removing
134         // more zeros would make the number smaller than requested, and don't throw
135         // away any significant digits.
136         for(i = num->size; *from == 0 && i > requestedSize; from++, i--);
137         if(i < num->size)
138         {
139             num->size = i;
140             MemoryCopy(num->buffer, from, i);
141         }
142     }
143     // This is a request to shift the number to the right (add leading zeros)
144     else
145     {
146         MemoryCopy(&num->buffer[requestedSize - num->size], num->buffer, num->size);
147         MemorySet(num->buffer, 0, requestedSize- num->size);
148         num->size = requestedSize;
149     }
150     return num->size;
151 }
```

9.11.2.6 ShiftLeft()

This function shifts a byte buffer (a TPM2B) one byte to the left. That is, the most significant bit of the most significant byte is lost.

```
152 TPM2B *
153 ShiftLeft(
154     TPM2B       *value          // IN/OUT: value to shift and shifted value out
155 )
156 {
157     UINT16       count = value->size;
158     BYTE        *buffer = value->buffer;
159     if(count > 0)
160     {
161         for(count -= 1; count > 0; buffer++, count--)
162         {
163             buffer[0] = (buffer[0] << 1) + ((buffer[1] & 0x80) ? 1 : 0);
164         }
165         *buffer <<= 1;
```



```
166         }
167         return value;
168     }
```

9.11.2.7 IsNumeric()

Verifies that all the characters are simple numeric (0-9)

```
169     BOOL
170     IsNumeric(
171         TPM2B       *value
172     )
173     {
174         UINT16      i;
175         for(i = 0; i < value->size; i++)
176         {
177             if(value->buffer[i] < '0' || value->buffer[i] > '9')
178                 return FALSE;
179         }
180         return TRUE;
181     }
```

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

```
void
MemoryCopy(
    void        *dest,
    const void  *src,
    int          sSize
    )
{
    memmove(dest, src, sSize);
}
```

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(1)	all octets are the same
FALSE(0)	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
    // due to timing differences.
    for(; size > 0; size--)
```

```
26          equal |= (*b1++ ^ *b2++);
27      return (equal == 0);
28  }
```

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.

```
29  LIB_EXPORT INT16
30  MemoryCopy2B(
31      TPM2B           *dest,          // OUT: receiving TPM2B
32      const TPM2B     *source,        // IN: source TPM2B
33      unsigned int     dSize          // IN: size of the receiving buffer
34      )
35  {
36      pAssert(dest != NULL);
37      if(source == NULL)
38          dest->size = 0;
39      else
40      {
41          pAssert(source->size <= dSize);
42          MemoryCopy(dest->buffer, source->buffer, source->size);
43          dest->size = source->size;
44      }
45      return dest->size;
46  }
```

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))*.

```
47  void
48  MemoryConcat2B(
49      TPM2B           *aInOut,        // IN/OUT: destination 2B
50      TPM2B           *bIn,           // IN: second 2B
51      unsigned int     aMaxSize       // IN: The size of aInOut.buffer (max values for
52                                      //     aInOut.size)
53      )
54  {
55      pAssert(bIn->size <= aMaxSize - aInOut->size);
56      MemoryCopy(&aInOut->buffer[aInOut->size], &bIn->buffer, bIn->size);
57      aInOut->size = aInOut->size + bIn->size;
58      return;
59  }
```

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(1)	size and buffer contents are the same
FALSE(0)	size or buffer contents are not the same

```
60  BOOL
61  MemoryEqual2B(
```



```
62      const TPM2B     *aIn,           // IN: compare value
63      const TPM2B     *bIn            // IN: compare value
64      )
65  {
66      if(aIn->size != bIn->size)
67          return FALSE;
68      return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
69  }
```

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.

```
70  void
71  MemorySet(
72      void            *dest,
73      int              value,
74      size_t           size
75      )
76  {
77      memset(dest, value, size);
78  }
```

9.12.3.7 MemoryPad2B()

Function to pad a TPM2B with zeros and adjust the size.

```
79  void
80  MemoryPad2B(
81      TPM2B           *b,
82      UINT16           newSize
83      )
84  {
85      MemorySet(&b->buffer[b->size], 0, newSize - b->size);
86      b->size = newSize;
87  }
```

9.12.3.8 Uint16ToByteArray()

Function to write an integer to a byte array

```
88  void
89  Uint16ToByteArray(
90      UINT16              i,
91      BYTE                *a
92      )
93  {
94      a[1] = (BYTE)(i); i >>= 8;
95      a[0] = (BYTE)(i);
96  }
```

9.12.3.9 Uint32ToByteArray()

Function to write an integer to a byte array

```
97  void
```



```
98   Uint32ToByteArray(
99       UINT32              i,
100      BYTE                *a
101      )
102  {
103      a[3] = (BYTE)(i); i >>= 8;
104      a[2] = (BYTE)(i); i >>= 8;
105      a[1] = (BYTE)(i); i >>= 8;
106      a[0] = (BYTE)(i);
107  }
```

9.12.3.10 Uint64ToByteArray()

Function to write an integer to a byte array

```
108  void
109  Uint64ToByteArray(
110      UINT64               i,
111      BYTE                *a
112      )
113  {
114      a[7] = (BYTE)(i); i >>= 8;
115      a[6] = (BYTE)(i); i >>= 8;
116      a[5] = (BYTE)(i); i >>= 8;
117      a[4] = (BYTE)(i); i >>= 8;
118      a[3] = (BYTE)(i); i >>= 8;
119      a[2] = (BYTE)(i); i >>= 8;
120      a[1] = (BYTE)(i); i >>= 8;
121      a[0] = (BYTE)(i);
122  }
```

9.12.3.11 ByteArrayToUint16()

Function to write an integer to a byte array

```
123  UINT16
124  ByteArrayToUint16(
125      BYTE                *a
126      )
127  {
128      UINT16      retVal;
129      retVal  = a[0]; retVal <<= 8;
130      retVal += a[1];
131      return retVal;
132  }
```

9.12.3.12 ByteArrayToUint32()

Function to write an integer to a byte array

```
133  UINT32
134  ByteArrayToUint32(
135      BYTE                *a
136      )
137  {
138      UINT32      retVal;
139      retVal  = a[0]; retVal <<= 8;
140      retVal += a[1]; retVal <<= 8;
141      retVal += a[2]; retVal <<= 8;
142      retVal += a[3];
143      return retVal;
```

```
144 }
```

9.12.3.13 ByteArrayToUint64()

Function to write an integer to a byte array

```
145 UINT64
146 ByteArrayToUint64(
147     BYTE                *a
148     )
149 {
150     UINT64      retVal;
151     retVal  = a[0]; retVal <<= 8;
152     retVal += a[1]; retVal <<= 8;
153     retVal += a[2]; retVal <<= 8;
154     retVal += a[3]; retVal <<= 8;
155     retVal += a[4]; retVal <<= 8;
156     retVal += a[5]; retVal <<= 8;
157     retVal += a[6]; retVal <<= 8;
158     retVal += a[7];
159     return retVal;
160 }
```

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.

```
void
TPMInit(
    void
    )
{
    // Set state as not initialized. This means that Startup is required
    g_initialized = FALSE;
    return;
}
```

9.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2_Startup() has completed successfully).

```
void
TPMRegisterStartup(
    void
    )
{
    g_initialized = TRUE;
    return;
}
```

9.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2_Startup() has completed successfully after a _TPM_Init()).

Return Value	Meaning
TRUE(1)	TPM has been initialized
FALSE(0)	TPM has not been initialized

```
BOOL
TPMIsStarted(
    void
    )
{
```



```
25      return g_initialized;
26  }
```

9.14 PropertyCap.c

9.14.1 Description

This file contains the functions that are used for accessing the TPM_CAP_TPM_PROPERTY values.

9.14.2 Includes

```
1  #include "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(1)	referenced property exists and *value* set
FALSE(0)	referenced property does not exist

```
2  static BOOL
3  TPMPropertyIsDefined(
4      TPM_PT           property,       // IN: property
5      UINT32          *value           // OUT: property value
6      )
7  {
8      switch(property)
9      {
10         case TPM_PT_FAMILY_INDICATOR:
11             // from the title page of the specification
12             // For this specification, the value is "2.0".
13             *value = TPM_SPEC_FAMILY;
14             break;
15         case TPM_PT_LEVEL:
16             // from the title page of the specification
17             *value = TPM_SPEC_LEVEL;
18             break;
19         case TPM_PT_REVISION:
20             // from the title page of the specification
21             *value = TPM_SPEC_VERSION;
22             break;
23         case TPM_PT_DAY_OF_YEAR:
24             // computed from the date value on the title page of the specification
25             *value = TPM_SPEC_DAY_OF_YEAR;
26             break;
27         case TPM_PT_YEAR:
28             // from the title page of the specification
29             *value = TPM_SPEC_YEAR;
30             break;
31         case TPM_PT_MANUFACTURER:
32             // vendor ID unique to each TPM manufacturer
33             *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
34             break;
35         case TPM_PT_VENDOR_STRING_1:
36             // first four characters of the vendor ID string
37             *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
```



```
38              break;
39          case TPM_PT_VENDOR_STRING_2:
40              // second four characters of the vendor ID string
41  #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                 SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, sharedNV);
133                 SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, objectCopiedToRam);
134
135                 // Note: For a LSb0 machine, the bits in a bit field are in the correct
136                 // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
137                 // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
138                 // be NO) so the bits are manipulate correctly.
139                 *value = attributes.u32;
140                 break;
141             }
142         case TPM_PT_CLOCK_UPDATE:
143             // interval, in seconds, between updates to the copy of
144             // TPMS_TIME_INFO .clock in NV
145             *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
146             break;
147         case TPM_PT_CONTEXT_HASH:
148             // algorithm used for the integrity hash on saved contexts and
149             // for digesting the fuData of TPM2_FirmwareRead()
150             *value = CONTEXT_INTEGRITY_HASH_ALG;
151             break;
152         case TPM_PT_CONTEXT_SYM:
153             // algorithm used for encryption of saved contexts
154             *value = CONTEXT_ENCRYPT_ALG;
155             break;
156         case TPM_PT_CONTEXT_SYM_SIZE:
157             // size of the key used for encryption of saved contexts
158             *value = CONTEXT_ENCRYPT_KEY_BITS;
159             break;
160         case TPM_PT_ORDERLY_COUNT:
161             // maximum difference between the volatile and non-volatile
162             // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
163             *value = MAX_ORDERLY_COUNT;
164             break;
165         case TPM_PT_MAX_COMMAND_SIZE:
166             // maximum value for 'commandSize'
167             *value = MAX_COMMAND_SIZE;
168             break;
169         case TPM_PT_MAX_RESPONSE_SIZE:
```



```
170                 // maximum value for 'responseSize'
171                 *value = MAX_RESPONSE_SIZE;
172                 break;
173             case TPM_PT_MAX_DIGEST:
174                 // maximum size of a digest that can be produced by the TPM
175                 *value = sizeof(TPMU_HA);
176                 break;
177             case TPM_PT_MAX_OBJECT_CONTEXT:
178     // Header has 'sequence', 'handle' and 'hierarchy'
179     #define SIZE_OF_CONTEXT_HEADER                    \
180         sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) + sizeof(TPMI_RH_HIERARCHY)
181     #define SIZE_OF_CONTEXT_INTEGRITY (sizeof(UINT16) + CONTEXT_INTEGRITY_HASH_SIZE)
182     #define SIZE_OF_FINGERPRINT     sizeof(UINT64)
183     #define SIZE_OF_CONTEXT_BLOB_OVERHEAD                 \
184             (sizeof(UINT16)  + SIZE_OF_CONTEXT_INTEGRITY + SIZE_OF_FINGERPRINT)
185     #define SIZE_OF_CONTEXT_OVERHEAD                      \
186             (SIZE_OF_CONTEXT_HEADER + SIZE_OF_CONTEXT_BLOB_OVERHEAD)
187     #if 0
188                 // maximum size of a TPMS_CONTEXT that will be returned by
189                 // TPM2_ContextSave for object context
190                 *value = 0;
191                 // adding sequence, saved handle and hierarchy
192                 *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
193                     sizeof(TPMI_RH_HIERARCHY);
194                 // add size field in TPM2B_CONTEXT
195                 *value += sizeof(UINT16);
196                 // add integrity hash size
197                 *value += sizeof(UINT16) +
198                     CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
199                 // Add fingerprint size, which is the same as sequence size
200                 *value += sizeof(UINT64);
201                 // Add OBJECT structure size
202                 *value += sizeof(OBJECT);
203     #else
204                 // the maximum size of a TPMS_CONTEXT that will be returned by
205                 // TPM2_ContextSave for object context
206                 *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(OBJECT);
207     #endif
208                 break;
209             case TPM_PT_MAX_SESSION_CONTEXT:
210     #if 0
211
212                 // the maximum size of a TPMS_CONTEXT that will be returned by
213                 // TPM2_ContextSave for object context
214                 *value = 0;
215                 // adding sequence, saved handle and hierarchy
216                 *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
217                     sizeof(TPMI_RH_HIERARCHY);
218                 // Add size field in TPM2B_CONTEXT
219                 *value += sizeof(UINT16);
220     // Add integrity hash size
221                 *value += sizeof(UINT16) +
222                     CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
223           // Add fingerprint size, which is the same as sequence size
224                 *value += sizeof(UINT64);
225                 // Add SESSION structure size
226                 *value += sizeof(SESSION);
227     #else
228                  // the maximum size of a TPMS_CONTEXT that will be returned by
229                 // TPM2_ContextSave for object context
230                 *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(SESSION);
231     #endif
232                 break;
233             case TPM_PT_PS_FAMILY_INDICATOR:
234                 // platform specific values for the TPM_PT_PS parameters from
235                 // the relevant platform-specific specification
```

```
236                 // In this reference implementation, all of these values are 0.
237                 *value = PLATFORM_FAMILY;
238                 break;
239             case TPM_PT_PS_LEVEL:
240                 // level of the platform-specific specification
241                 *value = PLATFORM_LEVEL;
242                 break;
243             case TPM_PT_PS_REVISION:
244                 // specification Revision times 100 for the platform-specific
245                 // specification
246                 *value = PLATFORM_VERSION;
247                 break;
248             case TPM_PT_PS_DAY_OF_YEAR:
249                 // platform-specific specification day of year using TCG calendar
250                 *value = PLATFORM_DAY_OF_YEAR;
251                 break;
252             case TPM_PT_PS_YEAR:
253                 // platform-specific specification year using the CE
254                 *value = PLATFORM_YEAR;
255                 break;
256             case TPM_PT_SPLIT_MAX:
257                 // number of split signing operations supported by the TPM
258                 *value = 0;
259     #if     ALG_ECC
260                 *value = sizeof(gr.commitArray) * 8;
261     #endif
262                 break;
263             case TPM_PT_TOTAL_COMMANDS:
264                 // total number of commands implemented in the TPM
265                 // Since the reference implementation does not have any
266                 // vendor-defined commands, this will be the same as the
267                 // number of library commands.
268             {
269     #if COMPRESSED_LISTS
270                 (*value) = COMMAND_COUNT;
271     #else
272                 COMMAND_INDEX       commandIndex;
273                 *value = 0;
274
275                 // scan all implemented commands
276                 for(commandIndex = GetClosestCommandIndex(0);
277                 commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
278                     commandIndex = GetNextCommandIndex(commandIndex))
279                 {
280                     (*value)++;     // count of all implemented
281                 }
282     #endif
283                 break;
284             }
285             case TPM_PT_LIBRARY_COMMANDS:
286                 // number of commands from the TPM library that are implemented
287             {
288     #if COMPRESSED_LISTS
289                 *value = LIBRARY_COMMAND_ARRAY_SIZE;
290     #else
291                 COMMAND_INDEX       commandIndex;
292                 *value = 0;
293
294                 // scan all implemented commands
295                 for(commandIndex = GetClosestCommandIndex(0);
296                 commandIndex < LIBRARY_COMMAND_ARRAY_SIZE;
297                     commandIndex = GetNextCommandIndex(commandIndex))
298                 {
299                     (*value)++;
300                 }
301     #endif
```



```
302             break;
303         }
304         case TPM_PT_VENDOR_COMMANDS:
305             // number of vendor commands that are implemented
306             *value = VENDOR_COMMAND_ARRAY_SIZE;
307             break;
308         case TPM_PT_NV_BUFFER_MAX:
309             // Maximum data size in an NV write command
310             *value = MAX_NV_BUFFER_SIZE;
311             break;
312         case TPM_PT_MODES:
313 #if FIPS_COMPLIANT
314             *value = 1;
315 #else
316             *value = 0;
317 #endif
318             break;
319         case TPM_PT_MAX_CAP_BUFFER:
320             *value = MAX_CAP_BUFFER;
321             break;
322
323         // Start of variable commands
324         case TPM_PT_PERMANENT:
325             // TPMA_PERMANENT
326         {
327             union {
328                 TPMA_PERMANENT      attr;
329                 UINT32              u32;
330             } flags = {{0}};
331             if(gp.ownerAuth.t.size != 0)
332                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, ownerAuthSet);
333             if(gp.endorsementAuth.t.size != 0)
334                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, endorsementAuthSet);
335             if(gp.lockoutAuth.t.size != 0)
336                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, lockoutAuthSet);
337             if(gp.disableClear)
338                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, disableClear);
339             if(gp.failedTries >= gp.maxTries)
340                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, inLockout);
341             // In this implementation, EPS is always generated by TPM
342             SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, tpmGeneratedEPS);
343
344             // Note: For a LSb0 machine, the bits in a bit field are in the correct
345             // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
346             // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
347             // be NO) so the bits are manipulate correctly.
348             *value = flags.u32;
349             break;
350         }
351         case TPM_PT_STARTUP_CLEAR:
352             // TPMA_STARTUP_CLEAR
353         {
354             union {
355                 TPMA_STARTUP_CLEAR  attr;
356                 UINT32              u32;
357             } flags = {{0}};
358 //
359             if(g_phEnable)
360                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnable);
361             if(gc.shEnable)
362                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, shEnable);
363             if(gc.ehEnable)
364                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, ehEnable);
365             if(gc.phEnableNV)
366                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnableNV);
367             if(g_prevOrderlyState != SU_NONE_VALUE)
```

```
368                     SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, orderly);
369
370                 // Note: For a LSb0 machine, the bits in a bit field are in the correct
371                 // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
372                 // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
373                 // be NO) so the bits are manipulate correctly.
374                 *value = flags.u32;
375                 break;
376             }
377             case TPM_PT_HR_NV_INDEX:
378                 // number of NV indexes currently defined
379                 *value = NvCapGetIndexNumber();
380                 break;
381             case TPM_PT_HR_LOADED:
382                 // number of authorization sessions currently loaded into TPM
383                 // RAM
384                 *value = SessionCapGetLoadedNumber();
385                 break;
386             case TPM_PT_HR_LOADED_AVAIL:
387                 // number of additional authorization sessions, of any type,
388                 // that could be loaded into TPM RAM
389                 *value = SessionCapGetLoadedAvail();
390                 break;
391             case TPM_PT_HR_ACTIVE:
392                 // number of active authorization sessions currently being
393                 // tracked by the TPM
394                 *value = SessionCapGetActiveNumber();
395                 break;
396             case TPM_PT_HR_ACTIVE_AVAIL:
397                 // number of additional authorization sessions, of any type,
398                 // that could be created
399                 *value = SessionCapGetActiveAvail();
400                 break;
401             case TPM_PT_HR_TRANSIENT_AVAIL:
402                 // estimate of the number of additional transient objects that
403                 // could be loaded into TPM RAM
404                 *value = ObjectCapGetTransientAvail();
405                 break;
406             case TPM_PT_HR_PERSISTENT:
407                 // number of persistent objects currently loaded into TPM
408                 // NV memory
409                 *value = NvCapGetPersistentNumber();
410                 break;
411             case TPM_PT_HR_PERSISTENT_AVAIL:
412                 // number of additional persistent objects that could be loaded
413                 // into NV memory
414                 *value = NvCapGetPersistentAvail();
415                 break;
416             case TPM_PT_NV_COUNTERS:
417                 // number of defined NV indexes that have NV TPMA_NV_COUNTER
418                 // attribute SET
419                 *value = NvCapGetCounterNumber();
420                 break;
421             case TPM_PT_NV_COUNTERS_AVAIL:
422                 // number of additional NV indexes that can be defined with their
423                 // TPMA_NV_COUNTER attribute SET
424                 *value = NvCapGetCounterAvail();
425                 break;
426             case TPM_PT_ALGORITHM_SET:
427                 // region code for the TPM
428                 *value = gp.algorithmSet;
429                 break;
430             case TPM_PT_LOADED_CURVES:
431 #if         ALG_ECC
432             // number of loaded ECC curves
433                 *value = ECC_CURVE_COUNT;
```



```
434     #else // ALG_ECC
435                 *value = 0;
436     #endif // ALG_ECC
437                 break;
438             case TPM_PT_LOCKOUT_COUNTER:
439                 // current value of the lockout counter
440                 *value = gp.failedTries;
441                 break;
442             case TPM_PT_MAX_AUTH_FAIL:
443                 // number of authorization failures before DA lockout is invoked
444                 *value = gp.maxTries;
445                 break;
446             case TPM_PT_LOCKOUT_INTERVAL:
447                 // number of seconds before the value reported by
448                 // TPM_PT_LOCKOUT_COUNTER is decremented
449                 *value = gp.recoveryTime;
450                 break;
451             case TPM_PT_LOCKOUT_RECOVERY:
452                 // number of seconds after a lockoutAuth failure before use of
453                 // lockoutAuth may be attempted again
454                 *value = gp.lockoutRecovery;
455                 break;
456             case TPM_PT_NV_WRITE_RECOVERY:
457                 // number of milliseconds before the TPM will accept another command
458                 // that will modify NV.
459                 // This should make a call to the platform code that is doing rate
460                 // limiting of NV. Rate limiting is not implemented in the reference
461                 // code so no call is made.
462                 *value = 0;
463                 break;
464             case TPM_PT_AUDIT_COUNTER_0:
465                 // high-order 32 bits of the command audit counter
466                 *value = (UINT32)(gp.auditCounter >> 32);
467                 break;
468             case TPM_PT_AUDIT_COUNTER_1:
469                 // low-order 32 bits of the command audit counter
470                 *value = (UINT32)(gp.auditCounter);
471                 break;
472             default:
473                 // property is not defined
474                 return FALSE;
475                 break;
476         }
477         return TRUE;
478     }
```

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

```
479     TPMI_YES_NO
480     TPMCapGetProperties(
481         TPM_PT                      property,       // IN: the starting TPM property
482         UINT32                      count,          // IN: maximum number of returned
483                                                     //     properties
484         TPML_TAGGED_TPM_PROPERTY    *propertyList   // OUT: property list
```

```
485         )
486     {
487         TPMI_YES_NO     more = NO;
488         UINT32          i;
489         UINT32          nextGroup;
490
491         // initialize output property list
492         propertyList->count = 0;
493
494         // maximum count of properties we may return is MAX_PCR_PROPERTIES
495         if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;
496
497         // if property is less than PT_FIXED, start from PT_FIXED
498         if(property < PT_FIXED)
499             property = PT_FIXED;
500         // There is only the fixed and variable groups with the variable group coming
501         // last
502         if(property >= (PT_VAR + PT_GROUP))
503             return more;
504
505         // Don't read past the end of the selected group
506         nextGroup = ((property / PT_GROUP) * PT_GROUP) + PT_GROUP;
507
508         // Scan through the TPM properties of the requested group.
509         for(i = property; i < nextGroup; i++)
510         {
511             UINT32          value;
512             // if we have hit the end of the group, quit
513             if(i != property && ((i % PT_GROUP) == 0))
514                 break;
515             if(TPMPropertyIsDefined((TPM_PT)i, &value))
516             {
517                 if(propertyList->count < count)
518                 {
519                     // If the list is not full, add this property
520                     propertyList->tpmProperty[propertyList->count].property =
521                         (TPM_PT)i;
522                     propertyList->tpmProperty[propertyList->count].value = value;
523                     propertyList->count++;
524                 }
525                 else
526                 {
527                     // If the return list is full but there are more properties
528                     // available, set the indication and exit the loop.
529                     more = YES;
530                     break;
531                 }
532             }
533         }
534         return more;
535     }
```


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 response 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

```
#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.

```
TPM_RC
RcSafeAddToResult(
    TPM_RC          responseCode,
    TPM_RC          modifier
    )
{
    if((responseCode & RC_FMT1) && !(responseCode & 0xf40))
       return responseCode + modifier;
    else
        return responseCode;
}
```


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
{
```

```
    BYTE            test[sizeof(TEST_RESPONSE)];
    BYTE            cap[sizeof(CAPABILITY_RESPONSE)];
} 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.

```
#ifndef __IGNORE_STATE__ // Don't define this value
static BYTE response[sizeof(RESPONSES)];
#endif
```

9.17.3 Local Functions

9.17.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```
static INT32
MarshalUint16(
    UINT16          integer,
    BYTE            **buffer
    )
{
    return UINT16_Marshal(&integer, buffer, NULL);
}
```

9.17.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```
static INT32
MarshalUint32(
    UINT32          integer,
    BYTE            **buffer
    )
{
    return UINT32_Marshal(&integer, buffer, NULL);
}
```

9.17.3.3 UnmarshalHeader()

function to unmarshal the 10-byte command header.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
static BOOL
UnmarshalHeader(
    HEADER          *header,
    BYTE            **buffer,
    INT32           *size
    )
{
```



```
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

9.17.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```
85  #if SIMULATION
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  #if FAIL_TRACE
98      const char      *function,
99      int             line,
100 #endif
101     int             code
102     )
103 {
104     // Save the values that indicate where the error occurred.
105     // On a 64-bit machine, this may truncate the address of the string
106     // of the function name where the error occurred.
107 #if FAIL_TRACE
108     s_failFunction = (UINT32)function;
109     s_failLine = line;
110 #else
111     s_failFunction = (UINT32)NULL;
112     s_failLine = 0;
113 #endif
114     s_failCode = code;
115
116     // We are in failure mode
117     g_inFailureMode = TRUE;
118
119     // if asserts are enabled, then do an assert unless the failure mode code
120     // is being tested.
121 #if SIMULATION
122 #   ifndef NDEBUG
123     assert(g_forceFailureMode);
```

```
124 #   endif
125     // Clear this flag
126     g_forceFailureMode = FALSE;
127 #endif
128     // Jump to the failure mode code.
129     // Note: only get here if asserts are off or if we are testing failure mode
130     _plat__Fail();
131 }
```

9.17.4.3 TpmFailureMode

This function is called by the interface code when the platform is in failure mode.

```
132 void
133 TpmFailureMode(
134     unsigned int     inRequestSize,     // IN: command buffer size
135     unsigned char   *inRequest,         // IN: command buffer
136     unsigned int    *outResponseSize,   // OUT: response buffer size
137     unsigned char   **outResponse       // OUT: response buffer
138     )
139 {
140     BYTE            *buffer;
141     UINT32           marshalSize;
142     UINT32           capability;
143     HEADER           header;    // unmarshaled command header
144     UINT32           pt;    // unmarshaled property type
145     UINT32           count; // unmarshaled property count
146
147     // If there is no command buffer, then just return TPM_RC_FAILURE
148     if(inRequestSize == 0 || inRequest == NULL)
149         goto FailureModeReturn;
150     // If the header is not correct for TPM2_GetCapability() or
151     // TPM2_GetTestResult() then just return the in failure mode response;
152     buffer = inRequest;
153     if(!UnmarshalHeader(&header, &inRequest, (INT32 *)&inRequestSize))
154         goto FailureModeReturn;
155     if(header.tag != TPM_ST_NO_SESSIONS
156        || header.size < 10)
157         goto FailureModeReturn;
158     switch(header.code)
159     {
160         case TPM_CC_GetTestResult:
161             // make sure that the command size is correct
162             if(header.size != 10)
163                 goto FailureModeReturn;
164             buffer = &response[10];
165             marshalSize = MarshalUint16(3 * sizeof(UINT32), &buffer);
166             marshalSize += MarshalUint32(s_failFunction, &buffer);
167             marshalSize += MarshalUint32(s_failLine, &buffer);
168             marshalSize += MarshalUint32(s_failCode, &buffer);
169             if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
170                 marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
171             else
172                 marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
173             break;
174         case TPM_CC_GetCapability:
175             // make sure that the size of the command is exactly the size
176             // returned for the capability, property, and count
177             if(header.size != (10 + (3 * sizeof(UINT32)))
178                     // also verify that this is requesting TPM properties
179                || TPM_RC_SUCCESS != UINT32_Unmarshal(&capability, &inRequest,
180                                                      (INT32 *)&inRequestSize)
181                || capability != TPM_CAP_TPM_PROPERTIES
182                || TPM_RC_SUCCESS != UINT32_Unmarshal(&pt, &inRequest,
```



```
183                                                     (INT32 *)&inRequestSize)
184                 || TPM_RC_SUCCESS != UINT32_Unmarshal(&count, &inRequest,
185                                                     (INT32 *)&inRequestSize))
186                     goto FailureModeReturn;
187             // If in failure mode because of an unrecoverable read error, and the
188             // property is 0 and the count is 0, then this is an indication to
189             // re-manufacture the TPM. Do the re-manufacture but stay in failure
190             // mode until the TPM is reset.
191             // Note: this behavior is not required by the specification and it is
192             // OK to leave the TPM permanently bricked due to an unrecoverable NV
193             // error.
194             if(count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
195             {
196                 g_manufactured = FALSE;
197                 TPM_Manufacture(0);
198             }
199             if(count > 0)
200                 count = 1;
201             else if(pt > TPM_PT_FIRMWARE_VERSION_2)
202                 count = 0;
203             if(pt < TPM_PT_MANUFACTURER)
204                 pt = TPM_PT_MANUFACTURER;
205             // set up for return
206             buffer = &response[10];
207             // if the request was for a PT less than the last one
208             // then we indicate more, otherwise, not.
209             if(pt < TPM_PT_FIRMWARE_VERSION_2)
210                 *buffer++ = YES;
211             else
212                 *buffer++ = NO;
213             marshalSize = 1;
214
215             // indicate the capability type
216             marshalSize += MarshalUint32(capability, &buffer);
217             // indicate the number of values that are being returned (0 or 1)
218             marshalSize += MarshalUint32(count, &buffer);
219             // indicate the property
220             marshalSize += MarshalUint32(pt, &buffer);
221
222             if(count > 0)
223                 switch(pt)
224                 {
225                     case TPM_PT_MANUFACTURER:
226                     // the vendor ID unique to each TPM manufacturer
227 #ifdef  MANUFACTURER
228                         pt = *(UINT32*)MANUFACTURER;
229 #else
230                         pt = 0;
231 #endif
232                         break;
233                     case TPM_PT_VENDOR_STRING_1:
234                         // the first four characters of the vendor ID string
235 #ifdef  VENDOR_STRING_1
236                         pt = *(UINT32*)VENDOR_STRING_1;
237 #else
238                         pt = 0;
239 #endif
240                         break;
241                     case TPM_PT_VENDOR_STRING_2:
242                         // the second four characters of the vendor ID string
243 #ifdef  VENDOR_STRING_2
244                         pt = *(UINT32*)VENDOR_STRING_2;
245 #else
246                         pt = 0;
247 #endif
248                         break;
```



```
249                     case TPM_PT_VENDOR_STRING_3:
250                         // the third four characters of the vendor ID string
251 #ifdef  VENDOR_STRING_3
252                         pt = *(UINT32*)VENDOR_STRING_3;
253 #else
254                         pt = 0;
255 #endif
256                         break;
257                     case TPM_PT_VENDOR_STRING_4:
258                         // the fourth four characters of the vendor ID string
259 #ifdef  VENDOR_STRING_4
260                         pt = *(UINT32*)VENDOR_STRING_4;
261 #else
262                         pt = 0;
263 #endif
264                         break;
265                     case TPM_PT_VENDOR_TPM_TYPE:
266                         // vendor-defined value indicating the TPM model
267                         // We just make up a number here
268                         pt = 1;
269                         break;
270                     case TPM_PT_FIRMWARE_VERSION_1:
271                         // the more significant 32-bits of a vendor-specific value
272                         // indicating the version of the firmware
273 #ifdef  FIRMWARE_V1
274                         pt = FIRMWARE_V1;
275 #else
276                         pt = 0;
277 #endif
278                         break;
279                     default: // TPM_PT_FIRMWARE_VERSION_2:
280                         // the less significant 32-bits of a vendor-specific value
281                         // indicating the version of the firmware
282 #ifdef  FIRMWARE_V2
283                         pt = FIRMWARE_V2;
284 #else
285                         pt = 0;
286 #endif
287                         break;
288                 }
289             marshalSize += MarshalUint32(pt, &buffer);
290             break;
291         default: // default for switch (cc)
292             goto FailureModeReturn;
293     }
294     // Now do the header
295     buffer = response;
296     marshalSize = marshalSize + 10; // Add the header size to the
297                                     // stuff already marshaled
298     MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
299     MarshalUint32(marshalSize, &buffer);  // responseSize
300     MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
301
302     *outResponseSize = marshalSize;
303     *outResponse = (unsigned char *)&response;
304     return;
305 FailureModeReturn:
306     buffer = response;
307     marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
308     marshalSize += MarshalUint32(10, &buffer);
309     marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
310     *outResponseSize = marshalSize;
311     *outResponse = (unsigned char *)response;
312     return;
313 }
```


9.17.4.4 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...

```
314 void
315 UnmarshalFail(
316     void            *type,
317     BYTE            **buffer,
318     INT32           *size
319     )
320 {
321     NOT_REFERENCED(type);
322     NOT_REFERENCED(buffer);
323     NOT_REFERENCED(size);
324     FAIL(FATAL_ERROR_INTERNAL);
325 }
```

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

```
1   #ifndef _BN_NUMBERS_H
2   #define _BN_NUMBERS_H
3   #if RADIX_BITS == 64
4   # define RADIX_LOG2          6
5   #elif RADIX_BITS == 32
6   #define RADIX_LOG2          5
7   #else
8   # error "Unsupported radix"
9   #endif
10  #define RADIX_MOD(x)        ((x) & ((1 << RADIX_LOG2) - 1))
11  #define RADIX_DIV(x)        ((x) >> RADIX_LOG2)
12  #define RADIX_MASK  ((((crypt_uword_t)1) << RADIX_LOG2) - 1)
13  #define BITS_TO_CRYPT_WORDS(bits)       RADIX_DIV((bits) + (RADIX_BITS - 1))
14  #define BYTES_TO_CRYPT_WORDS(bytes)     BITS_TO_CRYPT_WORDS(bytes * 8)
15  #define SIZE_IN_CRYPT_WORDS(thing)      BYTES_TO_CRYPT_WORDS(sizeof(thing))
16  #if RADIX_BITS == 64
17  #define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_64(x)
18      typedef uint64_t    crypt_uword_t;
19      typedef int64_t     crypt_word_t;
20  #   define TO_CRYPT_WORD_64            BIG_ENDIAN_BYTES_TO_UINT64
21  #   define TO_CRYPT_WORD_32(a, b, c, d) TO_CRYPT_WORD_64(0, 0, 0, 0, a, b, c, d)
22  #elif RADIX_BITS == 32
23  #   define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_32((x))
24      typedef uint32_t    crypt_uword_t;
25      typedef int32_t     crypt_word_t;
26  #   define TO_CRYPT_WORD_64(a, b, c, d, e, f, g, h)                          \
27          BIG_ENDIAN_BYTES_TO_UINT32(e, f, g, h),                            \
28          BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)
29  #endif
30  #define MAX_CRYPT_UWORD (~((crypt_uword_t)0))
31  #define MAX_CRYPT_WORD  ((crypt_word_t)(MAX_CRYPT_UWORD >> 1))
32  #define MIN_CRYPT_WORD  (~MAX_CRYPT_WORD)
33  #define LARGEST_NUMBER (MAX((ALG_RSA * MAX_RSA_KEY_BYTES),                  \
34                          MAX((ALG_ECC * MAX_ECC_KEY_BYTES), MAX_DIGEST_SIZE)))
35  #define LARGEST_NUMBER_BITS (LARGEST_NUMBER * 8)
36  #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.

```
37  #define BN_STRUCT_DEF(count) struct {        \
38      crypt_uword_t       allocated;           \
39      crypt_uword_t       size;                \
40      crypt_uword_t       d[count];            \
41      }
42  typedef BN_STRUCT_DEF(1) bignum_t;
43  #ifndef bigNum
44  typedef bignum_t       *bigNum;
45  typedef const bignum_t *bigConst;
46  #endif
47  extern const bignum_t   BnConstZero;
```

The Functions to access the properties of a big number. Get number of allocated words

```
48  #define BnGetAllocated(x)    (unsigned)((x)->allocated)
```

Get number of words used

```
49  #define BnGetSize(x)         ((x)->size)
```

Get a pointer to the data array

```
50  #define BnGetArray(x)        ((crypt_uword_t *)&((x)->d[0]))
```

Get the nth word of a BIGNUM (zero-based)

```
51  #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

```
52  #define BnEqualZero(bn)    (BnGetSize(bn) == 0)
```

Test to see if a bignum_t is equal to a word type

```
53  #define BnEqualWord(bn, word)                          \
54          ((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.

```
55  #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*.

```
56  #define BN_INIT(name)                                          \
57      (bigNum)BnInit((bigNum)&(name),                            \
58                  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_*

```
59  #define BN_ADDRESS(name)  (bigNum)&name##_
60  #define BN_STRUCT_ALLOCATION(bits)  (BITS_TO_CRYPT_WORDS(bits) + 1)
```

Create a structure of the correct size.

```
61  #define BN_STRUCT(bits)                                          \
62      BN_STRUCT_DEF(BN_STRUCT_ALLOCATION(bits))
```

Define a BIGNUM type with a specific allocation

```
63  #define BN_TYPE(name, bits)                                      \
64      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.

```
65  #define BN_INITIALIZED(name, bits, initializer)                  \
66      BN_STRUCT(bits)  name##_;                                    \
67      bigNum           name = BnFrom2B(BN_INIT(name##_),           \
68                                      (const TPM2B *)initializer)
```

Create a local variable that can hold a number with *bits*

```
69  #define BN_VAR(name, bits)                                       \
70      BN_STRUCT(bits)  _##name;                                     \
71      bigNum           name = BN_INIT(_##name)
```

Create a type that can hold the largest number defined by the implementation.

```
72  #define BN_MAX(name)   BN_VAR(name, LARGEST_NUMBER_BITS)
73  #define BN_MAX_INITIALIZED(name, initializer)                    \
74      BN_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)
```

A word size value is useful

```
75  #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.

```
76  #define BN_WORD_INITIALIZED(name, initial)                          \
77      BN_STRUCT(RADIX_BITS)  name##_;                                 \
78      bigNum                 name = BnInitializeWord((bigNum)&name##_,    \
79                                    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

```
80  typedef struct _bn_point_t
81  {
82      bigNum          x;
83      bigNum          y;
84      bigNum          z;
85  } bn_point_t;
86  typedef bn_point_t          *bigPoint;
87  typedef const bn_point_t    *pointConst;
88  typedef struct constant_point_t
89  {
90      bigConst        x;
91      bigConst        y;
92      bigConst        z;
93  } constant_point_t;
94  #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 CryptEccData.c.

```
#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)                                \
            +    ((UINT64)(c) << 40)                                \
            +    ((UINT64)(d) << 32)                                \
```

```
147             +    ((UINT64)(e) << 24)                                  \
148             +    ((UINT64)(f) << 16)                                  \
149             +    ((UINT64)(g) << 8)                                   \
150             +    ((UINT64)(h))                                        \
151             )
```

Add implementation dependent definitions for other ECC Values and for linkages. MATH_LIB_H is defined in LibSupport.h

```
152 #include MATHLIB_H
153 #ifndef RADIX_BYTES
154 #   if RADIX_BITS == 32
155 #       define RADIX_BYTES 4
156 #   elif RADIX_BITS == 64
157 #       define RADIX_BYTES 8
158 #   else
159 #       error "RADIX_BITS must either be 32 or 64"
160 #   endif
161 #endif
162 #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

```
#ifndef _CRYPT_ECC_H
#define _CRYPT_ECC_H
```

10.1.2.2 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.

```
#ifdef NAMED_CURVES
#define CURVE_NAME(a) , a
#define CURVE_NAME_DEF const char *name;
#else
#  define CURVE_NAME(a)
#  define CURVE_NAME_DEF
#endif
typedef struct ECC_CURVE
{
    const TPM_ECC_CURVE          curveId;
    const UINT16                 keySizeBits;
    const TPMT_KDF_SCHEME        kdf;
    const TPMT_ECC_SCHEME        sign;
    const ECC_CURVE_DATA        *curveData; // the address of the curve data
    CURVE_NAME_DEF
} ECC_CURVE;
extern const ECC_CURVE eccCurves[ECC_CURVE_COUNT];
#endif
```

10.1.3 CryptHash.h

10.1.3.1 Introduction

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.

```
1   #ifndef _CRYPT_HASH_H
2   #define _CRYPT_HASH_H
```

10.1.3.2 Hash-related Structures

```
3   typedef struct
4   {
5       const TPM_ALG_ID     alg;
6       const UINT16         digestSize;
7       const UINT16         blockSize;
8       const UINT16         derSize;
9       const BYTE           der[20];
10  } HASH_INFO;
11  union SMAC_STATES;
```

These definitions add the high-level methods for processing state that may be an SMAC

```
12  typedef void(* SMAC_DATA_METHOD)(
13      union SMAC_STATES       *state,
14      UINT32                   size,
15      const BYTE              *buffer
16      );
17  typedef UINT16(* SMAC_END_METHOD)(
18      union SMAC_STATES       *state,
19      UINT32                   size,
20      BYTE                    *buffer
21      );
22  typedef struct sequenceMethods {
23      SMAC_DATA_METHOD          data;
24      SMAC_END_METHOD           end;
25  } SMAC_METHODS;
26  #define SMAC_IMPLEMENTED (defined TPM_CC_MAC || defined TPM_CC_MAC_Start)
```

These definitions are here because the SMAC state is in the union of hash states.

```
27  typedef struct tpmCmacState {
28      TPM_ALG_ID              symAlg;
29      UINT16                  keySizeBits;
30      INT16                   bcount; // current count of bytes accumulated in IV
31      TPM2B_IV                iv;     // IV buffer
32      TPM2B_SYM_KEY           symKey;
33  } tpmCmacState_t;
34  typedef union SMAC_STATES {
35  #if     ALG_CMAC
36      tpmCmacState_t          cmac;
37  #endif
38      UINT64                  pad;
39  } SMAC_STATES;
40  typedef struct SMAC_STATE {
41      SMAC_METHODS            smacMethods;
42      SMAC_STATES             state;
43  } SMAC_STATE;
```



```
44  typedef union
45  {
46  #if     ALG_SHA1
47      tpmHashStateSHA1_t          Sha1;
48  #endif
49  #if     ALG_SHA256
50      tpmHashStateSHA256_t        Sha256;
51  #endif
52  #if     ALG_SHA384
53      tpmHashStateSHA384_t        Sha384;
54  #endif
55  #if     ALG_SHA512
56      tpmHashStateSHA512_t        Sha512;
57  #endif
58
59  // Additions for symmetric block cipher MAC
60  #if SMAC_IMPLEMENTED
61      SMAC_STATE                  smac;
62  #endif
63      // to force structure alignment to be no worse than HASH_ALIGNMENT
64  #if HASH_ALIGNMENT == 4
65      uint32_t            align;
66  #else
67      uint64_t            align;
68  #endif
69  } ANY_HASH_STATE;
70  typedef ANY_HASH_STATE *PANY_HASH_STATE;
71  typedef const ANY_HASH_STATE    *PCANY_HASH_STATE;
72  #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

```
73  #define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
74  #define MAX_HASH_STATE_SIZE_ALIGNED                                    \
75                      ALIGNED_SIZE(MAX_HASH_STATE_SIZE, HASH_ALIGNMENT)
```

This is an aligned byte array that will hold any of the hash contexts.

```
76  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

```
77  #ifndef HASH_START_METHOD_DEF
78  #   define HASH_START_METHOD_DEF    void (HASH_START_METHOD)(void)
79  #endif
80  #ifndef HASH_DATA_METHOD_DEF
81  #   define HASH_DATA_METHOD_DEF     void (HASH_DATA_METHOD)(void)
82  #endif
83  #ifndef HASH_END_METHOD_DEF
84  #   define HASH_END_METHOD_DEF      void (HASH_END_METHOD)(void)
85  #endif
86  #ifndef HASH_STATE_COPY_METHOD_DEF
87  #   define HASH_STATE_COPY_METHOD_DEF     void (HASH_STATE_COPY_METHOD)(void)
88  #endif
89  #ifndef  HASH_STATE_EXPORT_METHOD_DEF
90  #   define  HASH_STATE_EXPORT_METHOD_DEF   void (HASH_STATE_EXPORT_METHOD)(void)
91  #endif
92  #ifndef  HASH_STATE_IMPORT_METHOD_DEF
```

```
93  #   define  HASH_STATE_IMPORT_METHOD_DEF   void (HASH_STATE_IMPORT_METHOD)(void)
94  #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.

```
95  typedef HASH_START_METHOD_DEF;
96  typedef HASH_DATA_METHOD_DEF;
97  typedef HASH_END_METHOD_DEF;
98  typedef HASH_STATE_COPY_METHOD_DEF;
99  typedef HASH_STATE_EXPORT_METHOD_DEF;
100 typedef HASH_STATE_IMPORT_METHOD_DEF;
101 typedef struct _HASH_METHODS
102 {
103     HASH_START_METHOD           *start;
104     HASH_DATA_METHOD            *data;
105     HASH_END_METHOD             *end;
106     HASH_STATE_COPY_METHOD      *copy;      // Copy a hash block
107     HASH_STATE_EXPORT_METHOD    *copyOut;   // Copy a hash block from a hash
108                                             // context
109     HASH_STATE_IMPORT_METHOD    *copyIn;    // Copy a hash block to a proper hash
110                                             // context
111 } HASH_METHODS, *PHASH_METHODS;
112 #if ALG_SHA1
113     TPM2B_TYPE(SHA1_DIGEST, SHA1_DIGEST_SIZE);
114 #endif
115 #if ALG_SHA256
116     TPM2B_TYPE(SHA256_DIGEST, SHA256_DIGEST_SIZE);
117 #endif
118 #if ALG_SHA384
119     TPM2B_TYPE(SHA384_DIGEST, SHA384_DIGEST_SIZE);
120 #endif
121 #if ALG_SHA512
122     TPM2B_TYPE(SHA512_DIGEST, SHA512_DIGEST_SIZE);
123 #endif
124 #if ALG_SM3_256
125     TPM2B_TYPE(SM3_256_DIGEST, SM3_256_DIGEST_SIZE);
126 #endif
127 typedef const struct
128 {
129     HASH_METHODS        method;
130     uint16_t            blockSize;
131     uint16_t            digestSize;
132     uint16_t            contextSize;
133     uint16_t            hashAlg;
134 } 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.

```
135 #define HASH_DEF_TEMPLATE(HASH)                                                 \
136     HASH_DEF    HASH##_Def= {                                                   \
137                         {(HASH_START_METHOD *)&tpmHashStart_##HASH,             \
138                          (HASH_DATA_METHOD *)&tpmHashData_##HASH,               \
139                          (HASH_END_METHOD *)&tpmHashEnd_##HASH,                 \
140                          (HASH_STATE_COPY_METHOD *)&tpmHashStateCopy_##HASH,    \
141                          (HASH_STATE_EXPORT_METHOD *)&tpmHashStateExport_##HASH,\
142                          (HASH_STATE_IMPORT_METHOD *)&tpmHashStateImport_##HASH,\
143                         },                                                      \
144                         HASH##_BLOCK_SIZE,     /*block size */                  \
145                         HASH##_DIGEST_SIZE,    /*data size */                   \
146                         sizeof(tpmHashState##HASH##_t),                        \
147                         TPM_ALG_##HASH}
```


These definitions are for the types that can be in a hash state structure. These types are used in the cryptographic 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)
#if CC_MAC || CC_MAC_Start
#define HASH_STATE_SMAC         ((HASH_STATE_TYPE) 3)
#endif
```

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.

NOTE: This version of the state had the pointer to the update method in the state. This is to allow the SMAC functions to use the same structure without having to replicate the entire HASH_DEF structure.

```
typedef struct _HASH_STATE
{
    HASH_STATE_TYPE          type;              // type of the context
    TPM_ALG_ID               hashAlg;
    PHASH_DEF                def;
    ANY_HASH_STATE           state;
} HASH_STATE, *PHASH_STATE;
typedef const HASH_STATE *PCHASH_STATE;
```

10.1.3.3 HMAC State Structures

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 hmacState
{
    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.

```
typedef struct
{
    BYTE                     buffer[sizeof(HASH_STATE)];
} EXPORT_HASH_STATE, *PEXPORT_HASH_STATE;
typedef const EXPORT_HASH_STATE *PCEXPORT_HASH_STATE;
#endif //  _CRYPT_HASH_H
```

10.1.4 CryptRand.h

10.1.4.1 Introduction

This file contains constant definition shared by CryptUtil() and the parts of the Crypto Engine.

```
#ifndef _CRYPT_RAND_H
#define _CRYPT_RAND_H
```

10.1.4.2 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

```
#define DRBG_KEY_SIZE_BITS      MAX_AES_KEY_BITS
#define DRBG_IV_SIZE_BITS       (MAX_AES_BLOCK_SIZE_BYTES * 8)
#define DRBG_ALGORITHM          TPM_ALG_AES
typedef tpmKeyScheduleAES     DRBG_KEY_SCHEDULE;
#define DRBG_ENCRYPT_SETUP(key, keySizeInBits, schedule)        \
            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[DRBG_KEY_SIZE_WORDS];
} DRBG_KEY;
typedef union
{
    BYTE            bytes[DRBG_IV_SIZE_BYTES];
    crypt_uword_t   words[DRBG_IV_SIZE_WORDS];
} DRBG_IV;
typedef union
{
```



```
40      BYTE            bytes[DRBG_SEED_SIZE_BYTES];
41      crypt_uword_t   words[DRBG_SEED_SIZE_WORDS];
42  } DRBG_SEED;
43  #define CTR_DRBG_MAX_REQUESTS_PER_RESEED        ((UINT64)1 << 20)
44  #define CTR_DRBG_MAX_BYTES_PER_REQUEST          (1 << 16)
45  #   define CTR_DRBG_MIN_ENTROPY_INPUT_LENGTH    DRBG_SEED_SIZE_BYTES
46  #   define CTR_DRBG_MAX_ENTROPY_INPUT_LENGTH    DRBG_SEED_SIZE_BYTES
47  #   define CTR_DRBG_MAX_ADDITIONAL_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
48  #define     TESTING         (1 << 0)
49  #define     ENTROPY         (1 << 1)
50  #define     TESTED          (1 << 2)
51  #define     IsTestStateSet(BIT)     ((g_cryptoSelfTestState.rng & BIT) != 0)
52  #define     SetTestStateBit(BIT)    (g_cryptoSelfTestState.rng |= BIT)
53  #define     ClearTestStateBit(BIT)  (g_cryptoSelfTestState.rng &= ~BIT)
54  #define     IsSelfTest()    IsTestStateSet(TESTING)
55  #define     SetSelfTest()   SetTestStateBit(TESTING)
56  #define     ClearSelfTest() ClearTestStateBit(TESTING)
57  #define     IsEntropyBad()      IsTestStateSet(ENTROPY)
58  #define     SetEntropyBad()     SetTestStateBit(ENTROPY)
59  #define     ClearEntropyBad()   ClearTestStateBit(ENTROPY)
60  #define     IsDrbgTested()      IsTestStateSet(TESTED)
61  #define     SetDrbgTested()     SetTestStateBit(TESTED)
62  #define     ClearDrbgTested()   ClearTestStateBit(TESTED)
63  typedef struct
64  {
65      UINT64      reseedCounter;
66      UINT32      magic;
67      DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
68      UINT32      lastValue[4];   // used when the TPM does continuous self-test
69                                  // for FIPS compliance of DRBG
70  } 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      UINT32          limit;
77      TPM2B          *seed;
78      const TPM2B    *label;
79      TPM2B          *context;
80      TPM_ALG_ID      hash;
81      TPM_ALG_ID      kdf;
82      UINT16          digestSize;
83      TPM2B_DIGEST    residual;
84  } KDF_STATE, *pKDR_STATE;
85  #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

```
86  typedef union
87  {
88      DRBG_STATE      drbg;
89      KDF_STATE       kdf;
90  } 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.

```
91  extern RAND_STATE           *s_random;
```

When instrumenting RSA key sieve

```
92    #if  RSA_INSTRUMENT
93    #define PRIME_INDEX(x)  ((x) == 512 ? 0 : (x) == 1024 ? 1 : 2)
94    #   define INSTRUMENT_SET(a, b) ((a) = (b))
95    #   define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
96    #   define INSTRUMENT_INC(a)    (a) = (a) + 1
97    extern UINT32  PrimeIndex;
98    extern UINT32  failedAtIteration[10];
99    extern UINT32  PrimeCounts[3];
100   extern UINT32  MillerRabinTrials[3];
101   extern UINT32  totalFieldsSieved[3];
102   extern UINT32  bitsInFieldAfterSieve[3];
103   extern UINT32  emptyFieldsSieved[3];
104   extern UINT32  noPrimeFields[3];
105   extern UINT32  primesChecked[3];
106   extern UINT16  lastSievePrime;
107   #else
108   #   define INSTRUMENT_SET(a, b)
109   #   define INSTRUMENT_ADD(a, b)
110   #   define INSTRUMENT_INC(a)
111   #endif
112   #endif // _CRYPT_RAND_H
```

10.1.5 CryptRsa.h

This file contains the RSA-related structures and defines.

```
#ifndef _CRYPT_RSA_H
#define _CRYPT_RSA_H
```

This structure is a succinct representation of the cryptographic components of an RSA key. It is used in testing

```
typedef struct
{
    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
```

10.1.6 CryptTest.h

This file contains constant definitions used for self-test.

```
1   #ifndef _CRYPT_TEST_H
2   #define _CRYPT_TEST_H
```

This is the definition of a bit array with one bit per algorithm.

NOTE: Since bit numbering starts at zero, when ALG_LAST_VALUE is a multiple of 8, ALGORITHM_VECTOR will need to have byte for the single bit in the last byte. So, for example, when ALG_LAST_VECTOR is 8, ALGORITHM_VECTOR will need 2 bytes.

```
3   #define ALGORITHM_VECTOR_BYTES  ((ALG_LAST_VALUE + 8) / 8)
4   typedef BYTE    ALGORITHM_VECTOR[ALGORITHM_VECTOR_BYTES];
5   #ifdef  TEST_SELF_TEST
6   LIB_EXPORT    extern  ALGORITHM_VECTOR    LibToTest;
7   #endif
```

This structure is used to contain self-test tracking information for the cryptographic modules. 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.

```
8   typedef struct
9   {
10      UINT32      rng;
11      UINT32      hash;
12      UINT32      sym;
13  #if     ALG_RSA
14      UINT32      rsa;
15  #endif
16  #if     ALG_ECC
17      UINT32      ecc;
18  #endif
19  } CRYPTO_SELF_TEST_STATE;
```

This structure contains the self-test state values for the cryptographic modules.

```
20  extern CRYPTO_SELF_TEST_STATE   g_cryptoSelfTestState;
21  #endif // _CRYPT_TEST_H
```


10.1.7 HashTestData.h

Hash Test Vectors

```
TPM2B_TYPE(HASH_TEST_KEY, 128); // Twice the largest digest size
TPM2B_HASH_TEST_KEY        c_hashTestKey = {{128, {
  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,
  0x41,0x44,0x9d,0x90,0xab,0xd9,0x6b,0x7d,0x48,0x99,0x25,0x93,0x29,0x14,0x2b,0xce,
  0x93,0x8d,0x8c,0xaf,0x31,0x0e,0x9c,0x57,0xd8,0x5b,0x57,0x20,0x1b,0x9f,0x2d,0xa5
  }}};
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_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
  }}};
#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_SHA512 == YES
TPM2B_TYPE(SHA512, 64);
TPM2B_SHA512     c_SHA512_digest = {{64, {
  0xe2,0x7b,0x10,0x3d,0x5e,0x48,0x58,0x44,0x67,0xac,0xa3,0x81,0x8c,0x1d,0xc5,0x71,
  0x66,0x92,0x8a,0x89,0xaa,0xd4,0x35,0x51,0x60,0x37,0x31,0xd7,0xba,0xe7,0x93,0x0b,
  0x16,0x4d,0xb3,0xc8,0x34,0x98,0x3c,0xd3,0x53,0xde,0x5e,0xe8,0x0c,0xbc,0xaf,0xc9,
  0x24,0x2c,0xcc,0xed,0xdb,0xde,0xba,0x1f,0x14,0x14,0x5a,0x95,0x80,0xde,0x66,0xbd
  }}};
#endif
```

10.1.8 KdfTestData.h

Hash Test Vectors

```
1   #define TEST_KDF_KEY_SIZE   20
2   TPM2B_TYPE(KDF_TEST_KEY, TEST_KDF_KEY_SIZE);
3   TPM2B_KDF_TEST_KEY      c_kdfTestKeyIn = {{TEST_KDF_KEY_SIZE, {
4       0x27, 0x1F, 0xA0, 0x8B, 0xBD, 0xC5, 0x06, 0x0E, 0xC3, 0xDF,
5       0xA9, 0x28, 0xFF, 0x9B, 0x73, 0x12, 0x3A, 0x12, 0xDA, 0x0C }}};
6   TPM2B_TYPE(KDF_TEST_LABEL, 17);
7   TPM2B_KDF_TEST_LABEL    c_kdfTestLabel = {{17, {
8       0x4B, 0x44, 0x46, 0x53, 0x45, 0x4C, 0x46, 0x54,
9       0x45, 0x53, 0x54, 0x4C, 0x41, 0x42, 0x45, 0x4C, 0x00 }}};
10  TPM2B_TYPE(KDF_TEST_CONTEXT, 8);
11  TPM2B_KDF_TEST_CONTEXT  c_kdfTestContextU = {{8, {
12      0xCE, 0x24, 0x4F, 0x39, 0x5D, 0xCA, 0x73, 0x91 }}};
13  TPM2B_KDF_TEST_CONTEXT  c_kdfTestContextV = {{8, {
14      0xDA, 0x50, 0x40, 0x31, 0xDD, 0xF1, 0x2E, 0x83 }}};
15  #if ALG_SHA512 == ALG_YES
16      TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
17          0x8b, 0xe2, 0xc1, 0xb8, 0x5b, 0x78, 0x56, 0x9b, 0x9f, 0xa7,
18          0x59, 0xf5, 0x85, 0x7c, 0x56, 0xd6, 0x84, 0x81, 0x0f, 0xd3 }}};
19      #define KDF_TEST_ALG    TPM_ALG_SHA512
20  #elif ALG_SHA384 == ALG_YES
21      TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
22          0x1d, 0xce, 0x70, 0xc9, 0x11, 0x3e, 0xb2, 0xdb, 0xa4, 0x7b,
23          0xd9, 0xcf, 0xc7, 0x2b, 0xf4, 0x6f, 0x45, 0xb0, 0x93, 0x12 }}};
24      #define KDF_TEST_ALG    TPM_ALG_SHA384
25  #elif ALG_SHA256 == ALG_YES
26      TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
27          0xbb, 0x02, 0x59, 0xe1, 0xc8, 0xba, 0x60, 0x7e, 0x6a, 0x2c,
28          0xd7, 0x04, 0xb6, 0x9a, 0x90, 0x2e, 0x9a, 0xde, 0x84, 0xc4 }}};
29      #define KDF_TEST_ALG    TPM_ALG_SHA256
30  #elif ALG_SHA1 == ALG_YES
31      TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
32          0x55, 0xb5, 0xa7, 0x18, 0x4a, 0xa0, 0x74, 0x23, 0xc4, 0x7d,
33          0xae, 0x76, 0x6c, 0x26, 0xa2, 0x37, 0x7d, 0x7c, 0xf8, 0x51 }}};
34      #define KDF_TEST_ALG    TPM_ALG_SHA1
35  #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  const TPM2B_RSA_TEST_VALUE    c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
57    0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x9a,0x07,0xb6,
58    0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
59    0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
60    0x93,0xb8,0x07,0x53,0xcc,0xdb,0x7f,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
61    0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,
```

```
  0x9a,0x76,0x71,0x2a,0x35,0x2a,0xa1,0x08,0xbb,0x95,0xe3,0x64,0xdc,0xdb,0xc2,0x33,
  0xa9,0x5f,0xbe,0x4c,0xc4,0xcc,0x28,0xc9,0x25,0xff,0xee,0x17,0x15,0x9a,0x50,0x90,
  0x0e,0x15,0xb4,0xea,0x6a,0x09,0xe6,0xff,0xa4,0xee,0xc7,0x7e,0xce,0xa9,0x73,0xe4,
  0xa0,0x56,0xbd,0x53,0x2a,0xe4,0xc0,0x2b,0xa8,0x9b,0x09,0x30,0x72,0x62,0x0f,0xf9,
  0xf6,0xa1,0x52,0xd2,0x8a,0x37,0xee,0xa5,0xc8,0x47,0xe1,0x99,0x21,0x47,0xeb,0xdd,
  0x37,0xaa,0xe4,0xbd,0x55,0x46,0x5a,0x5a,0x5d,0xfb,0x7b,0xfc,0xff,0xbf,0x26,0x71,
  0xf6,0x1e,0xad,0xbc,0xbf,0x33,0xca,0xe1,0x92,0x8f,0x2a,0x89,0x6c,0x45,0x24,0xd1,
  0xa6,0x52,0x56,0x24,0x5e,0x90,0x47,0xe5,0xcb,0x12,0xb0,0x32,0xf9,0xa6,0xbb,0xea,
  0x37,0xa9,0xbd,0xef,0x23,0xef,0x63,0x07,0x6c,0xc4,0x4e,0x64,0x3c,0xc6,0x11,0x84,
  0x7d,0x65,0xd6,0x5d,0x7a,0x17,0x58,0xa5,0xf7,0x74,0x3b,0x42,0xe3,0xd2,0xda,0x5f,
  0x6f,0xe0,0x1e,0x4b,0xcf,0x46,0xe2,0xdf,0x3e,0x41,0x8e,0x0e,0xb0,0x3f,0x8b,0x65}};
#define    OAEP_TEST_LABEL     "OAEP Test Value"
#if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
  0x32,0x68,0x84,0x0b,0x9c,0xc9,0x25,0x26,0xd9,0xc0,0xd0,0xb1,0xde,0x60,0x55,0xae,
  0x33,0xe5,0xcf,0x6c,0x85,0xbe,0x0d,0x71,0x11,0xe1,0x45,0x60,0xbb,0x42,0x3d,0xf3,
  0xb1,0x18,0x84,0x7b,0xc6,0x5d,0xce,0x1d,0x5f,0x9a,0x97,0xcf,0xb1,0x97,0x9a,0x85,
  0x7c,0xa7,0xa1,0x63,0x23,0xb6,0x74,0x0f,0x1a,0xee,0x29,0x51,0xeb,0x50,0x8f,0x3c,
  0x8e,0x4e,0x31,0x38,0xdc,0x11,0xfc,0x9a,0x4e,0xaf,0x93,0xc9,0x7f,0x6e,0x35,0xf3,
  0xc9,0xe4,0x89,0x14,0x53,0xe2,0xc2,0x1a,0xf7,0x6b,0x9b,0xf0,0x7a,0xa4,0x69,0x52,
  0xe0,0x24,0x8f,0xea,0x31,0xa7,0x5c,0x43,0xb0,0x65,0xc9,0xfe,0xba,0xfe,0x80,0x9e,
  0xa5,0xc0,0xf5,0x8d,0xce,0x41,0xf9,0x83,0x0d,0x8e,0x0f,0xef,0x3d,0x1f,0x6a,0xcc,
  0x8a,0x3d,0x3b,0xdf,0x22,0x38,0xd7,0x34,0x58,0x7b,0x55,0xc9,0xf6,0xbc,0x7c,0x4c,
  0x3f,0xd7,0xde,0x4e,0x30,0xa9,0x69,0xf3,0x5f,0x56,0x8f,0xc2,0xe7,0x75,0x79,0xb8,
  0xa5,0xc8,0x0d,0xc0,0xcd,0xb6,0xc9,0x63,0xad,0x7c,0xe4,0x8f,0x39,0x60,0x4d,0x7d,
  0xdb,0x34,0x49,0x2a,0x47,0xde,0xc0,0x42,0x4a,0x19,0x94,0x2e,0x50,0x21,0x03,0x47,
  0xff,0x73,0xb3,0xb7,0x89,0xcc,0x7b,0x2c,0xeb,0x03,0xa7,0x9a,0x06,0xfd,0xed,0x19,
  0xbb,0x82,0xa0,0x13,0xe9,0xfa,0xac,0x06,0x5f,0xc5,0xa9,0x2b,0xda,0x88,0x23,0xa2,
  0x5d,0xc2,0x7f,0xda,0xc8,0x5a,0x94,0x31,0xc1,0x21,0xd7,0x1e,0x6b,0xd7,0x89,0xb1,
  0x93,0x80,0xab,0xd1,0x37,0xf2,0x6f,0x50,0xcd,0x2a,0xea,0xb1,0xc4,0xcd,0xcb,0xb5}};
const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
  0x29,0xa4,0x2f,0xbb,0x8a,0x14,0x05,0x1e,0x3c,0x72,0x76,0x77,0x38,0xe7,0x73,0xe3,
  0x6e,0x24,0x4b,0x38,0xd2,0x1a,0xcf,0x23,0x58,0x78,0x36,0x82,0x23,0x6e,0x6b,0xef,
  0x2c,0x3d,0xf2,0xe8,0xd6,0xc6,0x87,0x8e,0x78,0x9b,0x27,0x39,0xc0,0xd6,0xef,0x4d,
  0x0b,0xfc,0x51,0x27,0x18,0xf3,0x51,0x5e,0x4d,0x96,0x3a,0xe2,0x15,0xe2,0x7e,0x42,
  0xf4,0x16,0xd5,0xc6,0x52,0x5d,0x17,0x44,0x76,0x09,0x7a,0xcf,0xe3,0x30,0xe3,0x84,
  0xf6,0x6f,0x3a,0x33,0xfb,0x32,0x0d,0x1d,0xe7,0x7c,0x80,0x82,0x4f,0xed,0xda,0x87,
  0x11,0x9c,0xc3,0x7e,0x85,0xbd,0x18,0x58,0x08,0x2b,0x23,0x37,0xe7,0x9d,0xd0,0xd1,
  0x79,0xe2,0x05,0xbd,0xf5,0x4f,0x0e,0x0f,0xdb,0x4a,0x74,0xeb,0x09,0x01,0xb3,0xca,
  0xbd,0xa6,0x7b,0x09,0xb1,0x13,0x77,0x30,0x4d,0x87,0x41,0x06,0x57,0x2e,0x5f,0x36,
  0x6e,0xfc,0x35,0x69,0xfe,0x0a,0x24,0x6c,0x98,0x8c,0xda,0x97,0xf4,0xfb,0xc7,0x83,
  0x2d,0x3e,0x7d,0xc0,0x5c,0x34,0xfd,0x11,0x2a,0x12,0xa7,0xae,0x4a,0xde,0xc8,0x4e,
  0xcf,0xf4,0x85,0x63,0x77,0xc6,0x33,0x34,0xe0,0x27,0xe4,0x9e,0x91,0x0b,0x4b,0x85,
  0xf0,0xb0,0x79,0xaa,0x7c,0xc6,0xff,0x3b,0xbc,0x04,0x73,0xb8,0x95,0xd7,0x31,0x54,
  0x3b,0x56,0xec,0x52,0x15,0xd7,0x3e,0x62,0xf5,0x82,0x99,0x3e,0x2a,0xc0,0x4b,0x2e,
  0x06,0x57,0x6d,0x3f,0x3e,0x77,0x1f,0x2b,0x2d,0xc5,0xb9,0x3b,0x68,0x56,0x73,0x70,
  0x32,0x6b,0x6b,0x65,0x25,0x76,0x45,0x6c,0x45,0xf1,0x6c,0x59,0xfc,0x94,0xa7,0x15}};
const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
  0x01,0xfe,0xd5,0x83,0x0b,0x15,0xba,0x90,0x2c,0xdf,0xf7,0x26,0xb7,0x8f,0xb1,0xd7,
  0x0b,0xfd,0x83,0xf9,0x95,0xd5,0xd7,0xb5,0xc5,0xc5,0x4a,0xde,0xd5,0xe6,0x20,0x78,
  0xca,0x73,0x77,0x3d,0x61,0x36,0x48,0xae,0x3e,0x8f,0xee,0x43,0x29,0x96,0xdf,0x3f,
  0x1c,0x97,0x5a,0xbe,0xe5,0xa2,0x7e,0x5b,0xd0,0xc0,0x29,0x39,0x83,0x81,0x77,0x24,
  0x43,0xdb,0x3c,0x64,0x4d,0xf0,0x23,0xe4,0xae,0x0f,0x78,0x31,0x8c,0xda,0x0c,0xec,
  0xf1,0xdf,0x09,0xf2,0x14,0x6a,0x4d,0xaf,0x36,0x81,0x6e,0xbd,0xbe,0x36,0x79,0x88,
  0x98,0xb6,0x6f,0x5a,0xad,0xcf,0x7c,0xee,0xe0,0xdd,0x00,0xbe,0x59,0x97,0x88,0x00,
  0x34,0xc0,0x8b,0x48,0x42,0x05,0x04,0x5a,0xb7,0x85,0x38,0xa0,0x35,0xd7,0x3b,0x51,
  0xb8,0x7b,0x81,0x83,0xee,0xff,0x76,0x6f,0x50,0x39,0x4d,0xab,0x89,0x63,0x07,0x6d,
  0xf5,0xe5,0x01,0x10,0x56,0xfe,0x93,0x06,0x8f,0xd3,0xc9,0x41,0xab,0xc9,0xdf,0x6e,
  0x59,0xa8,0xc3,0x1d,0xbf,0x96,0x4a,0x59,0x80,0x3c,0x90,0x3a,0x59,0x56,0x4c,0x6d,
  0x44,0x6d,0xeb,0xdc,0x73,0xcd,0xc1,0xec,0xb8,0x41,0xbf,0x89,0x8c,0x03,0x69,0x4c,
  0xaf,0x3f,0xc1,0xc5,0xc7,0xe7,0x7d,0xa7,0x83,0x39,0x70,0xa2,0x6b,0x83,0xbc,0xbe,
  0xf5,0xbf,0x1c,0xee,0x6e,0xa3,0x22,0x1e,0x25,0x2f,0x16,0x68,0x69,0x5a,0x1d,0xfa,
  0x2c,0x3a,0x0f,0x67,0xe1,0x77,0x12,0xe8,0x3d,0xba,0xaa,0xef,0x96,0x9c,0x1f,0x64,
  0x32,0xf4,0xa7,0xb3,0x3f,0x7d,0x61,0xbb,0x9a,0x27,0xad,0xfb,0x2f,0x33,0xc4,0x70}};
const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
  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_SHA256_VALUE == DEFAULT_TEST_HASH
145   const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
146     0x33,0x20,0x6e,0x21,0xc3,0xf6,0xcd,0xf8,0xd7,0x5d,0x9f,0xe9,0x05,0x14,0x8c,0x7c,
147     0xbb,0x69,0x24,0x9e,0x52,0x8f,0xaf,0x84,0x73,0x21,0x2c,0x85,0xa5,0x30,0x4d,0xb6,
148     0xb8,0xfa,0x15,0x9b,0xc7,0x8f,0xc9,0x7a,0x72,0x4b,0x85,0xa4,0x1c,0xc5,0xd8,0xe4,
149     0x92,0xb3,0xec,0xd9,0xa8,0xca,0x5e,0x74,0x73,0x89,0x7f,0xb4,0xac,0x7e,0x68,0x12,
150     0xb2,0x53,0x27,0x4b,0xbf,0xd0,0x71,0x69,0x46,0x9f,0xef,0xf4,0x70,0x60,0xf8,0xd7,
151     0xae,0xc7,0x5a,0x27,0x38,0x25,0x2d,0x25,0xab,0x96,0x56,0x66,0x3a,0x23,0x40,0xa8,
152     0xdb,0xbc,0x86,0xe8,0xf3,0xd2,0x58,0x0b,0x44,0xfc,0x94,0x1e,0xb7,0x5d,0xb4,0x57,
153     0xb5,0xf3,0x56,0xee,0x9b,0xcf,0x97,0x91,0x29,0x36,0xe3,0x06,0x13,0xa2,0xea,0xd6,
154     0xd6,0x0b,0x86,0x0b,0x1a,0x27,0xe6,0x22,0xc4,0x7b,0xff,0xde,0x0f,0xbf,0x79,0xc8,
155     0x1b,0xed,0xf1,0x27,0x62,0xb5,0x8b,0xf9,0xd9,0x76,0x90,0xf6,0xcc,0x83,0x0f,0xce,
156     0xce,0x2e,0x63,0x7a,0x9b,0xf4,0x48,0x5b,0xd7,0x81,0x2c,0x3a,0xdb,0x59,0x0d,0x4d,
157     0x9e,0x46,0xe9,0x9e,0x92,0x22,0x27,0x1c,0xb0,0x67,0x8a,0xe6,0x8a,0x16,0x8a,0xdf,
158     0x95,0x76,0x24,0x82,0xad,0xf1,0xbc,0x97,0xbf,0xd3,0x5e,0x6e,0x14,0x0c,0x5b,0x25,
159     0xfe,0x58,0xfa,0x64,0xe5,0x14,0x46,0xb7,0x58,0xc6,0x3f,0x7f,0x42,0xd2,0x8e,0x45,
160     0x13,0x41,0x85,0x12,0x2e,0x96,0x19,0xd0,0x5e,0x7d,0x34,0x06,0x32,0x2b,0xc8,0xd9,
161     0x0d,0x6c,0x06,0x36,0xa0,0xff,0x47,0x57,0x2c,0x25,0xbc,0x8a,0xa5,0xe2,0xc7,0xe3}};
162   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
163     0x39,0xfc,0x10,0x5d,0xf4,0x45,0x3d,0x94,0x53,0x06,0x89,0x24,0xe7,0xe8,0xfd,0x03,
164     0xac,0xfd,0xbd,0xb2,0x28,0xd3,0x4a,0x52,0xc5,0xd4,0xdb,0x17,0xd4,0x24,0x05,0xc4,
165     0xeb,0x6a,0xce,0x1d,0xbb,0x37,0xcb,0x09,0xd8,0x6c,0x83,0x19,0x93,0xd4,0xe2,0x88,
166     0x88,0x9b,0xaf,0x92,0x16,0xc4,0x15,0xbd,0x49,0x13,0x22,0xb7,0x84,0xcf,0x23,0xf2,
167     0x6f,0x0c,0x3e,0x8f,0xde,0x04,0x09,0x31,0x2d,0x99,0xdf,0xe6,0x74,0x70,0x30,0xde,
168     0x8c,0xad,0x32,0x86,0xe2,0x7c,0x12,0x90,0x21,0xf3,0x86,0xb7,0xe2,0x64,0xca,0x98,
169     0xcc,0x64,0x4b,0xef,0x57,0x4f,0x5a,0x16,0x6e,0xd7,0x2f,0x5b,0xf6,0x07,0xad,0x33,
170     0xb4,0x8f,0x3b,0x3a,0x8b,0xd9,0x06,0x2b,0xed,0x3c,0x3c,0x76,0xf6,0x21,0x31,0xe3,
171     0xfb,0x2c,0x45,0x61,0x42,0xba,0xe0,0xc3,0x72,0x63,0xd0,0x6b,0x8f,0x36,0x26,0xfb,
172     0x9e,0x89,0x0e,0x44,0x9a,0xc1,0x84,0x5e,0x84,0x8d,0xb6,0xea,0xf1,0x0d,0x66,0xc7,
173     0xdb,0x44,0xbd,0x19,0x7c,0x05,0xbe,0xc4,0xab,0x88,0x32,0xbe,0xc7,0x63,0x31,0xe6,
174     0x38,0xd4,0xe5,0xb8,0x4b,0xf5,0x0e,0x55,0x9a,0x3a,0xe6,0x0a,0xec,0xee,0xe2,0xa8,
175     0x88,0x04,0xf2,0xb8,0xaa,0x5a,0xd8,0x97,0x5d,0xa0,0xa8,0x42,0xfb,0xd9,0xde,0x80,
176     0xae,0x4c,0xb3,0xa1,0x90,0x47,0x57,0x03,0x10,0x78,0xa6,0x8f,0x11,0xba,0x4b,0xce,
177     0x2d,0x56,0xa4,0xe1,0xbd,0xf8,0xa0,0xa4,0xd5,0x48,0x3c,0x63,0x20,0x00,0x38,0xa0,
178     0xd1,0xe6,0x12,0xe9,0x1d,0xd8,0x49,0xe3,0xd5,0x24,0xb5,0xc5,0x3a,0x1f,0xb0,0xd4}};
179   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
180     0x74,0x89,0x29,0x3e,0x1b,0xac,0xc6,0x85,0xca,0xf0,0x63,0x43,0x30,0x7d,0x1c,0x9b,
181     0x2f,0xbd,0x4d,0x69,0x39,0x5e,0x85,0xe2,0xef,0x86,0x0a,0xc6,0x6b,0xa6,0x08,0x19,
182     0x6c,0x56,0x38,0x24,0x55,0x92,0x84,0x9b,0x1b,0x8b,0x04,0xcf,0x24,0x14,0x24,0x13,
183     0x0e,0x8b,0x82,0x6f,0x96,0xc8,0x9a,0x68,0xfc,0x4c,0x02,0xf0,0xdc,0xcd,0x36,0x25,
184     0x31,0xd5,0x82,0xcf,0xc9,0x69,0x72,0xf6,0x1d,0xab,0x68,0x20,0x2e,0x2d,0x19,0x49,
185     0xf0,0x2e,0xad,0xd2,0xda,0xaf,0xff,0xb6,0x92,0x83,0x5b,0x8a,0x06,0x2d,0x0c,0x32,
186     0x11,0x32,0x3b,0x77,0x17,0xf6,0x50,0xfb,0xf8,0x57,0xc9,0xc7,0x9b,0x9e,0xc6,0xd1,
187     0xa9,0x55,0xf0,0x22,0x35,0xda,0xca,0x3c,0x8e,0xc6,0x9a,0xd8,0x25,0xc8,0x5e,0x93,
188     0x0d,0xaa,0xa7,0x06,0xaf,0x11,0x29,0x99,0xe7,0x7c,0xee,0x49,0x82,0x30,0xba,0x2c,
189     0xe2,0x40,0x8f,0x0a,0xa6,0x7b,0x24,0x75,0xc5,0xcd,0x03,0x12,0xf4,0xb2,0x4b,0x3a,
190     0xd1,0x91,0x3c,0x20,0x0e,0x58,0x2b,0x31,0xf8,0x8b,0xee,0xbc,0x1f,0x95,0x35,0x58,
191     0x6a,0x73,0xee,0x99,0xb0,0x01,0x42,0x4f,0x66,0xc0,0x66,0xbb,0x35,0x86,0xeb,0xd9,
192     0x7b,0x55,0x77,0x2d,0x54,0x78,0x19,0x49,0xe8,0xcc,0xfd,0xb1,0xcb,0x49,0xc9,0xea,
193     0x20,0xab,0xed,0xb5,0xed,0xfe,0xb2,0xb5,0xa8,0xcf,0x05,0x06,0xd5,0x7d,0x2b,0xbb,
```

```
194     0x0b,0x65,0x6b,0x2b,0x6d,0x55,0x95,0x85,0x44,0x8b,0x12,0x05,0xf3,0x4b,0xd4,0x8e,
195     0x3d,0x68,0x2d,0x29,0x9c,0x05,0x79,0xd6,0xfc,0x72,0x90,0x6a,0xab,0x46,0x38,0x81}};
196   const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
197     0x8a,0xb1,0x0a,0xb5,0xe4,0x02,0xf7,0xdd,0x45,0x2a,0xcc,0x2b,0x6b,0x8c,0x0e,0x9a,
198     0x92,0x4f,0x9b,0xc5,0xe4,0x8b,0x82,0xb9,0xb0,0xd9,0x87,0x8c,0xcb,0xf0,0xb0,0x59,
199     0xa5,0x92,0x21,0xa0,0xa7,0x61,0x5c,0xed,0xa8,0x6e,0x22,0x29,0x46,0xc7,0x86,0x37,
200     0x4b,0x1b,0x1e,0x94,0x93,0xc8,0x4c,0x17,0x7a,0xae,0x59,0x91,0xf8,0x83,0x84,0xc4,
201     0x8c,0x38,0xc2,0x35,0x0e,0x7e,0x50,0x67,0x76,0xe7,0xd3,0xec,0x6f,0x0d,0xa0,0x5c,
202     0x2f,0x0a,0x80,0x28,0xd3,0xc5,0x7d,0x2d,0x1a,0x0b,0x96,0xd6,0xe5,0x98,0x05,0x8c,
203     0x4d,0xa0,0x1f,0x8c,0xb6,0xfb,0xb1,0xcf,0xe9,0xcb,0x38,0x27,0x60,0x64,0x17,0xca,
204     0xf4,0x8b,0x61,0xb7,0x1d,0xb6,0x20,0x9d,0x40,0x2a,0x1c,0xfd,0x55,0x40,0x4b,0x95,
205     0x39,0x52,0x18,0x3b,0xab,0x44,0xe8,0x83,0x4b,0x7c,0x47,0xfb,0xed,0x06,0x9c,0xcd,
206     0x4f,0xba,0x81,0xd6,0xb7,0x31,0xcf,0x5c,0x23,0xf8,0x25,0xab,0x95,0x77,0x0a,0x8f,
207     0x46,0xef,0xfb,0x59,0xb8,0x04,0xd7,0x1e,0xf5,0xaf,0x6a,0x1a,0x26,0x9b,0xae,0xf4,
208     0xf5,0x7f,0x84,0x6f,0x3c,0xed,0xf8,0x24,0x0b,0x43,0xd1,0xba,0x74,0x89,0x4e,0x39,
209     0xfe,0xab,0xa5,0x16,0xa5,0x28,0xee,0x96,0x84,0x3e,0x16,0x6d,0x5f,0x4e,0x0b,0x7d,
210     0x94,0x16,0x1b,0x8c,0xf9,0xaa,0x9b,0xc0,0x49,0x02,0x4c,0x3e,0x62,0xff,0xfe,0xa2,
211     0x20,0x33,0x5e,0xa6,0xdd,0xda,0x15,0x2d,0xb7,0xcd,0xda,0xff,0xb1,0x0b,0x45,0x7b,
212     0xd3,0xa0,0x42,0x29,0xab,0xa9,0x73,0xe9,0xa4,0xd9,0x8d,0xac,0xa1,0x88,0x2c,0x2d}};
213   #endif // SHA256
214   #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
215   const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
216     0x0f,0x3c,0x42,0x4d,0x8c,0x91,0x96,0x05,0x3c,0xfd,0x59,0x3b,0x7f,0x29,0xbc,0x03,
217     0x67,0xc1,0xff,0x74,0xe7,0x09,0xf4,0x13,0x45,0xbe,0x13,0x1d,0xc9,0x86,0x94,0xfe,
218     0xed,0xa6,0xe8,0x3a,0xcb,0x89,0x4d,0xec,0x86,0x63,0x4c,0xdb,0xf1,0x95,0xee,0xc1,
219     0x46,0xc5,0x3b,0xd8,0xf8,0xa2,0x41,0x6a,0x60,0x8b,0x9e,0x5e,0x7f,0x20,0x16,0xe3,
220     0x69,0xb6,0x2d,0x92,0xfc,0x60,0xa2,0x74,0x88,0xd5,0xc7,0xa6,0xd1,0xff,0xe3,0x45,
221     0x02,0x51,0x39,0xd9,0xf3,0x56,0x0b,0x91,0x80,0xe0,0x6c,0xa8,0xc3,0x78,0xef,0x34,
222     0x22,0x8c,0xf5,0xfb,0x47,0x98,0x5d,0x57,0x8e,0x3a,0xb9,0xff,0x92,0x04,0xc7,0xc2,
223     0x6e,0xfa,0x14,0xc1,0xb9,0x68,0x15,0x5c,0x12,0xe8,0xa8,0xbe,0xea,0xe8,0x8d,0x9b,
224     0x48,0x28,0x35,0xdb,0x4b,0x52,0xc1,0x2d,0x85,0x47,0x83,0xd0,0xe9,0xae,0x90,0x6e,
225     0x65,0xd4,0x34,0x7f,0x81,0xce,0x69,0xf0,0x96,0x62,0xf7,0xec,0x41,0xd5,0xc2,0xe3,
226     0x4b,0xba,0x9c,0x8a,0x02,0xce,0xf0,0x5d,0x14,0xf7,0x09,0x42,0x8e,0x4a,0x27,0xfe,
227     0x3e,0x66,0x42,0x99,0x03,0xe1,0x69,0xbd,0xdb,0x7f,0x9b,0x70,0xeb,0x4e,0x9c,0xac,
228     0x45,0x67,0x91,0x9f,0x75,0x10,0xc6,0xfc,0x14,0xe1,0x28,0xc1,0x0e,0xe0,0x7e,0xc0,
229     0x5c,0x1d,0xee,0xe8,0xff,0x45,0x79,0x51,0x86,0x08,0xe6,0x39,0xac,0xb5,0xfd,0xb8,
230     0xf1,0xdd,0x2e,0xf4,0xb2,0x1a,0x69,0x0d,0xd9,0x98,0x8e,0xdb,0x85,0x61,0x70,0x20,
231     0x82,0x91,0x26,0x87,0x80,0xc4,0x6a,0xd8,0x3b,0x91,0x4d,0xd3,0x33,0x84,0xad,0xb7}};
232   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
233     0x44,0xd5,0x9f,0xbc,0x48,0x03,0x3d,0x9f,0x22,0x91,0x2a,0xab,0x3c,0x31,0x71,0xab,
234     0x86,0x3f,0x0f,0x6f,0x59,0x5b,0x93,0x27,0xbc,0xbc,0xcd,0x29,0x38,0x43,0x2a,0x3b,
235     0x3b,0xd2,0xb3,0x45,0x40,0xba,0x15,0xb4,0x45,0xe3,0x56,0xab,0xff,0xb3,0x20,0x26,
236     0x39,0xcc,0x48,0xc5,0x5d,0x41,0x0d,0x2f,0x57,0x7f,0x9d,0x16,0x2e,0x26,0x57,0xc7,
237     0x6b,0xf3,0x36,0x54,0xbd,0xb6,0x1d,0x46,0x4e,0x13,0x50,0xd7,0x61,0x9d,0x8d,0x7b,
238     0xeb,0x21,0x9f,0x79,0xf3,0xfd,0xe0,0x1b,0xa8,0xed,0x6d,0x29,0x33,0x0d,0x65,0x94,
239     0x24,0x1e,0x62,0x88,0x6b,0x2b,0x4e,0x39,0xf5,0x80,0x39,0xca,0x76,0x95,0xbc,0x7c,
240     0x27,0x1d,0xdd,0x3a,0x11,0xf1,0x3e,0x54,0x03,0xb7,0x43,0x91,0x99,0x33,0xfe,0x9d,
241     0x14,0x2c,0x87,0x9a,0x95,0x18,0x1f,0x02,0x04,0x6a,0xe2,0xb7,0x81,0x14,0x13,0x45,
242     0x16,0xfb,0xe4,0xb7,0x8f,0xab,0x2b,0xd7,0x60,0x34,0x8a,0x55,0xbc,0x01,0x8c,0x49,
243     0x02,0x29,0xf1,0x9c,0x94,0x98,0x44,0xd0,0x94,0xcb,0xd4,0x85,0x4c,0x3b,0x77,0x72,
244     0x99,0xd5,0x4b,0xc6,0x3b,0xe4,0xd2,0xc8,0xe9,0x6a,0x23,0x18,0x3b,0x3b,0x5e,0x32,
245     0xec,0x70,0x84,0x5d,0xbb,0x6a,0x8f,0x0c,0x5f,0x55,0xa5,0x30,0x34,0x48,0xbb,0xc2,
246     0xdf,0x12,0xb9,0x81,0xad,0x36,0x3f,0xf0,0x24,0x16,0x48,0x04,0x4a,0x7f,0xfd,0x9f,
247     0x4c,0xea,0xfe,0x1d,0x83,0xd0,0x81,0xad,0x25,0x6c,0x5f,0x45,0x36,0x91,0xf0,0xd5,
248     0x8b,0x53,0x0a,0xdf,0xec,0x9f,0x04,0x58,0xc4,0x35,0xa0,0x78,0x1f,0x68,0xe0,0x22}};
249   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
250     0x3f,0x3a,0x82,0x6d,0x42,0xe3,0x8b,0x4f,0x45,0x9c,0xda,0x6c,0xbe,0xbe,0xcd,0x00,
251     0x98,0xfb,0xbe,0x59,0x30,0xc6,0x3c,0xaa,0xb3,0x06,0x27,0xb5,0xda,0xfa,0xb2,0xc3,
252     0x43,0xb7,0xbd,0xe9,0xd3,0x23,0xed,0x80,0xce,0x74,0xb3,0xb8,0x77,0x8d,0xe6,0x8d,
253     0x3c,0xe5,0xf5,0xd7,0x80,0xcf,0x38,0x55,0x76,0xd7,0x87,0xa8,0xd6,0x3a,0xcf,0xfd,
254     0xd8,0x91,0x65,0xab,0x43,0x66,0x50,0xb7,0x9a,0x13,0x6b,0x45,0x80,0x76,0x86,0x22,
255     0x27,0x72,0xf7,0xbb,0x65,0x22,0x5c,0x55,0x60,0xd8,0x84,0x9f,0xf2,0x61,0x52,0xac,
256     0xf2,0x4f,0x5b,0x7b,0x21,0xe1,0xf5,0x4b,0x8f,0x01,0xf2,0x4b,0xcf,0xd3,0xfb,0x74,
257     0x5e,0x6e,0x96,0xb4,0xa8,0x0f,0x01,0x9b,0x26,0x54,0x0a,0x70,0x55,0x26,0xb7,0x0b,
258     0xe8,0x01,0x68,0x66,0x0d,0x6f,0xb5,0xfc,0x66,0xbd,0x9e,0x44,0xed,0x6a,0x1e,0x3c,
259     0x3b,0x61,0x5d,0xe8,0xdb,0x99,0x5b,0x67,0xbf,0x94,0xfb,0xe6,0x8c,0x4b,0x07,0xcb,
```



```
260     0x43,0x3a,0x0d,0xb1,0x1b,0x10,0x66,0x81,0xe2,0x0d,0xe7,0xd1,0xca,0x85,0xa7,0x50,
261     0x82,0x2d,0xbf,0xed,0xcf,0x43,0x6d,0xdb,0x2c,0x7b,0x73,0x20,0xfe,0x73,0x3f,0x19,
262     0xc6,0xdb,0x69,0xb8,0xc3,0xd3,0xf4,0xe5,0x64,0xf8,0x36,0x8e,0xd5,0xd8,0x09,0x2a,
263     0x5f,0x26,0x70,0xa1,0xd9,0x5b,0x14,0xf8,0x22,0xe9,0x9d,0x22,0x51,0xf4,0x52,0xc1,
264     0x6f,0x53,0xf5,0xca,0x0d,0xda,0x39,0x8c,0x29,0x42,0xe8,0x58,0x89,0xbb,0xd1,0x2e,
265     0xc5,0xdb,0x86,0x8d,0xaf,0xec,0x58,0x36,0x8d,0x8d,0x57,0x23,0xd5,0xdd,0xb9,0x24}};
266   const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
267     0x39,0x10,0x58,0x7d,0x6d,0xa8,0xd5,0x90,0x07,0xd6,0x2b,0x13,0xe9,0xd8,0x93,0x7e,
268     0xf3,0x5d,0x71,0xe0,0xf0,0x33,0x3a,0x4a,0x22,0xf3,0xe6,0x95,0xd3,0x8e,0x8c,0x41,
269     0xe7,0xb3,0x13,0xde,0x4a,0x45,0xd3,0xd1,0xfb,0xb1,0x3f,0x9b,0x39,0xa5,0x50,0x58,
270     0xef,0xb6,0x3a,0x43,0xdd,0x54,0xab,0xda,0x9d,0x32,0x49,0xe4,0x57,0x96,0xe5,0x1b,
271     0x1d,0x8f,0x33,0x8e,0x07,0x67,0x56,0x14,0xc1,0x18,0x78,0xa2,0x52,0xe6,0x2e,0x07,
272     0x81,0xbe,0xd8,0xca,0x76,0x63,0x68,0xc5,0x47,0xa2,0x92,0x5e,0x4c,0xfd,0x14,0xc7,
273     0x46,0x14,0xbe,0xc7,0x85,0xef,0xe6,0xb8,0x46,0xcb,0x3a,0x67,0x66,0x89,0xc6,0xee,
274     0x9d,0x64,0xf5,0x0d,0x09,0x80,0x9a,0x6f,0x0e,0xeb,0xe4,0xb9,0xe9,0xab,0x90,0x4f,
275     0xe7,0x5a,0xc8,0xca,0xf6,0x16,0x0a,0x82,0xbd,0xb7,0x76,0x59,0x08,0x2d,0xd9,0x40,
276     0x5d,0xaa,0xa5,0xef,0xfb,0xe3,0x81,0x2c,0x2c,0x5c,0xa8,0x16,0xbd,0x63,0x20,0xc2,
277     0x4d,0x3b,0x51,0xaa,0x62,0x1f,0x06,0xe5,0xbb,0x78,0x44,0x04,0x0c,0x5c,0xe1,0x1b,
278     0x6b,0x9d,0x21,0x10,0xaf,0x48,0x48,0x98,0x97,0x77,0xc2,0x73,0xb4,0x98,0x64,0xcc,
279     0x94,0x2c,0x29,0x28,0x45,0x36,0xd1,0xc5,0xd0,0x2f,0x97,0x27,0x92,0x65,0x22,0xbb,
280     0x63,0x79,0xea,0xf5,0xff,0x77,0x0f,0x4b,0x56,0x8a,0x9f,0xad,0x1a,0x97,0x67,0x39,
281     0x69,0xb8,0x4c,0x6c,0xc2,0x56,0xc5,0x7a,0xa8,0x14,0x5a,0x24,0x7a,0xa4,0x6e,0x55,
282     0xb2,0x86,0x1d,0xf4,0x62,0x5a,0x2d,0x87,0x6d,0xde,0x99,0x78,0x2d,0xef,0xd7,0xdc}};
283   #endif // SHA384
284   #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
285   const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
286     0x48,0x45,0xa7,0x70,0xb2,0x41,0xb7,0x48,0x5e,0x79,0x8c,0xdf,0x1c,0xc6,0x7e,0xbb,
287     0x11,0x80,0x82,0x52,0xbf,0x40,0x3d,0x90,0x03,0x6e,0x20,0x3a,0xb9,0x65,0xc8,0x51,
288     0x4c,0xbd,0x9c,0xa9,0x43,0x89,0xd0,0x57,0x0c,0xa3,0x69,0x22,0x7e,0x82,0x2a,0x1c,
289     0x1d,0x5a,0x80,0x84,0x81,0xbb,0x5e,0x5e,0xd0,0xc1,0x66,0x9a,0xac,0x00,0xba,0x14,
290     0xa2,0xe9,0xd0,0x3a,0x89,0x5a,0x63,0xe2,0xec,0x92,0x05,0xf4,0x47,0x66,0x12,0x7f,
291     0xdb,0xa7,0x3c,0x5b,0x67,0xe1,0x55,0xca,0x0a,0x27,0xbf,0x39,0x89,0x11,0x05,0xba,
292     0x9b,0x5a,0x9b,0x65,0x44,0xad,0x78,0xcf,0x8f,0x94,0xf6,0x9a,0xb4,0x52,0x39,0x0e,
293     0x00,0xba,0xbc,0xe0,0xbd,0x6f,0x81,0x2d,0x76,0x42,0x66,0x70,0x07,0x77,0xbf,0x09,
294     0x88,0x2a,0x0c,0xb1,0x56,0x3e,0xee,0xfd,0xdc,0xb6,0x3c,0x0d,0xc5,0xa4,0x0d,0x10,
295     0x32,0x80,0x3e,0x1e,0xfe,0x36,0x8f,0xb5,0x42,0xc1,0x21,0x7b,0xdf,0xdf,0x4a,0xd2,
296     0x68,0x0c,0x01,0x9f,0x4a,0xfd,0xd4,0xec,0xf7,0x49,0x06,0xab,0xed,0xc6,0xd5,0x1b,
297     0x63,0x76,0x38,0xc8,0x6c,0xc7,0x4f,0xcb,0x29,0x8a,0x0e,0x6f,0x33,0xaf,0x69,0x31,
298     0x8e,0xa7,0xdd,0x9a,0x36,0xde,0x9b,0xf1,0x0b,0xfb,0x20,0xa0,0x6d,0x33,0x31,0xc9,
299     0x9e,0xb4,0x2e,0xc5,0x40,0x0e,0x60,0x71,0x36,0x75,0x05,0xf9,0x37,0xe0,0xca,0x8e,
300     0x8f,0x56,0xe0,0xea,0x9b,0xeb,0x17,0xf3,0xca,0x40,0xc3,0x48,0x01,0xba,0xdc,0xc6,
301     0x4b,0x2b,0x5b,0x7b,0x5c,0x81,0xa6,0xbb,0xc7,0x43,0xc0,0xbe,0xc0,0x30,0x7b,0x55}};
302   const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
303     0x74,0x83,0xfa,0x52,0x65,0x50,0x68,0xd0,0x82,0x05,0x72,0x70,0x78,0x1c,0xac,0x10,
304     0x23,0xc5,0x07,0xf8,0x93,0xd2,0xeb,0x65,0x87,0xbb,0x47,0xc2,0xfb,0x30,0x9e,0x61,
305     0x4c,0xac,0x04,0x57,0x5a,0x7c,0xeb,0x29,0x08,0x84,0x86,0x89,0x1e,0x8f,0x07,0x32,
306     0xa3,0x8b,0x70,0xe7,0xa2,0x9f,0x9c,0x42,0x71,0x3d,0x23,0x59,0x82,0x5e,0x8a,0xde,
307     0xd6,0xfb,0xd8,0xc5,0x8b,0xc0,0xdb,0x10,0x38,0x87,0xd3,0xbf,0x04,0xb0,0x66,0xb9,
308     0x85,0x81,0x54,0x4c,0x69,0xdc,0xba,0x78,0xf3,0x4a,0xdb,0x25,0xa2,0xf2,0x34,0x55,
309     0xdd,0xaa,0xa5,0xc4,0xed,0x55,0x06,0x0e,0x2a,0x30,0x77,0xab,0x82,0x79,0xf0,0xcd,
310     0x9d,0x6f,0x09,0xa0,0xc8,0x82,0xc9,0xe0,0x61,0xda,0x40,0xcd,0x17,0x59,0xc0,0xef,
311     0x95,0x6d,0xa3,0x6d,0x1c,0x2b,0xee,0x24,0xef,0xd8,0x4a,0x55,0x6c,0xd6,0x26,0x42,
312     0x32,0x17,0xfd,0x6a,0xb3,0x4f,0xde,0x07,0x2f,0x10,0xd4,0xac,0x14,0xea,0x89,0x68,
313     0xcc,0xd3,0x07,0xb7,0xcf,0xba,0x39,0x20,0x63,0x20,0x7b,0x44,0x8b,0x48,0x60,0x5d,
314     0x3a,0x2a,0x0a,0xe9,0x68,0xab,0x15,0x46,0x27,0x64,0xb5,0x82,0x06,0x29,0xe7,0x25,
315     0xca,0x46,0x48,0x6e,0x2a,0x34,0x57,0x4b,0x81,0x75,0xae,0xb6,0xfd,0x6f,0x51,0x5f,
316     0x04,0x59,0xc7,0x15,0x1f,0xe0,0x68,0xf7,0x36,0x2d,0xdf,0xc8,0x9d,0x05,0x27,0x2d,
317     0x3f,0x2b,0x59,0x5d,0xcb,0xf3,0xc4,0x92,0x6e,0x00,0xa8,0x8d,0xd0,0x69,0xe5,0x59,
318     0xda,0xba,0x4f,0x38,0xf5,0xa0,0x8b,0xf1,0x73,0xe9,0x0d,0xee,0x64,0xe5,0xa2,0xd8}};
319   const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
320     0x1b,0xca,0x8b,0x18,0x15,0x3b,0x95,0x5b,0x0a,0x89,0x10,0x03,0x7f,0x7c,0xa0,0xc9,
321     0x66,0x57,0x86,0x6a,0xc9,0xeb,0x82,0x71,0xf3,0x8d,0x6f,0xa9,0xa4,0x2d,0xd0,0x22,
322     0xdf,0xe9,0xc6,0x71,0x5b,0xf4,0x27,0x38,0x5b,0x2c,0x8a,0x54,0xcc,0x85,0x11,0x69,
323     0x6d,0x6f,0x42,0xe7,0x22,0xcb,0xd6,0xad,0x1a,0xc5,0xab,0x6a,0xa5,0xfc,0xa5,0x70,
324     0x72,0x4a,0x62,0x25,0xd0,0xa2,0x16,0x61,0xab,0xac,0x31,0xa0,0x46,0x24,0x4f,0xdd,
325     0x9a,0x36,0x55,0xb6,0x00,0x9e,0x23,0x50,0x0d,0x53,0x01,0xb3,0x46,0x56,0xb2,0x1d,
```

```
326     0x33,0x5b,0xca,0x41,0x7f,0x65,0x7e,0x00,0x5c,0x12,0xff,0x0a,0x70,0x5d,0x8c,0x69,
327     0x4a,0x02,0xee,0x72,0x30,0xa7,0x5c,0xa4,0xbb,0xbe,0x03,0x0c,0xe4,0x5f,0x33,0xb6,
328     0x78,0x91,0x9d,0xd8,0xec,0x34,0x03,0x2e,0x63,0x32,0xc7,0x2a,0x36,0x50,0xd5,0x8b,
329     0x0e,0x7f,0x54,0x4e,0xf4,0x29,0x11,0x1b,0xcd,0x0f,0x37,0xa5,0xbc,0x61,0x83,0x50,
330     0xfa,0x18,0x75,0xd9,0xfe,0xa7,0xe8,0x9b,0xc1,0x4f,0x96,0x37,0x81,0x71,0xdf,0x71,
331     0x8b,0x89,0x81,0xf4,0x95,0xb5,0x29,0x66,0x41,0x0c,0x73,0xd7,0x0b,0x21,0xb4,0xfb,
332     0xf9,0x63,0x2f,0xe9,0x7b,0x38,0xaa,0x20,0xc3,0x96,0xcc,0xb7,0xb2,0x24,0xa1,0xe0,
333     0x59,0x9c,0x10,0x9e,0x5a,0xf7,0xe3,0x02,0xe6,0x23,0xe2,0x44,0x21,0x3f,0x6e,0x5e,
334     0x79,0xb2,0x93,0x7d,0xce,0xed,0xe2,0xe1,0xab,0x98,0x07,0xa7,0xbd,0xbc,0xd8,0xf7,
335     0x06,0xeb,0xc5,0xa6,0x37,0x18,0x11,0x88,0xf7,0x63,0x39,0xb9,0x57,0x29,0xdc,0x03}};
336   const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
337     0x05,0x55,0x00,0x62,0x01,0xc6,0x04,0x31,0x55,0x73,0x3f,0x2a,0xf9,0xd4,0x0f,0xc1,
338     0x2b,0xeb,0xd8,0xc8,0xdb,0xb2,0xab,0x6c,0x26,0xde,0x2d,0x89,0xc2,0x2d,0x36,0x62,
339     0xc8,0x22,0x5d,0x58,0x03,0xb1,0x46,0x14,0xa5,0xd4,0xbc,0x25,0x6b,0x7f,0x8f,0x14,
340     0x7e,0x03,0x2f,0x3d,0xb8,0x39,0xa5,0x79,0x13,0x7e,0x22,0x2a,0xb9,0x3e,0x8f,0xaa,
341     0x01,0x7c,0x03,0x12,0x21,0x6c,0x2a,0xb4,0x39,0x98,0x6d,0xff,0x08,0x6c,0x59,0x2d,
342     0xdc,0xc6,0xf1,0x77,0x62,0x10,0xa6,0xcc,0xe2,0x71,0x8e,0x97,0x00,0x87,0x5b,0x0e,
343     0x20,0x00,0x3f,0x18,0x63,0x83,0xf0,0xe4,0x0a,0x64,0x8c,0xe9,0x8c,0x91,0xe7,0x89,
344     0x04,0x64,0x2c,0x8b,0x41,0xc8,0xac,0xf6,0x5a,0x75,0xe6,0xa5,0x76,0x43,0xcb,0xa5,
345     0x33,0x8b,0x07,0xc9,0x73,0x0f,0x45,0xa4,0xc3,0xac,0xc1,0xc3,0xe6,0xe7,0x21,0x66,
346     0x1c,0xba,0xbf,0xea,0x3e,0x39,0xfa,0xb2,0xe2,0x8f,0xfe,0x9c,0xb4,0x85,0x89,0x33,
347     0x2a,0x0c,0xc8,0x5d,0x58,0xe1,0x89,0x12,0xe9,0x4d,0x42,0xb3,0x1f,0x99,0x0c,0x3e,
348     0xd8,0xb2,0xeb,0xf5,0x88,0xfb,0xe1,0x4b,0x8e,0xdc,0xd3,0xa8,0xda,0xbe,0x04,0x45,
349     0xbf,0x56,0xc6,0x54,0x70,0x00,0xb8,0x66,0x46,0x3a,0xa3,0x1e,0xb6,0xeb,0x1a,0xa0,
350     0x0b,0xd3,0x9a,0x9a,0x52,0xda,0x60,0x69,0xb7,0xef,0x93,0x47,0x38,0xab,0x1a,0xa0,
351     0x22,0x6e,0x76,0x06,0xb6,0x74,0xaf,0x74,0x8f,0x51,0xc0,0x89,0x5a,0x4b,0xbe,0x6a,
352     0x91,0x18,0x25,0x7d,0xa6,0x77,0xe6,0xfd,0xc2,0x62,0x36,0x07,0xc6,0xef,0x79,0xc9}};
353   #endif // SHA512
```

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.

```
1   #if AES_128
2   const BYTE  key_AES128 [] = {
3                   0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
4                   0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
5   const BYTE  dataIn_AES128 [] = {
6                   0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
7                   0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
8                   0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
9                   0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
10  const BYTE  dataOut_AES128_ECB [] = {
11                  0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
12                  0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97,
13                  0xf5, 0xd3, 0xd5, 0x85, 0x03, 0xb9, 0x69, 0x9d,
14                  0xe7, 0x85, 0x89, 0x5a, 0x96, 0xfd, 0xba, 0xaf};
15  const BYTE  dataOut_AES128_CBC [] = {
16                  0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46,
17                  0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
18                  0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee,
19                  0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2};
20  const BYTE  dataOut_AES128_CFB [] = {
21                  0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
22                  0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
23                  0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
24                  0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b};
25  const BYTE  dataOut_AES128_OFB [] = {
26                  0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
27                  0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
28                  0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
29                  0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25};
30  const BYTE  dataOut_AES128_CTR [] = {
31                  0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
32                  0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
33                  0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
34                  0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff};
35  #endif
36  #if AES_192
37  const BYTE  key_AES192 [] = {
38                  0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
39                  0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
40                  0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b};
41  const BYTE  dataIn_AES192 [] = {
42                  0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
43                  0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
44                  0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
45                  0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
46  const BYTE  dataOut_AES192_ECB [] = {
47                  0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f,
48                  0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc,
49                  0x97, 0x41, 0x04, 0x84, 0x6d, 0x0a, 0xd3, 0xad,
50                  0x77, 0x34, 0xec, 0xb3, 0xec, 0xee, 0x4e, 0xef};
51  const BYTE  dataOut_AES192_CBC [] = {
52                  0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d,
53                  0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
54                  0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4,
55                  0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a};
56  const BYTE  dataOut_AES192_CFB [] = {
57                  0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
```

```
58                  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

10.1.11.1 Introduction

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.

```
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
6   #include    "SymmetricTestData.h"
```

10.1.11.2 Symmetric Test Structures

```
7   const SYMMETRIC_TEST_VECTOR   c_symTestValues[NUM_SYMS] = {
8   #undef  COMMA
9   #if AES_128
10      {ALG_AES_VALUE, 128, key_AES128, 16, sizeof(dataIn_AES128), dataIn_AES128,
11      {dataOut_AES128_CTR, dataOut_AES128_OFB, dataOut_AES128_CBC,
12       dataOut_AES128_CFB, dataOut_AES128_ECB}}
13  #   define COMMA ,
14  #endif
15  #if AES_192
16      COMMA
17      {ALG_AES_VALUE, 192, key_AES192, 16, sizeof(dataIn_AES192), dataIn_AES192,
18      {dataOut_AES192_CTR, dataOut_AES192_OFB, dataOut_AES192_CBC,
19       dataOut_AES192_CFB, dataOut_AES192_ECB}}
20  #   undef   COMMA
21  #   define COMMA ,
22  #endif
23  #if AES_256
24      COMMA
25      {ALG_AES_VALUE, 256, key_AES256, 16, sizeof(dataIn_AES256), dataIn_AES256,
26      {dataOut_AES256_CTR, dataOut_AES256_OFB, dataOut_AES256_CBC,
27      dataOut_AES256_CFB, dataOut_AES256_ECB}}
28  #   undef  COMMA
29  #   define COMMA ,
30  #endif
31  #if SM4_128
32      COMMA
33      {ALG_SM4_VALUE, 128, key_SM4128, 16, sizeof(dataIn_SM4128), dataIn_SM4128,
34      {dataOut_SM4128_CTR, dataOut_SM4128_OFB, dataOut_SM4128_CBC,
35       dataOut_SM4128_CFB, dataOut_AES128_ECB}}
36  #endif
37  };
38  #undef COMMA
39  #endif  // _SYMMETRIC_TEST_H
```

10.1.12 EccTestData.h

This file contains the parameter data for ECC testing.

```
1   #ifdef  SELF_TEST_DATA
2   TPM2B_TYPE(EC_TEST, 32);
3   const  TPM_ECC_CURVE        c_testCurve = 00003;
```

The **static** key

```
4   const  TPM2B_EC_TEST    c_ecTestKey_ds = {{32, {
5    0xdf,0x8d,0xa4,0xa3,0x88,0xf6,0x76,0x96,0x89,0xfc,0x2f,0x2d,0xa1,0xb4,0x39,0x7a,
6    0x78,0xc4,0x7f,0x71,0x8c,0xa6,0x91,0x85,0xc0,0xbf,0xf3,0x54,0x20,0x91,0x2f,0x73}}};
7   const  TPM2B_EC_TEST    c_ecTestKey_QsX = {{32, {
8    0x17,0xad,0x2f,0xcb,0x18,0xd4,0xdb,0x3f,0x2c,0x53,0x13,0x82,0x42,0x97,0xff,0x8d,
9    0x99,0x50,0x16,0x02,0x35,0xa7,0x06,0xae,0x1f,0xda,0xe2,0x9c,0x12,0x77,0xc0,0xf9}}};
10  const  TPM2B_EC_TEST    c_ecTestKey_QsY = {{32, {
11   0xa6,0xca,0xf2,0x18,0x45,0x96,0x6e,0x58,0xe6,0x72,0x34,0x12,0x89,0xcd,0xaa,0xad,
12   0xcb,0x68,0xb2,0x51,0xdc,0x5e,0xd1,0x6d,0x38,0x20,0x35,0x57,0xb2,0xfd,0xc7,0x52}}};
```

The **ephemeral** key

```
13  const  TPM2B_EC_TEST    c_ecTestKey_de = {{32, {
14   0xb6,0xb5,0x33,0x5c,0xd1,0xee,0x52,0x07,0x99,0xea,0x2e,0x8f,0x8b,0x19,0x18,0x07,
15   0xc1,0xf8,0xdf,0xdd,0xb8,0x77,0x00,0xc7,0xd6,0x53,0x21,0xed,0x02,0x53,0xee,0xac}}};
16  const  TPM2B_EC_TEST    c_ecTestKey_QeX = {{32, {
17   0xa5,0x1e,0x80,0xd1,0x76,0x3e,0x8b,0x96,0xce,0xcc,0x21,0x82,0xc9,0xa2,0xa2,0xed,
18   0x47,0x21,0x89,0x53,0x44,0xe9,0xc7,0x92,0xe7,0x31,0x48,0x38,0xe6,0xea,0x93,0x47}}};
19  const  TPM2B_EC_TEST    c_ecTestKey_QeY = {{32, {
20   0x30,0xe6,0x4f,0x97,0x03,0xa1,0xcb,0x3b,0x32,0x2a,0x70,0x39,0x94,0xeb,0x4e,0xea,
21   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  TPM2B_TYPE(TEST_VALUE, 64);
29  const TPM2B_TEST_VALUE        c_ecTestValue = {{64, {
30   0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
31   0xf4,0x45,0x7f,0x13,0xd8,0x13,0x81,0x0b,0xe5,0x76,0xbe,0xaa,0xb6,0x3f,0x8d,0x4d,
32   0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xcb,0x0c,0xc7,0x11,0x8d,0xc3,0xff,
33   0x62,0x69,0xa2,0xbe,0x46,0x90,0xe7,0x7d,0x81,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}}};
34  #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
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   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1b,0x08,0x9f,0xde,
43   0xef,0x62,0xe3,0xf1,0x14,0xcb,0x54,0x28,0x13,0x76,0xfc,0x6d,0x69,0x22,0xb5,0x3e}}};
44  const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
45   0xd9,0xd3,0x20,0xfb,0x4d,0x16,0xf2,0xe6,0xe2,0x45,0x07,0x45,0x1c,0x92,0x92,0x92,
46   0xa9,0x6b,0x48,0xf8,0xd1,0x98,0x29,0x4d,0xd3,0x8f,0x56,0xf2,0xbb,0x2e,0x22,0x3b}}};
47  #endif // SHA1
48  #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
49  const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
50   0x04,0x7d,0x54,0xeb,0x04,0x6f,0x56,0xec,0xa2,0x6c,0x38,0x8c,0xeb,0x43,0x0b,0x71,
```



```
51    0xf8,0xf2,0xf4,0xa5,0xe0,0x1d,0x3c,0xa2,0x39,0x31,0xe4,0xe7,0x36,0x3b,0xb5,0x5f}}};
52   const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
53    0x8f,0xd0,0x12,0xd9,0x24,0x75,0xf6,0xc4,0x3b,0xb5,0x46,0x75,0x3a,0x41,0x8d,0x80,
54    0x23,0x99,0x38,0xd7,0xe2,0x40,0xca,0x9a,0x19,0x2a,0xfc,0x54,0x75,0xd3,0x4a,0x6e}}};
55   const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32, {
56    0xf7,0xb9,0x15,0x4c,0x34,0xf6,0x41,0x19,0xa3,0xd2,0xf1,0xbd,0xf4,0x13,0x6a,0x4f,
57    0x63,0xb8,0x4d,0xb5,0xc8,0xcd,0xde,0x85,0x95,0xa5,0x39,0x0a,0x14,0x49,0x3d,0x2f}}};
58   const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
59    0xfe,0xbe,0x17,0xaa,0x31,0x22,0x9f,0xd0,0xd2,0xf5,0x25,0x04,0x92,0xb0,0xaa,0x4e,
60    0xcc,0x1c,0xb6,0x79,0xd6,0x42,0xb3,0x4e,0x3f,0xbb,0xfe,0x5f,0xd0,0xd0,0x8b,0xc3}}};
61   #endif // SHA256
62   #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
63   const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
64    0xf5,0x74,0x6d,0xd6,0xc6,0x56,0x86,0xbb,0xba,0x1c,0xba,0x75,0x65,0xee,0x64,0x31,
65    0xce,0x04,0xe3,0x9f,0x24,0x3f,0xbd,0xfe,0x04,0xcd,0xab,0x7e,0xfe,0xad,0xcb,0x82}}};
66   const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
67    0xc2,0x4f,0x32,0xa1,0x06,0xc0,0x85,0x4f,0xc6,0xd8,0x31,0x66,0x91,0x9f,0x79,0xcd,
68    0x5b,0xe5,0x7b,0x94,0xa1,0x91,0x38,0xac,0xd4,0x20,0xa2,0x10,0xf0,0xd5,0x9d,0xbf}}};
69   const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32, {
70    0x1e,0xb8,0xe1,0xbf,0xa1,0x9e,0x39,0x1e,0x58,0xa2,0xe6,0x59,0xd0,0x1a,0x6a,0x03,
71    0x6a,0x1f,0x1c,0x4f,0x36,0x19,0xc1,0xec,0x30,0xa4,0x85,0x1b,0xe9,0x74,0x35,0x66}}};
72   const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
73    0xb9,0xe6,0xe3,0x7e,0xcb,0xb9,0xea,0xf1,0xcc,0xf4,0x48,0x44,0x4a,0xda,0xc8,0xd7,
74    0x87,0xb4,0xba,0x40,0xfe,0x5b,0x68,0x11,0x14,0xcf,0xa0,0x0e,0x85,0x46,0x99,0x01}}};
75   #endif // SHA384
76   #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
77   const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
78    0xc9,0x71,0xa6,0xb4,0xaf,0x46,0x26,0x8c,0x27,0x00,0x06,0x3b,0x00,0x0f,0xa3,0x17,
79    0x72,0x48,0x40,0x49,0x4d,0x51,0x4f,0xa4,0xcb,0x7e,0x86,0xe9,0xe7,0xb4,0x79,0xb2}}};
80   const TPM2B_EC_TEST    c_TestEcDsa_s = {{32,{
81    0x87,0xbc,0xc0,0xed,0x74,0x60,0x9e,0xfa,0x4e,0xe8,0x16,0xf3,0xf9,0x6b,0x26,0x07,
82    0x3c,0x74,0x31,0x7e,0xf0,0x62,0x46,0xdc,0xd6,0x45,0x22,0x47,0x3e,0x0c,0xa0,0x02}}};
83   const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32,{
84    0xcc,0x07,0xad,0x65,0x91,0xdd,0xa0,0x10,0x23,0xae,0x53,0xec,0xdf,0xf1,0x50,0x90,
85    0x16,0x96,0xf4,0x45,0x09,0x73,0x9c,0x84,0xb5,0x5c,0x5f,0x08,0x51,0xcb,0x60,0x01}}};
86   const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
87    0x55,0x20,0x21,0x54,0xe2,0x49,0x07,0x47,0x71,0xf4,0x99,0x15,0x54,0xf3,0xab,0x14,
88    0xdb,0x8e,0xda,0x79,0xb6,0x02,0x0e,0xe3,0x5e,0x6f,0x2c,0xb6,0x05,0xbd,0x14,0x10}}};
89   #endif // SHA512
90   #endif // SELF_TEST_DATA
```

10.1.13 CryptSym.h

10.1.13.1 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 cryptographic library.

10.1.13.2 Includes, Defines, and Typedefs

```
#ifndef CRYPT_SYM_H
#define CRYPT_SYM_H
typedef union tpmCryptKeySchedule_t {
#if     ALG_AES
    tpmKeyScheduleAES           AES;
#endif
#if     ALG_SM4
    tpmKeyScheduleSM4           SM4;
#endif
#if     ALG_CAMELLIA
    tpmKeyScheduleCAMELLIA      CAMELLIA;
#endif

#if     ALG_TDES
    tpmKeyScheduleTDES          TDES[3];
#endif
#if SYMMETRIC_ALIGNMENT == 8
    uint64_t            alignment;
#else
    uint32_t            alignment;
#endif
} tpmCryptKeySchedule_t;
```

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.

```
#   define ENCRYPT(keySchedule, in, out)                \
      encrypt(SWIZZLE(keySchedule, in, out))
#   define DECRYPT(keySchedule, in, out)                \
      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.

```
#define ENCRYPT_CASE(ALG)                                                   \
    case TPM_ALG_##ALG:                                                     \
        TpmCryptSetEncryptKey##ALG(key, keySizeInBits, &keySchedule.ALG);   \
        encrypt = (TpmCryptSetSymKeyCall_t)TpmCryptEncrypt##ALG;            \
        break;
#define DECRYPT_CASE(ALG)                                                   \
    case TPM_ALG_##ALG:                                                     \
        TpmCryptSetDecryptKey##ALG(key, keySizeInBits, &keySchedule.ALG);   \
        decrypt = (TpmCryptSetSymKeyCall_t)TpmCryptDecrypt##ALG;            \
        break;
#if     ALG_AES
#define ENCRYPT_CASE_AES    ENCRYPT_CASE(AES)
```



```
39  #define DECRYPT_CASE_AES    DECRYPT_CASE(AES)
40  #else
41  #define ENCRYPT_CASE_AES
42  #define DECRYPT_CASE_AES
43  #endif
44  #if     ALG_SM4
45  #define ENCRYPT_CASE_SM4    ENCRYPT_CASE(SM4)
46  #define DECRYPT_CASE_SM4    DECRYPT_CASE(SM4)
47  #else
48  #define ENCRYPT_CASE_SM4
49  #define DECRYPT_CASE_SM4
50  #endif
51  #if     ALG_CAMELLIA
52  #define ENCRYPT_CASE_CAMELLIA    ENCRYPT_CASE(CAMELLIA)
53  #define DECRYPT_CASE_CAMELLIA    DECRYPT_CASE(CAMELLIA)
54  #else
55  #define ENCRYPT_CASE_CAMELLIA
56  #define DECRYPT_CASE_CAMELLIA
57  #endif
58  #if     ALG_TDES
59  #define ENCRYPT_CASE_TDES    ENCRYPT_CASE(TDES)
60  #define DECRYPT_CASE_TDES    DECRYPT_CASE(TDES)
61  #else
62  #define ENCRYPT_CASE_TDES
63  #define DECRYPT_CASE_TDES
64  #endif
```

For each algorithm the case will either be defined or null.

```
65  #define     SELECT(direction)                          \
66      switch(algorithm)                                  \
67      {                                                  \
68          direction##_CASE_AES                           \
69          direction##_CASE_SM4                           \
70          direction##_CASE_CAMELLIA                      \
71          direction##_CASE_TDES                          \
72          default:                                       \
73              FAIL(FATAL_ERROR_INTERNAL);                \
74      }
75  #endif // CRYPT_SYM_H
```


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.

NOTE: In this implementation, large local variables are made static to minimize stack usage, which is critical for stack-constrained platforms.

10.2.1.2 Includes and Defines

```
1   #include    "Tpm.h"
2   #define     SELF_TEST_DATA
3   #if SELF_TEST
```

These includes pull in the data structures. They contain data definitions for the various tests.

```
4   #include    "SelfTest.h"
5   #include    "SymmetricTest.h"
6   #include    "RsaTestData.h"
7   #include    "EccTestData.h"
8   #include    "HashTestData.h"
9   #include    "KdfTestData.h"
10  #define TEST_DEFAULT_TEST_HASH(vector)                                  \
11              if(TEST_BIT(DEFAULT_TEST_HASH, g_toTest))                   \
12                  TestHash(DEFAULT_TEST_HASH, vector);
```

Make sure that the algorithm has been tested

```
13  #define CLEAR_BOTH(alg)     {   CLEAR_BIT(alg, *toTest);                \
14                                  if(toTest != &g_toTest)                 \
15                                      CLEAR_BIT(alg, g_toTest); }
16  #define SET_BOTH(alg)     {   SET_BIT(alg, *toTest);                    \
17                                  if(toTest != &g_toTest)                 \
18                                      SET_BIT(alg, g_toTest); }
19  #define TEST_BOTH(alg)       ((toTest != &g_toTest)                      \
20                              ? TEST_BIT(alg, *toTest) || TEST_BIT(alg, g_toTest)  \
21                              : TEST_BIT(alg, *toTest))
```

Can only cancel if doing a list.

```
22  #define CHECK_CANCELED                                          \
23      if(_plat__IsCanceled() && toTest != &g_toTest)              \
24          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.


```
25  static TPM_RC
26  TestHash(
27      TPM_ALG_ID          hashAlg,
28      ALGORITHM_VECTOR    *toTest
29      )
30  {
31      static TPM2B_DIGEST      computed;  // value computed
32      static HMAC_STATE        state;
33      UINT16                   digestSize;
34      const TPM2B             *testDigest = NULL;
35  //    TPM2B_TYPE(HMAC_BLOCK, DEFAULT_TEST_HASH_BLOCK_SIZE);
36
37      pAssert(hashAlg != ALG_NULL_VALUE);
38      switch(hashAlg)
39      {
40  #if ALG_SHA1
41          case ALG_SHA1_VALUE:
42              testDigest = &c_SHA1_digest.b;
43              break;
44  #endif
45  #if ALG_SHA256
46          case ALG_SHA256_VALUE:
47              testDigest = &c_SHA256_digest.b;
48              break;
49  #endif
50  #if ALG_SHA384
51          case ALG_SHA384_VALUE:
52              testDigest = &c_SHA384_digest.b;
53              break;
54  #endif
55  #if ALG_SHA512
56          case ALG_SHA512_VALUE:
57              testDigest = &c_SHA512_digest.b;
58              break;
59  #endif
60  #if ALG_SM3_256
61          case ALG_SM3_256_VALUE:
62              testDigest = &c_SM3_256_digest.b;
63              break;
64  #endif
65          default:
66              FAIL(FATAL_ERROR_INTERNAL);
67      }
68      // Clear the to-test bits
69      CLEAR_BOTH(hashAlg);
70
71      // Set the HMAC key to twice the digest size
72      digestSize = CryptHashGetDigestSize(hashAlg);
73      CryptHmacStart(&state, hashAlg, digestSize * 2,
74                     (BYTE *)c_hashTestKey.t.buffer);
75      CryptDigestUpdate(&state.hashState, 2 * CryptHashGetBlockSize(hashAlg),
76                        (BYTE *)c_hashTestData.t.buffer);
77      computed.t.size = digestSize;
78      CryptHmacEnd(&state, digestSize, computed.t.buffer);
79      if((testDigest->size != computed.t.size)
80         || (memcmp(testDigest->buffer, computed.t.buffer, computed.b.size) != 0))
81          SELF_TEST_FAILURE;
82      return TPM_RC_SUCCESS;
83  }
```


10.2.1.4 Symmetric Test Functions

10.2.1.4.1 MakeIv()

Internal function to make the appropriate IV depending on the mode.

```
84  static UINT32
85  MakeIv(
86      TPM_ALG_ID    mode,     // IN: symmetric mode
87      UINT32        size,     // IN: block size of the algorithm
88      BYTE         *iv        // OUT: IV to fill in
89      )
90  {
91      BYTE          i;
92
93      if(mode == ALG_ECB_VALUE)
94          return 0;
95      if(mode == ALG_CTR_VALUE)
96      {
97          // The test uses an IV that has 0xff in the last byte
98          for(i = 1; i <= size; i++)
99              *iv++ = 0xff - (BYTE)(size - i);
100     }
101     else
102     {
103         for(i = 0; i < size; i++)
104             *iv++ = i;
105     }
106     return size;
107 }
```

10.2.1.4.2 TestSymmetricAlgorithm()

Function to test a specific algorithm, key size, and mode.

```
108 static void
109 TestSymmetricAlgorithm(
110     const SYMMETRIC_TEST_VECTOR     *test,          //
111     TPM_ALG_ID                       mode           //
112     )
113 {
114     static BYTE                 encrypted[MAX_SYM_BLOCK_SIZE * 2];
115     static BYTE                 decrypted[MAX_SYM_BLOCK_SIZE * 2];
116     static TPM2B_IV             iv;
117 //
118     // Get the appropriate IV
119     iv.t.size = (UINT16)MakeIv(mode, test->ivSize, iv.t.buffer);
120
121     // Encrypt known data
122     CryptSymmetricEncrypt(encrypted, test->alg, test->keyBits, test->key, &iv,
123                           mode, test->dataInOutSize, test->dataIn);
124     // Check that it matches the expected value
125     if(!MemoryEqual(encrypted, test->dataOut[mode - ALG_CTR_VALUE],
126                     test->dataInOutSize))
127         SELF_TEST_FAILURE;
128     // Reinitialize the iv for decryption
129     MakeIv(mode, test->ivSize, iv.t.buffer);
130     CryptSymmetricDecrypt(decrypted, test->alg, test->keyBits, test->key, &iv,
131                           mode, test->dataInOutSize,
132                           test->dataOut[mode - ALG_CTR_VALUE]);
133     // Make sure that it matches what we started with
134     if(!MemoryEqual(decrypted, test->dataIn, test->dataInOutSize))
135         SELF_TEST_FAILURE;
```



```
}
```

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

Return Value	Meaning
TRUE(1)	all symmetric algorithms tested
FALSE(0)	not all symmetric algorithms tested

```
static BOOL
AllSymsAreDone(
    ALGORITHM_VECTOR        *toTest
    )
{
    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

Return Value	Meaning
TRUE(1)	all modes tested
FALSE(0)	all modes not 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 || alg == ALG_CAMELLIA_VALUE)
    {
```

```
168         // Will test the algorithm for all modes and key sizes
169         CLEAR_BOTH(alg);
170
171         // A test this algorithm for all modes
172         for(index = 0; index < NUM_SYMS; index++)
173         {
174             if(c_symTestValues[index].alg == alg)
175             {
176                 for(mode = TPM_SYM_MODE_FIRST;
177                 mode <= TPM_SYM_MODE_LAST;
178                     mode++)
179                 {
180                     if(TEST_BIT(mode, *toTest))
181                         TestSymmetricAlgorithm(&c_symTestValues[index], mode);
182                 }
183             }
184         }
185         // if all the symmetric tests are done
186         if(AllSymsAreDone(toTest))
187         {
188             // all symmetric algorithms tested so no modes should be set
189             for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
190                 CLEAR_BOTH(alg);
191         }
192     }
193     else if(TPM_SYM_MODE_FIRST <= alg && alg <= TPM_SYM_MODE_LAST)
194     {
195         // Test this mode for all key sizes and algorithms
196         for(index = 0; index < NUM_SYMS; index++)
197         {
198             // The mode testing only comes into play when doing self tests
199             // by command. When doing self tests by command, the block ciphers are
200             // tested first. That means that all of their modes would have been
201             // tested for all key sizes. If there is no block cipher left to
202             // test, then clear this mode bit.
203             if(!TEST_BIT(ALG_AES_VALUE, *toTest)
204                && !TEST_BIT(ALG_SM4_VALUE, *toTest))
205             {
206                 CLEAR_BOTH(alg);
207             }
208             else
209             {
210                 for(index = 0; index < NUM_SYMS; index++)
211                 {
212                     if(TEST_BIT(c_symTestValues[index].alg, *toTest))
213                         TestSymmetricAlgorithm(&c_symTestValues[index], alg);
214                 }
215                 // have tested this mode for all algorithms
216                 CLEAR_BOTH(alg);
217             }
218         }
219         if(AllModesAreDone(toTest))
220         {
221             CLEAR_BOTH(ALG_AES_VALUE);
222             CLEAR_BOTH(ALG_SM4_VALUE);
223         }
224     }
225     else
226         pAssert(alg == 0 && alg != 0);
227     return TPM_RC_SUCCESS;
228 }
```

10.2.1.5 RSA Tests

```
229 #if     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 CryptRsaValidateSignature(). 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.

```
230 static void
231 RsaKeyInitialize(
232     OBJECT          *testObject
233     )
234 {
235     MemoryCopy2B(&testObject->publicArea.unique.rsa.b, (P2B)&c_rsaPublicModulus,
236                  sizeof(c_rsaPublicModulus));
237     MemoryCopy2B(&testObject->sensitive.sensitive.rsa.b, (P2B)&c_rsaPrivatePrime,
238                  sizeof(testObject->sensitive.sensitive.rsa.t.buffer));
239     testObject->publicArea.parameters.rsaDetail.keyBits = RSA_TEST_KEY_SIZE * 8;
240     // Use the default exponent
241     testObject->publicArea.parameters.rsaDetail.exponent = 0;
242     testObject->attributes.privateExp = 0;
243 }
```

10.2.1.5.3 TestRsaEncryptDecrypt()

These test are for an public key encryption that uses a random value

```
244 static TPM_RC
245 TestRsaEncryptDecrypt(
246     TPM_ALG_ID           scheme,            // IN: the scheme
247     ALGORITHM_VECTOR    *toTest             //
248     )
249 {
250     static TPM2B_PUBLIC_KEY_RSA      testInput;
251     static TPM2B_PUBLIC_KEY_RSA      testOutput;
252     static OBJECT                    testObject;
253     const TPM2B_RSA_TEST_KEY        *kvtValue = NULL;
254     TPM_RC                           result = TPM_RC_SUCCESS;
255     const TPM2B                     *testLabel = NULL;
256     TPMT_RSA_DECRYPT                 rsaScheme;
257 //
258     // Don't need to initialize much of the test object but do need to initialize
259     // the flag indicating that the private exponent has been computed.
260     testObject.attributes.privateExp = CLEAR;
261     RsaKeyInitialize(&testObject);
262     rsaScheme.scheme = scheme;
263     rsaScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
264     CLEAR_BOTH(scheme);
```

```
265     CLEAR_BOTH(ALG_NULL_VALUE);
266     if(scheme == ALG_NULL_VALUE)
267     {
268         // This is an encryption scheme using the private key without any encoding.
269         memcpy(testInput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue));
270         testInput.t.size = sizeof(c_RsaTestValue);
271         if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
272                                              &testObject, &rsaScheme, NULL, NULL))
273             SELF_TEST_FAILURE;
274         if(!MemoryEqual(testOutput.t.buffer, c_RsaepKvt.buffer, c_RsaepKvt.size))
275             SELF_TEST_FAILURE;
276         MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
277         if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
278                                              &testObject, &rsaScheme, NULL))
279             SELF_TEST_FAILURE;
280         if(!MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
281                         sizeof(c_RsaTestValue)))
282             SELF_TEST_FAILURE;
283     }
284     else
285     {
286         // ALG_RSAES_VALUE:
287         // This is an decryption scheme using padding according to
288         // PKCS#1v2.1, 7.2. This padding uses random bits. To test a public
289         // key encryption that uses random data, encrypt a value and then
290         // decrypt the value and see that we get the encrypted data back.
291         // The hash is not used by this encryption so it can be TMP_ALG_NULL
292
293         // ALG_OAEP_VALUE:
294         // This is also an decryption scheme and it also uses a
295         // pseudo-random
296         // value. However, this also uses a hash algorithm. So, we may need
297         // to test that algorithm before use.
298         if(scheme == ALG_OAEP_VALUE)
299         {
300             TEST_DEFAULT_TEST_HASH(toTest);
301             kvtValue = &c_OaepKvt;
302             testLabel = OAEP_TEST_STRING;
303         }
304         else if(scheme == ALG_RSAES_VALUE)
305         {
306             kvtValue = &c_RsaesKvt;
307             testLabel = NULL;
308         }
309         else
310             SELF_TEST_FAILURE;
311         // Only use a digest-size portion of the test value
312         memcpy(testInput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
313         testInput.t.size = DEFAULT_TEST_DIGEST_SIZE;
314
315         // See if the encryption works
316         if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
317                                              &testObject, &rsaScheme, testLabel,
318                                              NULL))
319             SELF_TEST_FAILURE;
320         MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
321         // see if we can decrypt this value and get the original data back
322         if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
323                                              &testObject, &rsaScheme, testLabel))
324             SELF_TEST_FAILURE;
325         // See if the results compare
326         if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
327            || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
328                            DEFAULT_TEST_DIGEST_SIZE))
329             SELF_TEST_FAILURE;
330         // Now check that the decryption works on a known value
```



```
331         MemoryCopy2B(&testInput.b, (P2B)kvtValue,
332                      sizeof(testInput.t.buffer));
333         if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
334                                              &testObject, &rsaScheme, testLabel))
335             SELF_TEST_FAILURE;
336         if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
337            || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
338                            DEFAULT_TEST_DIGEST_SIZE))
339             SELF_TEST_FAILURE;
340     }
341     return result;
342 }
```

10.2.1.5.4 TestRsaSignAndVerify()

This function does the testing of the RSA sign and verification functions. This test does a KVT.

```
343 static TPM_RC
344 TestRsaSignAndVerify(
345     TPM_ALG_ID                scheme,
346     ALGORITHM_VECTOR         *toTest
347     )
348 {
349     TPM_RC                      result = TPM_RC_SUCCESS;
350     static OBJECT               testObject;
351     static TPM2B_DIGEST         testDigest;
352     static TPMT_SIGNATURE       testSig;
353
354     // Do a sign and signature verification.
355     // RSASSA:
356     // This is a signing scheme according to PKCS#1-v2.1 8.2. It does not
357     // use random data so there is a KVT for the signing operation. On
358     // first use of the scheme for signing, use the TPM's RSA key to
359     // sign a portion of c_RsaTestData and compare the results to c_RsassaKvt. Then
360     // decrypt the data to see that it matches the starting value. This verifies
361     // the signature with a KVT
362
363     // Clear the bits indicating that the function has not been checked. This is to
364     // prevent looping
365     CLEAR_BOTH(scheme);
366     CLEAR_BOTH(ALG_NULL_VALUE);
367     CLEAR_BOTH(ALG_RSA_VALUE);
368
369     RsaKeyInitialize(&testObject);
370     memcpy(testDigest.t.buffer, (BYTE *)c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
371     testDigest.t.size = DEFAULT_TEST_DIGEST_SIZE;
372     testSig.sigAlg = scheme;
373     testSig.signature.rsapss.hash = DEFAULT_TEST_HASH;
374
375     // RSAPSS:
376     // This is a signing scheme a according to PKCS#1-v2.2 8.1 it uses
377     // random data in the signature so there is no KVT for the signing
378     // operation. To test signing, the TPM will use the TPM's RSA key
379     // to sign a portion of c_RsaTestValue and then it will verify the
380     // signature. For verification, c_RsapssKvt is verified before the
381     // user signature blob is verified. The worst case for testing of this
382     // algorithm is two private and one public key operation.
383
384     // The process is to sign known data. If RSASSA is being done, verify that the
385     // signature matches the precomputed value. For both, use the signed value and
386     // see that the verification says that it is a good signature. Then
387     // if testing RSAPSS, do a verify of a known good signature. This ensures that
388     // the validation function works.
389
```

```
390     if(TPM_RC_SUCCESS != CryptRsaSign(&testSig, &testObject, &testDigest, NULL))
391         SELF_TEST_FAILURE;
392     // For RSASSA, make sure the results is what we are looking for
393     if(testSig.sigAlg == ALG_RSASSA_VALUE)
394     {
395         if(testSig.signature.rsassa.sig.t.size != RSA_TEST_KEY_SIZE
396            || !MemoryEqual(c_RsassaKvt.buffer,
397                            testSig.signature.rsassa.sig.t.buffer,
398                            RSA_TEST_KEY_SIZE))
399             SELF_TEST_FAILURE;
400     }
401     // See if the TPM will validate its own signatures
402     if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
403                                                    &testDigest))
404         SELF_TEST_FAILURE;
405     // If this is RSAPSS, check the verification with known signature
406     // Have to copy because  CrytpRsaValidateSignature() eats the signature
407     if(ALG_RSAPSS_VALUE == scheme)
408     {
409         MemoryCopy2B(&testSig.signature.rsapss.sig.b, (P2B)&c_RsapssKvt,
410                      sizeof(testSig.signature.rsapss.sig.t.buffer));
411         if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
412                                                        &testDigest))
413             SELF_TEST_FAILURE;
414     }
415     return result;
416 }
```

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*

```
417 static TPM_RC
418 TestRsa(
419     TPM_ALG_ID              alg,
420     ALGORITHM_VECTOR        *toTest
421     )
422 {
423     TPM_RC                  result = TPM_RC_SUCCESS;
424 //
425     switch(alg)
426     {
427         case ALG_NULL_VALUE:
428         // This is the RSAEP/RSADP function. If we are processing a list, don't
429         // need to test these now because any other test will validate
430         // RSAEP/RSADP. Can tell this is list of test by checking to see if
431         // 'toTest' is pointing at g_toTest. If so, this is an isolated test
432         // an need to go ahead and do the test;
433             if((toTest == &g_toTest)
434                || (!TEST_BIT(ALG_RSASSA_VALUE, *toTest)
435                    && !TEST_BIT(ALG_RSAES_VALUE, *toTest)
436                    && !TEST_BIT(ALG_RSAPSS_VALUE, *toTest)
437                    && !TEST_BIT(ALG_OAEP_VALUE, *toTest)))
438                // Not running a list of tests or no other tests on the list
439                // so run the test now
440                 result = TestRsaEncryptDecrypt(alg, toTest);
441             // if not running the test now, leave the bit on, just in case things
442             // get interrupted
443             break;
444         case ALG_OAEP_VALUE:
445         case ALG_RSAES_VALUE:
446             result = TestRsaEncryptDecrypt(alg, toTest);
447             break;
```



```
448         case ALG_RSAPSS_VALUE:
449         case ALG_RSASSA_VALUE:
450             result = TestRsaSignAndVerify(alg, toTest);
451             break;
452         default:
453             SELF_TEST_FAILURE;
454     }
455     return result;
456 }
457 #endif // ALG_RSA
```

10.2.1.6 ECC Tests

```
458 #if     ALG_ECC
```

10.2.1.6.1 LoadEccParameter()

This function is mostly for readability and type checking

```
459 static void
460 LoadEccParameter(
461     TPM2B_ECC_PARAMETER          *to,        // target
462     const TPM2B_EC_TEST          *from       // source
463     )
464 {
465     MemoryCopy2B(&to->b, &from->b, sizeof(to->t.buffer));
466 }
```

10.2.1.6.2 LoadEccPoint()

```
467 static void
468 LoadEccPoint(
469     TPMS_ECC_POINT               *point,     // target
470     const TPM2B_EC_TEST          *x,         // source
471     const TPM2B_EC_TEST          *y
472     )
473 {
474     MemoryCopy2B(&point->x.b, (TPM2B *)x, sizeof(point->x.t.buffer));
475     MemoryCopy2B(&point->y.b, (TPM2B *)y, sizeof(point->y.t.buffer));
476 }
```

10.2.1.6.3 TestECDH()

This test does a KVT on a point multiply.

```
477 static TPM_RC
478 TestECDH(
479     TPM_ALG_ID          scheme,         // IN: for consistency
480     ALGORITHM_VECTOR    *toTest         // IN/OUT: modified after test is run
481     )
482 {
483     static TPMS_ECC_POINT       Z;
484     static TPMS_ECC_POINT       Qe;
485     static TPM2B_ECC_PARAMETER  ds;
486     TPM_RC                      result = TPM_RC_SUCCESS;
487 //
488     NOT_REFERENCED(scheme);
489     CLEAR_BOTH(ALG_ECDH_VALUE);
490     LoadEccParameter(&ds, &c_ecTestKey_ds);
491     LoadEccPoint(&Qe, &c_ecTestKey_QeX, &c_ecTestKey_QeY);
492     if(TPM_RC_SUCCESS != CryptEccPointMultiply(&Z, c_testCurve, &Qe, &ds,
```

```
493                                                 NULL, NULL))
494         SELF_TEST_FAILURE;
495     if(!MemoryEqual2B(&c_ecTestEcdh_X.b, &Z.x.b)
496        || !MemoryEqual2B(&c_ecTestEcdh_Y.b, &Z.y.b))
497         SELF_TEST_FAILURE;
498     return result;
499 }
```

10.2.1.6.4 TestEccSignAndVerify()

```
500 static TPM_RC
501 TestEccSignAndVerify(
502     TPM_ALG_ID                  scheme,
503     ALGORITHM_VECTOR           *toTest
504     )
505 {
506     static OBJECT               testObject;
507     static TPMT_SIGNATURE       testSig;
508     static TPMT_ECC_SCHEME      eccScheme;
509
510     testSig.sigAlg = scheme;
511     testSig.signature.ecdsa.hash = DEFAULT_TEST_HASH;
512
513     eccScheme.scheme = scheme;
514     eccScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
515
516     CLEAR_BOTH(scheme);
517     CLEAR_BOTH(ALG_ECDH_VALUE);
518
519     // ECC signature verification testing uses a KVT.
520     switch(scheme)
521     {
522         case ALG_ECDSA_VALUE:
523             LoadEccParameter(&testSig.signature.ecdsa.signatureR, &c_TestEcDsa_r);
524             LoadEccParameter(&testSig.signature.ecdsa.signatureS, &c_TestEcDsa_s);
525             break;
526         case ALG_ECSCHNORR_VALUE:
527             LoadEccParameter(&testSig.signature.ecschnorr.signatureR,
528                              &c_TestEcSchnorr_r);
529             LoadEccParameter(&testSig.signature.ecschnorr.signatureS,
530                              &c_TestEcSchnorr_s);
531             break;
532         case ALG_SM2_VALUE:
533             // don't have a test for SM2
534             return TPM_RC_SUCCESS;
535         default:
536             SELF_TEST_FAILURE;
537             break;
538     }
539     TEST_DEFAULT_TEST_HASH(toTest);
540
541     // Have to copy the key. This is because the size used in the test vectors
542     // is the size of the ECC parameter for the test key while the size of a point
543     // is TPM dependent
544     MemoryCopy2B(&testObject.sensitive.sensitive.ecc.b, &c_ecTestKey_ds.b,
545                  sizeof(testObject.sensitive.sensitive.ecc.t.buffer));
546     LoadEccPoint(&testObject.publicArea.unique.ecc, &c_ecTestKey_QsX,
547                  &c_ecTestKey_QsY);
548     testObject.publicArea.parameters.eccDetail.curveID = c_testCurve;
549
550     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
551                                                    (TPM2B_DIGEST *)&c_ecTestValue.b))
552     {
553         SELF_TEST_FAILURE;
554     }
```



```
555         CHECK_CANCELED;
556
557         // Now sign and verify some data
558         if(TPM_RC_SUCCESS != CryptEccSign(&testSig, &testObject,
559                                           (TPM2B_DIGEST *)&c_ecTestValue,
560                                           &eccScheme, NULL))
561             SELF_TEST_FAILURE;
562
563         CHECK_CANCELED;
564
565         if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
566                                                        (TPM2B_DIGEST *)&c_ecTestValue))
567             SELF_TEST_FAILURE;
568
569         CHECK_CANCELED;
570
571         return TPM_RC_SUCCESS;
572     }
```

10.2.1.6.5 TestKDFa()

```
573     static TPM_RC
574     TestKDFa(
575         ALGORITHM_VECTOR        *toTest
576         )
577     {
578         static TPM2B_KDF_TEST_KEY   keyOut;
579         UINT32                      counter = 0;
580     //
581         CLEAR_BOTH(ALG_KDF1_SP800_108_VALUE);
582
583         keyOut.t.size = CryptKDFa(KDF_TEST_ALG, &c_kdfTestKeyIn.b, &c_kdfTestLabel.b,
584                                   &c_kdfTestContextU.b, &c_kdfTestContextV.b,
585                                   TEST_KDF_KEY_SIZE * 8, keyOut.t.buffer,
586                                   &counter, FALSE);
587         if (   keyOut.t.size != TEST_KDF_KEY_SIZE
588             || !MemoryEqual(keyOut.t.buffer, c_kdfTestKeyOut.t.buffer,
589                             TEST_KDF_KEY_SIZE))
590             SELF_TEST_FAILURE;
591
592         return TPM_RC_SUCCESS;
593     }
```

10.2.1.6.6 TestEcc()

```
594     static TPM_RC
595     TestEcc(
596         TPM_ALG_ID              alg,
597         ALGORITHM_VECTOR        *toTest
598         )
599     {
600         TPM_RC                  result = TPM_RC_SUCCESS;
601         NOT_REFERENCED(toTest);
602         switch(alg)
603         {
604             case ALG_ECC_VALUE:
605             case ALG_ECDH_VALUE:
606                 // If this is in a loop then see if another test is going to deal with
607                 // this.
608                 // If toTest is not a self-test list
609                 if((toTest == &g_toTest)
610                     // or this is the only ECC test in the list
611                    || !(TEST_BIT(ALG_ECDSA_VALUE, *toTest)
612                         || TEST_BIT(ALG_ECSCHNORR, *toTest)
```

```
613                     || TEST_BIT(ALG_SM2_VALUE, *toTest)))
614             {
615                 result = TestECDH(alg, toTest);
616             }
617             break;
618         case ALG_ECDSA_VALUE:
619         case ALG_ECSCHNORR_VALUE:
620         case ALG_SM2_VALUE:
621             result = TestEccSignAndVerify(alg, toTest);
622             break;
623         default:
624             SELF_TEST_FAILURE;
625             break;
626     }
627     return result;
628 }
629 #endif // ALG_ECC
```

10.2.1.6.7 TestAlgorithm()

Dispatches to the correct test function for the algorithm or gets a list of testable algorithms.

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 is quietly cleared.

If *alg* is zero (TPM_ALG_ERROR), then the *toTest* vector is cleared of any bits for which there is no 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_CANCELED	test was canceled

```
630 LIB_EXPORT
631 TPM_RC
632 TestAlgorithm(
633     TPM_ALG_ID              alg,
634     ALGORITHM_VECTOR        *toTest
635     )
636 {
637     TPM_ALG_ID              first = (alg == ALG_ERROR_VALUE) ? ALG_FIRST_VALUE : alg;
638     TPM_ALG_ID              last = (alg == ALG_ERROR_VALUE) ? ALG_LAST_VALUE : alg;
639     BOOL                    doTest = (alg != ALG_ERROR_VALUE);
640     TPM_RC                  result = TPM_RC_SUCCESS;
641
642     if(toTest == NULL)
643         toTest = &g_toTest;
644
645     // This is kind of strange. This function will either run a test of the selected
646     // algorithm or just clear a bit if there is no test for the algorithm. So,
647     // either this loop will be executed once for the selected algorithm or once for
648     // each of the possible algorithms. If it is executed more than once ('alg' ==
649     // ALG_ERROR), then no test will be run but bits will be cleared for
650     // unimplemented algorithms. This was done this way so that there is only one
651     // case statement with all of the algorithms. It was easier to have one case
652     // statement than to have multiple ones to manage whenever an algorithm ID is
653     // added.
654     for(alg = first; (alg <= last); alg++)
655     {
656         // if 'alg' was TPM_ALG_ERROR, then we will be cycling through
657         // values, some of which may not be implemented. If the bit in toTest
```



```
658         // happens to be set, then we could either generated an assert, or just
659         // silently CLEAR it. Decided to just clear.
660         if(!TEST_BIT(alg, g_implementedAlgorithms))
661         {
662             CLEAR_BIT(alg, *toTest);
663             continue;
664         }
665         // Process whatever is left.
666         // NOTE: since this switch will only be called if the algorithm is
667         // implemented, it is not necessary to modify this list except to comment
668         // out the algorithms for which there is no test
669         switch(alg)
670         {
671         // Symmetric block ciphers
672 #if     ALG_AES
673             case ALG_AES_VALUE:
674 #endif  // ALG_AES
675 #if     ALG_SM4
676             // if SM4 is implemented, its test is like other block ciphers but there
677             // aren't any test vectors for it yet
678 //            case ALG_SM4_VALUE:
679 #endif  // ALG_SM4
680 #if     ALG_CAMELLIA
681             // no test vectors for camellia
682 //            case ALG_CAMELLIA_VALUE:
683 #endif
684         // Symmetric modes
685 #if     !ALG_CFB
686 #   error   CFB is required in all TPM implementations
687 #endif // !ALG_CFB
688             case ALG_CFB_VALUE:
689                 if(doTest)
690                     result = TestSymmetric(alg, toTest);
691                 break;
692 #if     ALG_CTR
693             case ALG_CTR_VALUE:
694 #endif // ALG_CRT
695 #if     ALG_OFB
696             case ALG_OFB_VALUE:
697 #endif // ALG_OFB
698 #if     ALG_CBC
699             case ALG_CBC_VALUE:
700 #endif // ALG_CBC
701 #if     ALG_ECB
702             case ALG_ECB_VALUE:
703 #endif
704                 if(doTest)
705                     result = TestSymmetric(alg, toTest);
706                 else
707                     // If doing the initialization of g_toTest vector, only need
708                     // to test one of the modes for the symmetric algorithms. If
709                     // initializing for a SelfTest(FULL_TEST), allow all the modes.
710                     if(toTest == &g_toTest)
711                         CLEAR_BIT(alg, *toTest);
712                 break;
713 #if     !ALG_HMAC
714 #   error   HMAC is required in all TPM implementations
715 #endif
716             case ALG_HMAC_VALUE:
717                 // Clear the bit that indicates that HMAC is required because
718                 // HMAC is used as the basic test for all hash algorithms.
719                 CLEAR_BOTH(alg);
720                 // Testing HMAC means test the default hash
721                 if(doTest)
722                     TestHash(DEFAULT_TEST_HASH, toTest);
723                 else
```

```
724                     // If not testing, then indicate that the hash needs to be
725                     // tested because this uses HMAC
726                     SET_BOTH(DEFAULT_TEST_HASH);
727                 break;
728 #if     ALG_SHA1
729             case ALG_SHA1_VALUE:
730 #endif // ALG_SHA1
731 #if     ALG_SHA256
732             case ALG_SHA256_VALUE:
733 #endif // ALG_SHA256
734 #if     ALG_SHA384
735             case ALG_SHA384_VALUE:
736 #endif // ALG_SHA384
737 #if     ALG_SHA512
738             case ALG_SHA512_VALUE:
739 #endif // ALG_SHA512
740             // if SM3 is implemented its test is like any other hash, but there
741             // aren't any test vectors yet.
742 #if     ALG_SM3_256
743 //            case ALG_SM3_256_VALUE:
744 #endif // ALG_SM3_256
745                 if(doTest)
746                     result = TestHash(alg, toTest);
747                 break;
748     // RSA-dependent
749 #if     ALG_RSA
750             case ALG_RSA_VALUE:
751                 CLEAR_BOTH(alg);
752                 if(doTest)
753                     result = TestRsa(ALG_NULL_VALUE, toTest);
754                 else
755                     SET_BOTH(ALG_NULL_VALUE);
756                 break;
757             case ALG_RSASSA_VALUE:
758             case ALG_RSAES_VALUE:
759             case ALG_RSAPSS_VALUE:
760             case ALG_OAEP_VALUE:
761             case ALG_NULL_VALUE:    // used or RSADP
762                 if(doTest)
763                     result = TestRsa(alg, toTest);
764                 break;
765 #endif // ALG_RSA
766 #if     ALG_KDF1_SP800_108
767             case ALG_KDF1_SP800_108_VALUE:
768                 if(doTest)
769                     result = TestKDFa(toTest);
770                 break;
771 #endif // ALG_KDF1_SP800_108
772 #if     ALG_ECC
773     // ECC dependent but no tests
774     //        case ALG_ECDAA_VALUE:
775     //        case ALG_ECMQV_VALUE:
776     //        case ALG_KDF1_SP800_56a_VALUE:
777     //        case ALG_KDF2_VALUE:
778     //        case ALG_MGF1_VALUE:
779             case ALG_ECC_VALUE:
780                 CLEAR_BOTH(alg);
781                 if(doTest)
782                     result = TestEcc(ALG_ECDH_VALUE, toTest);
783                 else
784                     SET_BOTH(ALG_ECDH_VALUE);
785                 break;
786             case ALG_ECDSA_VALUE:
787             case ALG_ECDH_VALUE:
788             case ALG_ECSCHNORR_VALUE:
789 //            case ALG_SM2_VALUE:
```



```
790                     if(doTest)
791                         result = TestEcc(alg, toTest);
792                     break;
793     #endif // ALG_ECC
794                 default:
795                     CLEAR_BIT(alg, *toTest);
796                     break;
797             }
798             if(result != TPM_RC_SUCCESS)
799                 break;
800         }
801         return result;
802     }
803     #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

```
#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

```
LIB_EXPORT bigNum
BnFromBytes(
    bigNum           bn,
    const BYTE      *bytes,
    NUMBYTES         nBytes
    )
{
    const BYTE      *pFrom; // 'p' points to the least significant bytes of source
    BYTE            *pTo;   // points to least significant bytes of destination
    crypt_uword_t    size;
//

    size = (bytes != NULL) ? BYTES_TO_CRYPT_WORDS(nBytes) : 0;

    // If nothing in, nothing out
    if(bn == NULL)
        return NULL;

    // make sure things fit
    pAssert(BnGetAllocated(bn) >= size);

    if(size > 0)
    {
        // Clear the topmost word in case it is not filled with data
        bn->d[size - 1] = 0;
        // Moving the input bytes from the end of the list (LSB) end
        pFrom = bytes + nBytes - 1;
        // To the LS0 of the LSW of the bigNum.
        pTo = (BYTE *)bn->d;
        for(; nBytes != 0; nBytes--)
            *pTo++ = *pFrom--;
        // For a little-endian machine, the conversion is a straight byte
        // reversal. For a big-endian machine, we have to put the words in
        // big-endian byte order
#if BIG_ENDIAN_TPM
        {
            crypt_word_t   t;
            for(t = (crypt_word_t)size - 1; t >= 0; t--)
                bn->d[t] = SWAP_CRYPT_WORD(bn->d[t]);
        }
#endif
    }
```



```
44      BnSetTop(bn, size);
45      return bn;
46  }
```

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

```
47  LIB_EXPORT bigNum
48  BnFrom2B(
49      bigNum           bn,         // OUT:
50      const TPM2B     *a2B         // IN: number to convert
51      )
52  {
53      if(a2B != NULL)
54          return BnFromBytes(bn, a2B->buffer, a2B->size);
55      // Make sure that the number has an initialized value rather than whatever
56      // was there before
57      BnSetTop(bn, 0);    // Function accepts NULL
58      return NULL;
59  }
```

10.2.2.3.3 BnFromHex()

Convert a hex string into a *bigNum*. This is primarily used in debugging.

```
60  LIB_EXPORT bigNum
61  BnFromHex(
62      bigNum          bn,         // OUT:
63      const char      *hex        // IN:
64      )
65  {
66  #define FromHex(a)  ((a) - (((a) > 'a') ? ('a' + 10)                                \
67                                  : ((a) > 'A') ? ('A' - 10) : '0'))
68      unsigned            i;
69      unsigned            wordCount;
70      const char         *p;
71      BYTE               *d = (BYTE *)&(bn->d[0]);
72  //
73      pAssert(bn && hex);
74      i = strlen(hex);
75      wordCount = BYTES_TO_CRYPT_WORDS((i + 1) / 2);
76      if((i == 0) || (wordCount >= BnGetAllocated(bn)))
77          BnSetWord(bn, 0);
78      else
79      {
80          bn->d[wordCount - 1] = 0;
81          p = hex + i - 1;
82          for(;i > 1; i -= 2)
83          {
84              BYTE a;
85              a = FromHex(*p);
86              p--;
87              *d++ = a + (FromHex(*p) << 4);
88              p--;
89          }
90          if(i == 1)
91              *d = FromHex(*p);
92      }
93  #if !BIG_ENDIAN_TPM
94      for(i = 0; i < wordCount; i++)
95          bn->d[i] = SWAP_CRYPT_WORD(bn->d[i]);
```

```
96  #endif // BIG_ENDIAN_TPM
97      BnSetTop(bn, wordCount);
98      return bn;
99  }
```

10.2.2.3.4 BnToBytes()

This function converts a BIG_NUM to a byte array. 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).

The conversion for a little-endian machine simply requires that all significant bytes of the *bigNum* be reversed. For a big-endian machine, rather than unpack each word individually, the *bigNum* is converted to little-endian words, copied, and then converted back to big-endian.

```
100 LIB_EXPORT BOOL
101 BnToBytes(
102     bigConst             bn,
103     BYTE                *buffer,
104     NUMBYTES            *size           // This the number of bytes that are
105                                         // available in the buffer. The result
106                                         // should be this big.
107     )
108 {
109     crypt_uword_t        requiredSize;
110     BYTE                *pFrom;
111     BYTE                *pTo;
112     crypt_uword_t        count;
113 //
114     // validate inputs
115     pAssert(bn && buffer && size);
116
117     requiredSize = (BnSizeInBits(bn) + 7) / 8;
118     if(requiredSize == 0)
119     {
120         // If the input value is 0, return a byte of zero
121         *size = 1;
122         *buffer = 0;
123     }
124     else
125     {
126 #if BIG_ENDIAN_TPM
127         // Copy the constant input value into a modifiable value
128         BN_VAR(bnL, LARGEST_NUMBER_BITS * 2);
129         BnCopy(bnL, bn);
130         // byte swap the words in the local value to make them little-endian
131         for(count = 0; count < bnL->size; count++)
132             bnL->d[count] = SWAP_CRYPT_WORD(bnL->d[count]);
133         bn = (bigConst)bnL;
134 #endif
135         if(*size == 0)
136             *size = (NUMBYTES)requiredSize;
137         pAssert(requiredSize <= *size);
138         // Byte swap the number (not words but the whole value)
139         count = *size;
140         // Start from the least significant word and offset to the most significant
141         // byte which is in some high word
142         pFrom = (BYTE *)(&bn->d[0]) + requiredSize - 1;
143         pTo = buffer;
144
145         // If the number of output bytes is larger than the number bytes required
146         // for the input number, pad with zeros
147         for(count = *size; count > requiredSize; count--)
148             *pTo++ = 0;
```



```
149         // Move the most significant byte at the end of the BigNum to the next most
150         // significant byte position of the 2B and repeat for all significant bytes.
151         for(; requiredSize > 0; requiredSize--)
152             *pTo++ = *pFrom--;
153     }
154     return TRUE;
155 }
```

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.

```
156 LIB_EXPORT BOOL
157 BnTo2B(
158     bigConst         bn,                // IN:
159     TPM2B           *a2B,               // OUT:
160     NUMBYTES         size               // IN: the desired size
161     )
162 {
163     // Set the output size
164     if(bn && a2B)
165     {
166         a2B->size = size;
167         return BnToBytes(bn, a2B->buffer, &a2B->size);
168     }
169     return FALSE;
170 }
171 #if     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.

```
172 LIB_EXPORT bn_point_t    *
173 BnPointFrom2B(
174     bigPoint             ecP,        // OUT: the preallocated point structure
175     TPMS_ECC_POINT      *p           // IN: the number to convert
176     )
177 {
178     if(p == NULL)
179         return NULL;
180
181     if(NULL != ecP)
182     {
183         BnFrom2B(ecP->x, &p->x.b);
184         BnFrom2B(ecP->y, &p->y.b);
185         BnSetWord(ecP->z, 1);
186     }
187     return ecP;
188 }
```

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

```
189 LIB_EXPORT BOOL
190 BnPointTo2B(
191     TPMS_ECC_POINT  *p,             // OUT: the converted 2B structure
192     bigPoint         ecP,           // IN: the values to be converted
193     bigCurve         E              // IN: curve descriptor for the point
194     )
195 {
196     UINT16           size;
197 //
198     pAssert(p && ecP && E);
199     pAssert(BnEqualWord(ecP->z, 1));
200     // BnMsb is the bit number of the MSB. This is one less than the number of bits
201     size = (UINT16)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(AccessCurveData(E))));
202     BnTo2B(ecP->x, &p->x.b, size);
203     BnTo2B(ecP->y, &p->y.b, size);
204     return TRUE;
205 }
206 #endif // ALG_ECC
```

10.2.3 BnMath.c

10.2.3.1 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.

The 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.

10.2.3.2 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}};
```

10.2.3.3 Functions

10.2.3.3.1 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
14      for(i = 0; i < count; i++)
15      {
16          crypt_uword_t        a = op1[i];
17          crypt_uword_t        sum = a + op2[i];
18          result[i] = sum + carry;
19          // generate a carry if the sum is less than either of the inputs
20          // propagate a carry if there was a carry and the sum + carry is zero
21          // do this using bit operations rather than logical operations so that
22          // the time is about the same.
23          //              propagate term      | generate term
24          carry = ((result[i] == 0) & carry) | (sum < a);
25      }
26      return carry;
27  }
```

10.2.3.3.2 CarryProp()

Propagate a carry

```
28  static int
29  CarryProp(
30      crypt_uword_t           *result,
31      const crypt_uword_t     *op,
32      int                      count,
33      int                      carry
34      )
35  {
36      for(; count; count--)
37          carry = ((*result++ = *op++ + carry) == 0) & carry;
38      return carry;
39  }
40  static void
41  CarryResolve(
42      bigNum          result,
43      int             stop,
44      int             carry
45      )
46  {
47      if(carry)
48      {
49          pAssert((unsigned)stop < result->allocated);
50          result->d[stop++] = 1;
51      }
52      BnSetTop(result, stop);
53  }
```

10.2.3.3.3 BnAdd()

This function adds two *bigNum* values. This function always returns TRUE.

```
54  LIB_EXPORT BOOL
55  BnAdd(
56      bigNum          result,
57      bigConst        op1,
58      bigConst        op2
59      )
60  {
61      crypt_uword_t   stop;
62      int             carry;
63      const bignum_t  *n1 = op1;
64      const bignum_t  *n2 = op2;
65
```



```
66  //
67      if(n2->size > n1->size)
68      {
69          n1 = op2;
70          n2 = op1;
71      }
72      pAssert(result->allocated >= n1->size);
73      stop = MIN(n1->size, n2->allocated);
74      carry = AddSame(result->d, n1->d, n2->d, stop);
75      if(n1->size > stop)
76          carry = CarryProp(&result->d[stop], &n1->d[stop], n1->size - stop, carry);
77      CarryResolve(result, n1->size, carry);
78      return TRUE;
79  }
```

10.2.3.3.4 BnAddWord()

This function adds a word value to a *bigNum*. This function always returns TRUE.

```
80  LIB_EXPORT BOOL
81  BnAddWord(
82      bigNum          result,
83      bigConst        op,
84      crypt_uword_t   word
85      )
86  {
87      int             carry;
88  //
89      carry = (result->d[0] = op->d[0] + word) < word;
90      carry = CarryProp(&result->d[1], &op->d[1], op->size - 1, carry);
91      CarryResolve(result, op->size, carry);
92      return TRUE;
93  }
```

10.2.3.3.5 SubSame()

This function subtracts two values that have the same size.

```
94  static int
95  SubSame(
96      crypt_uword_t           *result,
97      const crypt_uword_t     *op1,
98      const crypt_uword_t     *op2,
99      int                      count
100     )
101 {
102     int                 borrow = 0;
103     int                 i;
104     for(i = 0; i < count; i++)
105     {
106         crypt_uword_t   a = op1[i];
107         crypt_uword_t   diff = a - op2[i];
108         result[i] = diff - borrow;
109         //      generate    |       propagate
110         borrow = (diff > a) | ((diff == 0) & borrow);
111     }
112     return borrow;
113 }
```


10.2.3.3.6 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.

```
114 static int
115 BorrowProp(
116     crypt_uword_t           *result,
117     const crypt_uword_t     *op,
118     int                      size,
119     int                      borrow
120     )
121 {
122     for(; size > 0; size--)
123         borrow = ((*result++ = *op++ - borrow) == MAX_CRYPT_UWORD) && borrow;
124     return borrow;
125 }
```

10.2.3.3.7 BnSub()

This function does subtraction of two *bigNum* values and returns result = op1 - op2 when op1 is greater than op2. If op2 is greater than op1, then a fault is generated. This function always returns TRUE.

```
126 LIB_EXPORT BOOL
127 BnSub(
128     bigNum           result,
129     bigConst         op1,
130     bigConst         op2
131     )
132 {
133     int             borrow;
134     crypt_uword_t   stop = MIN(op1->size, op2->allocated);
135 //
136     // Make sure that op2 is not obviously larger than op1
137     pAssert(op1->size >= op2->size);
138     borrow = SubSame(result->d, op1->d, op2->d, stop);
139     if(op1->size > stop)
140         borrow = BorrowProp(&result->d[stop], &op1->d[stop], op1->size - stop,
141                             borrow);
142     pAssert(!borrow);
143     BnSetTop(result, op1->size);
144     return TRUE;
145 }
```

10.2.3.3.8 BnSubWord()

This function subtracts a word value from a *bigNum*. This function always returns TRUE.

```
146 LIB_EXPORT BOOL
147 BnSubWord(
148     bigNum           result,
149     bigConst     op,
150     crypt_uword_t    word
151     )
152 {
153     int             borrow;
154 //
155     pAssert(op->size > 1 || word <= op->d[0]);
156     borrow = word > op->d[0];
```



```
157         result->d[0] = op->d[0] - word;
158         borrow = BorrowProp(&result->d[1], &op->d[1], op->size - 1, borrow);
159         pAssert(!borrow);
160         BnSetTop(result, op->size);
161         return TRUE;
162     }
```

10.2.3.3.9 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

```
163     LIB_EXPORT int
164     BnUnsignedCmp(
165         bigConst                op1,
166         bigConst                op2
167         )
168     {
169         int             retVal;
170         int             diff;
171         int              i;
172     //
173         pAssert((op1 != NULL) && (op2 != NULL));
174         retVal = op1->size - op2->size;
175         if(retVal == 0)
176         {
177             for(i = (int)(op1->size - 1); i >= 0; i--)
178             {
179                 diff = (op1->d[i] < op2->d[i]) ? -1 : (op1->d[i] != op2->d[i]);
180                 retVal = retVal == 0 ? diff : retVal;
181             }
182         }
183         else
184             retVal = (retVal < 0) ? -1 : 1;
185         return retVal;
186     }
```

10.2.3.3.10 BnUnsignedCmpWord()

Compare a *bigNum* to a crypt_uword_t.

Return Value	Meaning
-1	op1 is less that word
0	op1 is equal to word
1	op1 is greater than word

```
187     LIB_EXPORT int
188     BnUnsignedCmpWord(
189         bigConst            op1,
190         crypt_uword_t       word
191         )
192     {
```

```
193         if(op1->size > 1)
194             return 1;
195         else if(op1->size == 1)
196             return (op1->d[0] < word) ? -1 : (op1->d[0] > word);
197         else // op1 is zero
198             // equal if word is zero
199             return (word == 0) ? 0 : -1;
200     }
```

10.2.3.3.11 BnModWord()

This function does modular division of a big number when the modulus is a word value.

```
201     LIB_EXPORT crypt_word_t
202     BnModWord(
203         bigConst         numerator,
204         crypt_word_t     modulus
205         )
206     {
207         BN_MAX(remainder);
208         BN_VAR(mod, RADIX_BITS);
209     //
210         mod->d[0] = modulus;
211         mod->size = (modulus != 0);
212         BnDiv(NULL, remainder, numerator, mod);
213         return remainder->d[0];
214     }
```

10.2.3.3.12 Msb()

This function 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

```
215     LIB_EXPORT int
216     Msb(
217         crypt_uword_t           word
218         )
219     {
220         int             retVal = -1;
221     //
222     #if RADIX_BITS == 64
223         if(word & 0xffffffff00000000) { retVal += 32; word >>= 32; }
224     #endif
225         if(word & 0xffff0000) { retVal += 16; word >>= 16; }
226         if(word & 0x0000ff00) { retVal += 8; word >>= 8; }
227         if(word & 0x000000f0) { retVal += 4; word >>= 4; }
228         if(word & 0x0000000c) { retVal += 2; word >>= 2; }
229         if(word & 0x00000002) { retVal += 1; word >>= 1; }
230         return retVal + (int)word;
231     }
```

10.2.3.3.13 BnMsb()

This function returns the number of the MSb() of a *bigNum* value.

Return Value	Meaning
-1	the word was zero or *bn* was NULL
n	the bit number of the most significant bit in the word

```
LIB_EXPORT int
BnMsb(
    bigConst            bn
    )
{
    // If the value is NULL, or the size is zero then treat as zero and return -1
    if(bn != NULL && bn->size > 0)
    {
        int         retVal = Msb(bn->d[bn->size - 1]);
        retVal += (bn->size - 1) * RADIX_BITS;
        return retVal;
    }
    else
        return -1;
}
```

10.2.3.3.14 BnSizeInBits()

This function returns the number of bits required to hold a number. It is one greater than the Msb.

```
LIB_EXPORT unsigned
BnSizeInBits(
    bigConst                n
    )
{
    int     bits = BnMsb(n) + 1;
//
    return bits < 0? 0 : (unsigned)bits;
}
```

10.2.3.3.15 BnSetWord()

Change the value of a bignum_t to a word value.

```
LIB_EXPORT bigNum
BnSetWord(
    bigNum              n,
    crypt_uword_t       w
    )
{
    if(n != NULL)
    {
        pAssert(n->allocated > 1);
        n->d[0] = w;
        BnSetTop(n, (w != 0) ? 1 : 0);
    }
    return n;
}
```

10.2.3.3.16 BnSetBit()

This function will SET a bit in a *bigNum*. Bit 0 is the least-significant bit in the 0th digit_t. The function always return TRUE

```
LIB_EXPORT BOOL
```

```
271 BnSetBit(
272     bigNum          bn,         // IN/OUT: big number to modify
273     unsigned int    bitNum      // IN: Bit number to SET
274     )
275 {
276     crypt_uword_t           offset = bitNum / RADIX_BITS;
277     pAssert(bn->allocated * RADIX_BITS >= bitNum);
278     // Grow the number if necessary to set the bit.
279     while(bn->size <= offset)
280         bn->d[bn->size++] = 0;
281     bn->d[offset] |= (1 << RADIX_MOD(bitNum));
282     return TRUE;
283 }
```

10.2.3.3.17 BnTestBit()

This function is used to check to see if a bit is SET in a bignum_t. The 0th bit is the LSb() of d[0].

Return Value	Meaning
TRUE(1)	the bit is set
FALSE(0)	the bit is not set or the number is out of range

```
284 LIB_EXPORT BOOL
285 BnTestBit(
286     bigNum              bn,         // IN: number to check
287     unsigned int        bitNum      // IN: bit to test
288     )
289 {
290     crypt_uword_t        offset = RADIX_DIV(bitNum);
291 //
292     if(bn->size > offset)
293         return ((bn->d[offset] & (((crypt_uword_t)1) << RADIX_MOD(bitNum))) != 0);
294     else
295         return FALSE;
296 }
```

10.2.3.3.18 BnMaskBits()

This function is used 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(1)	result masked
FALSE(0)	the input was not as large as the mask

```
297 LIB_EXPORT BOOL
298 BnMaskBits(
299     bigNum          bn,         // IN/OUT: number to mask
300     crypt_uword_t   maskBit     // IN: the bit number for the mask.
301     )
302 {
303     crypt_uword_t   finalSize;
304     BOOL            retVal;
305
306     finalSize = BITS_TO_CRYPT_WORDS(maskBit);
307     retVal = (finalSize <= bn->allocated);
```



```
308     if(retVal && (finalSize > 0))
309     {
310         crypt_uword_t   mask;
311         mask = ~((crypt_uword_t)0) >> RADIX_MOD(maskBit);
312         bn->d[finalSize - 1] &= mask;
313     }
314     BnSetTop(bn, finalSize);
315     return retVal;
316 }
```

10.2.3.3.19 BnShiftRight()

This function will shift a *bigNum* to the right by the *shiftAmount*. This function always returns TRUE.

```
317 LIB_EXPORT BOOL
318 BnShiftRight(
319     bigNum          result,
320     bigConst        toShift,
321     uint32_t        shiftAmount
322     )
323 {
324     uint32_t        offset = (shiftAmount >> RADIX_LOG2);
325     uint32_t        i;
326     uint32_t        shiftIn;
327     crypt_uword_t   finalSize;
328 //
329     shiftAmount = shiftAmount & RADIX_MASK;
330     shiftIn = RADIX_BITS - shiftAmount;
331
332     // The end size is toShift->size - offset less one additional
333     // word if the shiftAmount would make the upper word == 0
334     if(toShift->size > offset)
335     {
336         finalSize = toShift->size - offset;
337         finalSize -= (toShift->d[toShift->size - 1] >> shiftAmount) == 0 ? 1 : 0;
338     }
339     else
340         finalSize = 0;
341
342     pAssert(finalSize <= result->allocated);
343     if(finalSize != 0)
344     {
345         for(i = 0; i < finalSize; i++)
346         {
347             result->d[i] = (toShift->d[i + offset] >> shiftAmount)
348                 | (toShift->d[i + offset + 1] << shiftIn);
349         }
350         if(offset == 0)
351             result->d[i] = toShift->d[i] >> shiftAmount;
352     }
353     BnSetTop(result, finalSize);
354     return TRUE;
355 }
```

10.2.3.3.20 BnGetRandomBits()

This function gets random bits for use in various places. To make sure that the number is generated in a portable format, it is created as a TPM2B and then converted to the internal format.

One consequence of the generation scheme is that, if the number of bits requested is not a multiple of 8, then the high-order bits are set to zero. This would come into play when generating a 521-bit ECC key. A 66-byte (528-bit) value is generated an the high order 7 bits are masked off (CLEAR).

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
LIB_EXPORT BOOL
BnGetRandomBits(
    bigNum          n,
    size_t          bits,
    RAND_STATE      *rand
)
{
    // Since this could be used for ECC key generation using the extra bits method,
    // make sure that the value is large enough
    TPM2B_TYPE(LARGEST, LARGEST_NUMBER + 8);
    TPM2B_LARGEST    large;
//
    large.b.size = (UINT16)BITS_TO_BYTES(bits);
    if(DRBG_Generate(rand, large.t.buffer, large.t.size) == large.t.size)
    {
        if(BnFrom2B(n, &large.b) != NULL)
        {
            if(BnMaskBits(n, bits))
                return TRUE;
        }
    }
    return FALSE;
}
```

10.2.3.3.21 BnGenerateRandomInRange()

This function is used 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.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure *(limit* is too small)

```
LIB_EXPORT BOOL
BnGenerateRandomInRange(
    bigNum          dest,
    bigConst        limit,
    RAND_STATE      *rand
    )
{
    size_t   bits = BnSizeInBits(limit);
//
    if(bits < 2)
    {
        BnSetWord(dest, 0);
        return FALSE;
    }
    else
    {
        do
        {
            BnGetRandomBits(dest, bits, rand);
```



```
398         } while(BnEqualZero(dest) || BnUnsignedCmp(dest, limit) >= 0);
399     }
400     return TRUE;
401 }
```

10.2.4 BnMemory.c

10.2.4.1 Introduction

This file contains the memory setup functions used by the *bigNum* functions in CryptoEngine()

10.2.4.2 Includes

```
#include "Tpm.h"
```

10.2.4.3 Functions

10.2.4.3.1 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;
}
```

10.2.4.3.2 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;
}
```


10.2.4.3.3 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  }
```

10.2.4.3.4 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  }
```

10.2.4.3.5 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      #if     ALG_ECC
```

10.2.4.3.6 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     }
```

10.2.4.3.7 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 // ALG_ECC
```

10.2.5 CryptCmac.c

10.2.5.1 Introduction

This file contains the implementation of the message authentication codes based on a symmetric block cipher. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.2.5.2 Includes, Defines, and Typedefs

```
1   #define _CRYPT_HASH_C_
2   #include "Tpm.h"
3   #include "CryptSym.h"
4   #if     ALG_CMAC
```

10.2.5.3 Functions

10.2.5.3.1 CryptCmacStart()

This is the function to start the CMAC sequence operation. It initializes the dispatch functions for the data and end operations for CMAC and initializes the parameters that are used for the processing of data, including the key, key size and block cipher algorithm.

```
5   UINT16
6   CryptCmacStart(
7       SMAC_STATE          *state,
8       TPMU_PUBLIC_PARMS   *keyParms,
9       TPM_ALG_ID           macAlg,
10      TPM2B               *key
11  )
12  {
13      tpmCmacState_t      *cState = &state->state.cmac;
14      TPMT_SYM_DEF_OBJECT *def = &keyParms->symDetail.sym;
15  //
16      if(macAlg != TPM_ALG_CMAC)
17          return 0;
18      // set up the encryption algorithm and parameters
19      cState->symAlg = def->algorithm;
20      cState->keySizeBits = def->keyBits.sym;
21      cState->iv.t.size = CryptGetSymmetricBlockSize(def->algorithm,
22                                                     def->keyBits.sym);
23      MemoryCopy2B(&cState->symKey.b, key, sizeof(cState->symKey.t.buffer));
24
25      // Set up the dispatch methods for the CMAC
26      state->smacMethods.data = CryptCmacData;
27      state->smacMethods.end = CryptCmacEnd;
28      return cState->iv.t.size;
29  }
```

10.2.5.3.2 CryptCmacData()

This function is used to add data to the CMAC sequence computation. The function will XOR new data into the IV. If the buffer is full, and there is additional input data, the data is encrypted into the IV buffer, the new data is then XOR into the IV. When the data runs out, the function returns without encrypting even if the buffer is full. The last data block of a sequence will not be encrypted until the call to CryptCmacEnd(). This is to allow the proper subkey to be computed and applied before the last block is encrypted.

```
30  void
31  CryptCmacData(
32      SMAC_STATES          *state,
33      UINT32                size,
34      const BYTE           *buffer
35  )
36  {
37      tpmCmacState_t           *cmacState = &state->cmac;
38      TPM_ALG_ID                algorithm = cmacState->symAlg;
39      BYTE                     *key = cmacState->symKey.t.buffer;
40      UINT16                    keySizeInBits = cmacState->keySizeBits;
41      tpmCryptKeySchedule_t     keySchedule;
42      TpmCryptSetSymKeyCall_t   encrypt;
43  //
44      SELECT(ENCRYPT);
45      while(size > 0)
46      {
47          if(cmacState->bcount == cmacState->iv.t.size)
48          {
49              ENCRYPT(&keySchedule, cmacState->iv.t.buffer, cmacState->iv.t.buffer);
50              cmacState->bcount = 0;
51          }
52          for(;(size > 0) && (cmacState->bcount < cmacState->iv.t.size);
53              size--, cmacState->bcount++)
54          {
55              cmacState->iv.t.buffer[cmacState->bcount] ^= *buffer++;
56          }
57      }
58  }
```

10.2.5.3.3 CryptCmacEnd()

This is the completion function for the CMAC. It does padding, if needed, and selects the subkey to be applied before the last block is encrypted.

```
59  UINT16
60  CryptCmacEnd(
61      SMAC_STATES              *state,
62      UINT32                    outSize,
63      BYTE                     *outBuffer
64  )
65  {
66      tpmCmacState_t           *cState = &state->cmac;
67      // Need to set algorithm, key, and keySizeInBits in the local context so that
68      // the SELECT and ENCRYPT macros will work here
69      TPM_ALG_ID                algorithm = cState->symAlg;
70      BYTE                     *key = cState->symKey.t.buffer;
71      UINT16                    keySizeInBits = cState->keySizeBits;
72      tpmCryptKeySchedule_t     keySchedule;
73      TpmCryptSetSymKeyCall_t   encrypt;
74      TPM2B_IV                  subkey = {{0, {0}}};
75      BOOL                      xorVal;
76      UINT16                    i;
77
78      subkey.t.size = cState->iv.t.size;
79      // Encrypt a block of zero
80      SELECT(ENCRYPT);
81      ENCRYPT(&keySchedule, subkey.t.buffer, subkey.t.buffer);
82
83      // shift left by 1 and XOR with 0x0...87 if the MSb was 0
84      xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
85      ShiftLeft(&subkey.b);
86      subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
87      // this is a sanity check to make sure that the algorithm is working properly.
```



```
 88         // remove this check when debug is done
 89         pAssert(cState->bcount <= cState->iv.t.size);
 90         // If the buffer is full then no need to compute subkey 2.
 91         if(cState->bcount < cState->iv.t.size)
 92         {
 93             //Pad the data
 94             cState->iv.t.buffer[cState->bcount++] ^= 0x80;
 95             // The rest of the data is a pad of zero which would simply be XORed
 96             // with the iv value so nothing to do...
 97             // Now compute K2
 98             xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
 99             ShiftLeft(&subkey.b);
100             subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
101         }
102         // XOR the subkey into the IV
103         for(i = 0; i < subkey.t.size; i++)
104             cState->iv.t.buffer[i] ^= subkey.t.buffer[i];
105         ENCRYPT(&keySchedule, cState->iv.t.buffer, cState->iv.t.buffer);
106         i = (UINT16)MIN(cState->iv.t.size, outSize);
107         MemoryCopy(outBuffer, cState->iv.t.buffer, i);
108
109         return i;
110     }
111     #endif
  8
```

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

```
#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

```
static TPM_RC
CryptHmacSign(
    TPMT_SIGNATURE      *signature,     // OUT: signature
    OBJECT              *signKey,       // IN: HMAC key sign the hash
    TPM2B_DIGEST        *hashData       // IN: hash to be signed
    )
{
    HMAC_STATE      hmacState;
    UINT32          digestSize;

    digestSize = CryptHmacStart2B(&hmacState, signature->signature.any.hashAlg,
                                  &signKey->sensitive.sensitive.bits.b);
    CryptDigestUpdate2B(&hmacState.hashState, &hashData->b);
    CryptHmacEnd(&hmacState, digestSize,
                 (BYTE *)&signature->signature.hmac.digest);
    return TPM_RC_SUCCESS;
}
```

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

```
static TPM_RC
CryptHMACVerifySignature(
    OBJECT              *signKey,       // IN: HMAC key signed the hash
    TPM2B_DIGEST        *hashData,      // IN: digest being verified
    TPMT_SIGNATURE      *signature      // IN: signature to be verified
    )
{
```



```
26      TPMT_SIGNATURE           test;
27      TPMT_KEYEDHASH_SCHEME   *keyScheme =
28                              &signKey->publicArea.parameters.keyedHashDetail.scheme;
29  //
30      if((signature->sigAlg != ALG_HMAC_VALUE)
31         || (signature->signature.hmac.hashAlg == ALG_NULL_VALUE))
32          return TPM_RC_SCHEME;
33      // This check is not really needed for verification purposes. However, it does
34      // prevent someone from trying to validate a signature using a weaker hash
35      // algorithm than otherwise allowed by the key. That is, a key with a scheme
36      // other than TMP_ALG_NULL can only be used to validate signatures that have
37      // a matching scheme.
38      if((keyScheme->scheme != ALG_NULL_VALUE)
39         && ((keyScheme->scheme != signature->sigAlg)
40             || (keyScheme->details.hmac.hashAlg
41                 != signature->signature.any.hashAlg)))
42          return TPM_RC_SIGNATURE;
43      test.sigAlg = signature->sigAlg;
44      test.signature.hmac.hashAlg = signature->signature.hmac.hashAlg;
45
46      CryptHmacSign(&test, signKey, hashData);
47
48      // Compare digest
49      if(!MemoryEqual(&test.signature.hmac.digest,
50                      &signature->signature.hmac.digest,
51                      CryptHashGetDigestSize(signature->signature.any.hashAlg)))
52          return TPM_RC_SIGNATURE;
53
54      return TPM_RC_SUCCESS;
55  }
```

10.2.6.3.3 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Error Returns	Meaning
TPM_RC_NO_RESULT	cannot get values from random number generator
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```
56  static TPM_RC
57  CryptGenerateKeyedHash(
58      TPMT_PUBLIC             *publicArea,        // IN/OUT: the public area template
59                                                  //     for the new key.
60      TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
61      TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
62      RAND_STATE              *rand               // IN: "entropy" source
63      )
64  {
65      TPMT_KEYEDHASH_SCHEME   *scheme;
66      TPM_ALG_ID               hashAlg;
67      UINT16                   hashBlockSize;
68      UINT16                   digestSize;
69
70      scheme = &publicArea->parameters.keyedHashDetail.scheme;
71
72      if(publicArea->type != ALG_KEYEDHASH_VALUE)
73          return TPM_RC_FAILURE;
74
75      // Pick the limiting hash algorithm
76      if(scheme->scheme == ALG_NULL_VALUE)
77          hashAlg = publicArea->nameAlg;
78      else if(scheme->scheme == ALG_XOR_VALUE)
```



```
79          hashAlg = scheme->details.xor.hashAlg;
80      else
81          hashAlg = scheme->details.hmac.hashAlg;
82      hashBlockSize = CryptHashGetBlockSize(hashAlg);
83      digestSize = CryptHashGetDigestSize(hashAlg);
84
85      // if this is a signing or a decryption key, then the limit
86      // for the data size is the block size of the hash. This limit
87      // is set because larger values have lower entropy because of the
88      // HMAC function. The lower limit is 1/2 the size of the digest
89      //
90      //If the user provided the key, check that it is a proper size
91      if(sensitiveCreate->data.t.size != 0)
92      {
93          if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
94             || IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
95          {
96              if(sensitiveCreate->data.t.size > hashBlockSize)
97                  return TPM_RC_SIZE;
98  #if 0   // May make this a FIPS-mode requirement
99              if(sensitiveCreate->data.t.size < (digestSize / 2))
100                 return TPM_RC_SIZE;
101 #endif
102         }
103         // If this is a data blob, then anything that will get past the unmarshaling
104         // is OK
105         MemoryCopy2B(&sensitive->sensitive.bits.b, &sensitiveCreate->data.b,
106                      sizeof(sensitive->sensitive.bits.t.buffer));
107     }
108     else
109     {
110         // The TPM is going to generate the data so set the size to be the
111         // size of the digest of the algorithm
112         sensitive->sensitive.bits.t.size =
113             DRBG_Generate(rand, sensitive->sensitive.bits.t.buffer, digestSize);
114         if(sensitive->sensitive.bits.t.size == 0)
115             return TPM_RC_NO_RESULT;
116     }
117     return TPM_RC_SUCCESS;
118 }
```

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.

```
119 BOOL
120 CryptIsSchemeAnonymous(
121     TPM_ALG_ID        scheme          // IN: the scheme algorithm to test
122     )
123 {
124     return scheme == ALG_ECDAA_VALUE;
125 }
```

10.2.6.4 Symmetric Functions

10.2.6.4.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```
126 void
127 ParmDecryptSym(
```



```
128         TPM_ALG_ID        symAlg,         // IN: the symmetric algorithm
129         TPM_ALG_ID        hash,           // IN: hash algorithm for KDFa
130         UINT16            keySizeInBits,  // IN: the key size in bits
131         TPM2B            *key,            // IN: KDF HMAC key
132         TPM2B            *nonceCaller,    // IN: nonce caller
133         TPM2B            *nonceTpm,       // IN: nonce TPM
134         UINT32            dataSize,       // IN: size of parameter buffer
135         BYTE             *data            // OUT: buffer to be decrypted
136         )
137     {
138         // KDF output buffer
139         // It contains parameters for the CFB encryption
140         // From MSB to LSB, they are the key and iv
141         BYTE              symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
142         // Symmetric key size in byte
143         UINT16            keySize = (keySizeInBits + 7) / 8;
144         TPM2B_IV          iv;
145
146         iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
147         // If there is decryption to do...
148         if(iv.t.size > 0)
149         {
150             // Generate key and iv
151             CryptKDFa(hash, key, CFB_KEY, nonceCaller, nonceTpm,
152                       keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
153             MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
154
155             CryptSymmetricDecrypt(data, symAlg, keySizeInBits, symParmString,
156                                   &iv, ALG_CFB_VALUE, dataSize, data);
157         }
158         return;
159     }
```

10.2.6.4.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
160     void
161     ParmEncryptSym(
162         TPM_ALG_ID        symAlg,         // IN: symmetric algorithm
163         TPM_ALG_ID        hash,           // IN: hash algorithm for KDFa
164         UINT16            keySizeInBits,  // IN: symmetric key size in bits
165         TPM2B            *key,            // IN: KDF HMAC key
166         TPM2B            *nonceCaller,    // IN: nonce caller
167         TPM2B            *nonceTpm,       // IN: nonce TPM
168         UINT32            dataSize,       // IN: size of parameter buffer
169         BYTE             *data            // OUT: buffer to be encrypted
170         )
171     {
172         // KDF output buffer
173         // It contains parameters for the CFB encryption
174         BYTE              symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
175
176         // Symmetric key size in bytes
177         UINT16            keySize = (keySizeInBits + 7) / 8;
178
179         TPM2B_IV          iv;
180
181         iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
182         // See if there is any encryption to do
183         if(iv.t.size > 0)
184         {
185             // Generate key and iv
186             CryptKDFa(hash, key, CFB_KEY, nonceTpm, nonceCaller,
```

```
187                     keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
188         MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
189
190         CryptSymmetricEncrypt(data, symAlg, keySizeInBits, symParmString, &iv,
191                               ALG_CFB_VALUE, dataSize, data);
192     }
193     return;
194 }
```

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_RC_NO_RESULT	cannot get a random value
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area
TPM_RC_KEY	provided key value is not allowed

```
195 static TPM_RC
196 CryptGenerateKeySymmetric(
197     TPMT_PUBLIC             *publicArea,        // IN/OUT: The public area template
198                                                 //     for the new key.
199     TPMT_SENSITIVE          *sensitive,         // OUT: sensitive area
200     TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation data
201     RAND_STATE              *rand               // IN: the "entropy" source for
202     )
203 {
204     UINT16          keyBits = publicArea->parameters.symDetail.sym.keyBits.sym;
205     TPM_RC          result;
206 //
207     // only do multiples of RADIX_BITS
208     if((keyBits % RADIX_BITS) != 0)
209         return TPM_RC_KEY_SIZE;
210     // If this is not a new key, then the provided key data must be the right size
211     if(sensitiveCreate->data.t.size != 0)
212     {
213         result = CryptSymKeyValidate(&publicArea->parameters.symDetail.sym,
214                                      (TPM2B_SYM_KEY *)&sensitiveCreate->data);
215         if(result == TPM_RC_SUCCESS)
216             MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
217                          sizeof(sensitive->sensitive.sym.t.buffer));
218     }
219 #if     ALG_TDES
220     else if(publicArea->parameters.symDetail.sym.algorithm == ALG_TDES_VALUE)
221     {
222         result = CryptGenerateKeyDes(publicArea, sensitive, rand);
223     }
224 #endif
225     else
226     {
227         sensitive->sensitive.sym.t.size =
228             DRBG_Generate(rand, sensitive->sensitive.sym.t.buffer,
229                           BITS_TO_BYTES(keyBits));
230         result = (sensitive->sensitive.sym.t.size == 0)
231             ? TPM_RC_NO_RESULT : TPM_RC_SUCCESS;
232     }
233     return result;
234 }
```


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.

```
235 void
236 CryptXORObfuscation(
237     TPM_ALG_ID       hash,          // IN: hash algorithm for KDF
238     TPM2B           *key,           // IN: KDF key
239     TPM2B           *contextU,      // IN: contextU
240     TPM2B           *contextV,      // IN: contextV
241     UINT32           dataSize,      // IN: size of data buffer
242     BYTE            *data           // IN/OUT: data to be XORed in place
243     )
244 {
245     BYTE             mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
246     BYTE            *pm;
247     UINT32           i;
248     UINT32           counter = 0;
249     UINT16           hLen = CryptHashGetDigestSize(hash);
250     UINT32           requestSize = dataSize * 8;
251     INT32            remainBytes = (INT32)dataSize;
252
253     pAssert((key != NULL) && (data != NULL) && (hLen != 0));
254
255     // Call KDFa to generate XOR mask
256     for(; remainBytes > 0; remainBytes -= hLen)
257     {
258         // Make a call to KDFa to get next iteration
259         CryptKDFa(hash, key, XOR_KEY, contextU, contextV,
260                   requestSize, mask, &counter, TRUE);
261
262     // XOR next piece of the data
263         pm = mask;
264         for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
265             *data++ ^= *pm++;
266     }
267     return;
268 }
```

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(1)	initializations succeeded
FALSE(0)	initialization failed and caller should place the TPM into Failure Mode

```
269 BOOL
270 CryptInit(
271     void
272     )
273 {
274     BOOL         ok;
275     // Initialize the vector of implemented algorithms
```

```
    AlgorithmGetImplementedVector(&g_implementedAlgorithms);

    // Indicate that all test are necessary
    CryptInitializeToTest();

    // Do any library initializations that are necessary. If any fails,
    // the caller should go into failure mode;
    ok = SupportLibInit();
    ok = ok && CryptSymInit();
    ok = ok && CryptRandInit();
    ok = ok && CryptHashInit();
#if     ALG_RSA
    ok = ok && CryptRsaInit();
#endif // ALG_RSA
#if     ALG_ECC
    ok = ok && CryptEccInit();
#endif // ALG_ECC
    return ok;
}
```

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(1)	startup succeeded
FALSE(0)	startup failed and caller should place the TPM into Failure Mode

```
BOOL
CryptStartup(
    STARTUP_TYPE     type           // IN: the startup type
    )
{
    BOOL            OK;
    NOT_REFERENCED(type);

    OK = CryptSymStartup() && CryptRandStartup() && CryptHashStartup()
#if     ALG_RSA
        &&  CryptRsaStartup()
#endif // ALG_RSA
#if     ALG_ECC
        &&  CryptEccStartup()
#endif // ALG_ECC
        ;
#if     ALG_ECC
    // Don't directly check for SU_RESET because that is the default
    if(OK && (type != SU_RESTART) && (type != SU_RESUME))
    {
    // If the shutdown was orderly, then the values recovered from NV will
    // be OK to use.
        // Get a new  random commit nonce
        gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
        CryptRandomGenerate(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
        // Reset the counter and commit array
        gr.commitCounter = 0;
        MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
    }
#endif // ALG_ECC
    return OK;
}
```


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(1)	if it is an asymmetric algorithm
FALSE(0)	if it is not an asymmetric algorithm

```
BOOL
CryptIsAsymAlgorithm(
    TPM_ALG_ID        algID           // IN: algorithm ID
    )
{
    switch(algID)
    {
#if     ALG_RSA
        case ALG_RSA_VALUE:
#endif
#if     ALG_ECC
        case ALG_ECC_VALUE:
#endif
            return TRUE;
            break;
        default:
            break;
    }
    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
    )
```

```
354 {
355     TPMT_RSA_DECRYPT         scheme;
356     TPM_RC                   result = TPM_RC_SUCCESS;
357 //
358     if(data == NULL || secret == NULL)
359         return TPM_RC_FAILURE;
360
361     // The output secret value has the size of the digest produced by the nameAlg.
362     data->t.size = CryptHashGetDigestSize(encryptKey->publicArea.nameAlg);
363     // The encryption scheme is OAEP using the nameAlg of the encrypt key.
364     scheme.scheme = ALG_OAEP_VALUE;
365     scheme.details.anySig.hashAlg = encryptKey->publicArea.nameAlg;
366
367     if(!IS_ATTRIBUTE(encryptKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
368         return TPM_RC_ATTRIBUTES;
369     switch(encryptKey->publicArea.type)
370     {
371 #if     ALG_RSA
372         case ALG_RSA_VALUE:
373         {
374             // Create secret data from RNG
375             CryptRandomGenerate(data->t.size, data->t.buffer);
376
377             // Encrypt the data by RSA OAEP into encrypted secret
378             result = CryptRsaEncrypt((TPM2B_PUBLIC_KEY_RSA *)secret, &data->b,
379                                      encryptKey, &scheme, label, NULL);
380         }
381         break;
382 #endif // ALG_RSA
383
384 #if     ALG_ECC
385         case ALG_ECC_VALUE:
386         {
387             TPMS_ECC_POINT      eccPublic;
388             TPM2B_ECC_PARAMETER eccPrivate;
389             TPMS_ECC_POINT      eccSecret;
390             BYTE                *buffer = secret->t.secret;
391
392             // Need to make sure that the public point of the key is on the
393             // curve defined by the key.
394             if(!CryptEccIsPointOnCurve(
395                 encryptKey->publicArea.parameters.eccDetail.curveID,
396                 &encryptKey->publicArea.unique.ecc))
397                 result = TPM_RC_KEY;
398             else
399             {
400                 // Call crypto engine to create an auxiliary ECC key
401                 // We assume crypt engine initialization should always success.
402                 // Otherwise, TPM should go to failure mode.
403
404                 CryptEccNewKeyPair(&eccPublic, &eccPrivate,
405                                    encryptKey->publicArea.parameters.eccDetail.curveID);
406                 // Marshal ECC public to secret structure. This will be used by the
407                 // recipient to decrypt the secret with their private key.
408                 secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
409
410                 // Compute ECDH shared secret which is R = [d]Q where d is the
411                 // private part of the ephemeral key and Q is the public part of a
412                 // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
413                 // because the auxiliary ECC key is just created according to the
414                 // parameters of input ECC encrypt key.
415                 if(CryptEccPointMultiply(&eccSecret,
416                              encryptKey->publicArea.parameters.eccDetail.curveID,
417                              &encryptKey->publicArea.unique.ecc, &eccPrivate,
418                              NULL, NULL)
419                    != TPM_RC_SUCCESS)
```



```
420                     result = TPM_RC_KEY;
421                 else
422                 {
423                     // The secret value is computed from Z using KDFe as:
424                     // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
425                     // Where:
426                     //  HashID  the nameAlg of the decrypt key
427                     //  Z   the x coordinate (Px) of the product (P) of the point
428                     //      (Q) of the secret and the private x coordinate (de,V)
429                     //      of the decryption key
430                     //  Use a null-terminated string containing "SECRET"
431                     //  PartyUInfo  the x coordinate of the point in the secret
432                     //              (Qe,U )
433                     //  PartyVInfo  the x coordinate of the public key (Qs,V )
434                     //  bits    the number of bits in the digest of HashID
435                     // Retrieve seed from KDFe
436                     CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
437                               label, &eccPublic.x.b,
438                               &encryptKey->publicArea.unique.ecc.x.b,
439                               data->t.size * 8, data->t.buffer);
440                 }
441             }
442         }
443         break;
444 #endif // ALG_ECC
445         default:
446             FAIL(FATAL_ERROR_INTERNAL);
447             break;
448     }
449     return result;
450 }
```

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

```
451 TPM_RC
452 CryptSecretDecrypt(
453     OBJECT                  *decryptKey,    // IN: decrypt key
454     TPM2B_NONCE             *nonceCaller,   // IN: nonceCaller.  It is needed for
455                                             //     symmetric decryption.  For
456                                             //     asymmetric decryption, this
```

```
457                                                 //     parameter is NULL
458         const TPM2B             *label,         // IN: a value for L
459         TPM2B_ENCRYPTED_SECRET  *secret,        // IN: input secret
460         TPM2B_DATA              *data           // OUT: decrypted secret value
461         )
462     {
463         TPM_RC      result = TPM_RC_SUCCESS;
464
465         // Decryption for secret
466         switch(decryptKey->publicArea.type)
467         {
468     #if     ALG_RSA
469             case ALG_RSA_VALUE:
470             {
471                 TPMT_RSA_DECRYPT        scheme;
472                 TPMT_RSA_SCHEME         *keyScheme
473                     = &decryptKey->publicArea.parameters.rsaDetail.scheme;
474                 UINT16                   digestSize;
475
476                 scheme = *(TPMT_RSA_DECRYPT *)keyScheme;
477                 // If the key scheme is ALG_NULL_VALUE, set the scheme to OAEP and
478                 // set the algorithm to the name algorithm.
479                 if(scheme.scheme == ALG_NULL_VALUE)
480                 {
481                     // Use OAEP scheme
482                     scheme.scheme = ALG_OAEP_VALUE;
483                     scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
484                 }
485                 // use the digestSize as an indicator of whether or not the scheme
486                 // is using a supported hash algorithm.
487                 // Note: depending on the scheme used for encryption, a hashAlg might
488                 // not be needed. However, the return value has to have some upper
489                 // limit on the size. In this case, it is the size of the digest of the
490                 // hash algorithm. It is checked after the decryption is done but, there
491                 // is no point in doing the decryption if the size is going to be
492                 // 'wrong' anyway.
493                 digestSize = CryptHashGetDigestSize(scheme.details.oaep.hashAlg);
494                 if(scheme.scheme != ALG_OAEP_VALUE || digestSize == 0)
495                     return TPM_RC_SCHEME;
496
497                 // Set the output buffer capacity
498                 data->t.size = sizeof(data->t.buffer);
499
500                 // Decrypt seed by RSA OAEP
501                 result = CryptRsaDecrypt(&data->b, &secret->b,
502                                          decryptKey, &scheme, label);
503                 if((result == TPM_RC_SUCCESS) && (data->t.size > digestSize))
504                     result = TPM_RC_VALUE;
505             }
506             break;
507     #endif // ALG_RSA
508     #if     ALG_ECC
509             case ALG_ECC_VALUE:
510             {
511                 TPMS_ECC_POINT       eccPublic;
512                 TPMS_ECC_POINT       eccSecret;
513                 BYTE                *buffer = secret->t.secret;
514                 INT32                size = secret->t.size;
515
516                 // Retrieve ECC point from secret buffer
517                 result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
518                 if(result == TPM_RC_SUCCESS)
519                 {
520                     result = CryptEccPointMultiply(&eccSecret,
521                                  decryptKey->publicArea.parameters.eccDetail.curveID,
522                                  &eccPublic, &decryptKey->sensitive.sensitive.ecc,
```



```
523                                 NULL, NULL);
524                     if(result == TPM_RC_SUCCESS)
525                     {
526                         // Set the size of the "recovered" secret value to be the size
527                         // of the digest produced by the nameAlg.
528                         data->t.size =
529                             CryptHashGetDigestSize(decryptKey->publicArea.nameAlg);
530
531                     // The secret value is computed from Z using KDFe as:
532                     // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
533                     // Where:
534                     //  HashID -- the nameAlg of the decrypt key
535                     //  Z --  the x coordinate (Px) of the product (P) of the point
536                     //        (Q) of the secret and the private x coordinate (de,V)
537                     //        of the decryption key
538                     //  Use -- a null-terminated string containing "SECRET"
539                     //  PartyUInfo -- the x coordinate of the point in the secret
540                     //               (Qe,U )
541                     //  PartyVInfo -- the x coordinate of the public key (Qs,V )
542                     //  bits -- the number of bits in the digest of HashID
543                     // Retrieve seed from KDFe
544                         CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
545                                   &eccPublic.x.b,
546                                   &decryptKey->publicArea.unique.ecc.x.b,
547                                   data->t.size * 8, data->t.buffer);
548                     }
549                 }
550             }
551             break;
552     #endif // ALG_ECC
553     #if     !ALG_KEYEDHASH
554     #   error   "KEYEDHASH support is required"
555     #endif
556             case ALG_KEYEDHASH_VALUE:
557                 // The seed size can not be bigger than the digest size of nameAlg
558                 if(secret->t.size >
559                    CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
560                     result = TPM_RC_VALUE;
561                 else
562                 {
563                     // Retrieve seed by XOR Obfuscation:
564                     //    seed = XOR(secret, hash, key, nonceCaller, nullNonce)
565                     //    where:
566                     //    secret  the secret parameter from the TPM2_StartAuthHMAC
567                     //            command that contains the seed value
568                     //    hash    nameAlg  of tpmKey
569                     //    key     the key or data value in the object referenced by
570                     //            entityHandle in the TPM2_StartAuthHMAC command
571                     //    nonceCaller the parameter from the TPM2_StartAuthHMAC command
572                     //    nullNonce   a zero-length nonce
573                     // XOR Obfuscation in place
574                     CryptXORObfuscation(decryptKey->publicArea.nameAlg,
575                                         &decryptKey->sensitive.sensitive.bits.b,
576                                         &nonceCaller->b, NULL,
577                                         secret->t.size, secret->t.secret);
578                     // Copy decrypted seed
579                     MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
580                 }
581                 break;
582             case ALG_SYMCIPHER_VALUE:
583             {
584                 TPM2B_IV                iv = {{0}};
585                 TPMT_SYM_DEF_OBJECT     *symDef;
586                 // The seed size can not be bigger than the digest size of nameAlg
587                 if(secret->t.size >
588                    CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
```

```
589                     result = TPM_RC_VALUE;
590                 else
591                 {
592                     symDef = &decryptKey->publicArea.parameters.symDetail.sym;
593                     iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
594                                                            symDef->keyBits.sym);
595                     if(iv.t.size == 0)
596                         return TPM_RC_FAILURE;
597                     if(nonceCaller->t.size >= iv.t.size)
598                     {
599                         MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
600                     }
601                     else
602                     {
603                         if(nonceCaller->t.size > sizeof(iv.t.buffer))
604                             return TPM_RC_FAILURE;
605                         MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
606                                    nonceCaller->t.size);
607                     }
608                     // make sure secret will fit
609                     if(secret->t.size > data->t.size)
610                         return TPM_RC_FAILURE;
611                     data->t.size = secret->t.size;
612                     // CFB decrypt, using nonceCaller as iv
613                     CryptSymmetricDecrypt(data->t.buffer, symDef->algorithm,
614                                           symDef->keyBits.sym,
615                                           decryptKey->sensitive.sensitive.sym.t.buffer,
616                                           &iv, ALG_CFB_VALUE, secret->t.size,
617                                           secret->t.secret);
618                 }
619             }
620             break;
621             default:
622                 FAIL(FATAL_ERROR_INTERNAL);
623                 break;
624         }
625         return result;
626     }
```

10.2.6.6.5 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```
627     void
628     CryptParameterEncryption(
629         TPM_HANDLE       handle,            // IN: encrypt session handle
630         TPM2B           *nonceCaller,       // IN: nonce caller
631         UINT16           leadingSizeInByte, // IN: the size of the leading size field in
632                                             //     bytes
633         TPM2B_AUTH      *extraKey,          // IN: additional key material other than
634                                             //     sessionAuth
635         BYTE            *buffer             // IN/OUT: parameter buffer to be encrypted
636         )
637     {
638         SESSION     *session = SessionGet(handle);  // encrypt session
639         TPM2B_TYPE(TEMP_KEY, (sizeof(extraKey->t.buffer)
640                               + sizeof(session->sessionKey.t.buffer)));
641         TPM2B_TEMP_KEY        key;                // encryption key
642         UINT32                cipherSize = 0;     // size of cipher text
643     //
644         // Retrieve encrypted data size.
645         if(leadingSizeInByte == 2)
646         {
647             // Extract the first two bytes as the size field as the data size
```

```
648         // encrypt
649         cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
650         // advance the buffer
651         buffer = &buffer[2];
652     }
653 #ifdef      TPM4B
654     else if(leadingSizeInByte == 4)
655     {
656         // use the first four bytes to indicate the number of bytes to encrypt
657         cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
658         //advance pointer
659         buffer = &buffer[4];
660     }
661 #endif
662     else
663     {
664         FAIL(FATAL_ERROR_INTERNAL);
665     }
666
667     // Compute encryption key by concatenating sessionKey with extra key
668     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
669     MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
670
671     if(session->symmetric.algorithm == ALG_XOR_VALUE)
672
673         // XOR parameter encryption formulation:
674         //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
675         CryptXORObfuscation(session->authHashAlg, &(key.b),
676                             &(session->nonceTPM.b),
677                             nonceCaller, cipherSize, buffer);
678     else
679         ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
680                        session->symmetric.keyBits.aes, &(key.b),
681                        nonceCaller, &(session->nonceTPM.b),
682                        cipherSize, buffer);
683     return;
684 }
```

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.

```
685 TPM_RC
686 CryptParameterDecryption(
687     TPM_HANDLE       handle,            // IN: encrypted session handle
688     TPM2B           *nonceCaller,       // IN: nonce caller
689     UINT32           bufferSize,        // IN: size of parameter buffer
690     UINT16           leadingSizeInByte, // IN: the size of the leading size field in
691                                         //     byte
692     TPM2B_AUTH      *extraKey,          // IN: the authValue
693     BYTE            *buffer             // IN/OUT: parameter buffer to be decrypted
694     )
695 {
696     SESSION         *session = SessionGet(handle);  // encrypt session
697     // The HMAC key is going to be the concatenation of the session key and any
698     // additional key material (like the authValue). The size of both of these
699     // is the size of the buffer which can contain a TPMT_HA.
700     TPM2B_TYPE(HMAC_KEY, (sizeof(extraKey->t.buffer)
701                           + sizeof(session->sessionKey.t.buffer)));
```



```
    TPM2B_HMAC_KEY          key;            // decryption key
    UINT32                  cipherSize = 0; // size of cipher text
//
    // Retrieve encrypted data size.
    if(leadingSizeInByte == 2)
    {
        // The first two bytes of the buffer are the size of the
        // data to be decrypted
        cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
        buffer = &buffer[2];   // advance the buffer
    }
#ifdef  TPM4B
    else if(leadingSizeInByte == 4)
    {
        // the leading size is four bytes so get the four byte size field
        cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
        buffer = &buffer[4];   //advance pointer
    }
#endif
    else
    {
        FAIL(FATAL_ERROR_INTERNAL);
    }
    if(cipherSize > bufferSize)
        return TPM_RC_SIZE;

    // Compute decryption key by concatenating sessionAuth with extra input key
    MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
    MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));

    if(session->symmetric.algorithm == ALG_XOR_VALUE)
        // XOR parameter decryption formulation:
        //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
        // Call XOR obfuscation function
        CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
                            &(session->nonceTPM.b), cipherSize, buffer);
    else
        // Assume that it is one of the symmetric block ciphers.
        ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
                       session->symmetric.keyBits.sym,
                       &key.b, nonceCaller, &session->nonceTPM.b,
                       cipherSize, buffer);

    return TPM_RC_SUCCESS;
}
```

10.2.6.6.7 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```
void
CryptComputeSymmetricUnique(
    TPMT_PUBLIC     *publicArea,    // IN: the object's public area
    TPMT_SENSITIVE  *sensitive,     // IN: the associated sensitive area
    TPM2B_DIGEST    *unique         // OUT: unique buffer
    )
{
    // For parents (symmetric and derivation), use an HMAC to compute
    // the 'unique' field
    if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
       && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt))
    {
        // Unique field is HMAC(sensitive->seedValue, sensitive->sensitive)
        HMAC_STATE      hmacState;
```

```
761         unique->b.size = CryptHmacStart2B(&hmacState, publicArea->nameAlg,
762                                           &sensitive->seedValue.b);
763         CryptDigestUpdate2B(&hmacState.hashState,
764                             &sensitive->sensitive.any.b);
765         CryptHmacEnd2B(&hmacState, &unique->b);
766     }
767     else
768     {
769         HASH_STATE  hashState;
770         // Unique := Hash(sensitive->seedValue || sensitive->sensitive)
771         unique->t.size = CryptHashStart(&hashState, publicArea->nameAlg);
772         CryptDigestUpdate2B(&hashState, &sensitive->seedValue.b);
773         CryptDigestUpdate2B(&hashState, &sensitive->sensitive.any.b);
774         CryptHashEnd2B(&hashState, &unique->b);
775     }
776     return;
777 }
```

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_CreateLoaded() 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_NO_RESULT	unable to get random values (only in derivation)
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

```
778 TPM_RC
779 CryptCreateObject(
780     OBJECT                  *object,            // IN: new object structure pointer
781     TPMS_SENSITIVE_CREATE   *sensitiveCreate,   // IN: sensitive creation
782     RAND_STATE              *rand               // IN: the random number generator
783                                                 //      to use
784     )
```

```
785 {
786     TPMT_PUBLIC             *publicArea = &object->publicArea;
787     TPM_RC                   result = TPM_RC_SUCCESS;
788 //
789     // Set the sensitive type for the object
790     object->sensitive.sensitiveType = publicArea->type;
791
792     // For all objects, copy the initial authorization data
793     object->sensitive.authValue = sensitiveCreate->userAuth;
794
795     // If the TPM is the source of the data, set the size of the provided data to
796     // zero so that there's no confusion about what to do.
797     if(IS_ATTRIBUTE(object->publicArea.objectAttributes,
798                     TPMA_OBJECT, sensitiveDataOrigin))
799         sensitiveCreate->data.t.size = 0;
800
801     // Generate the key and unique fields for the asymmetric keys and just the
802     // sensitive value for symmetric object
803     switch(publicArea->type)
804     {
805 #if     ALG_RSA
806         // Create RSA key
807         case ALG_RSA_VALUE:
808             // RSA uses full object so that it has a place to put the private
809             // exponent
810             result = CryptRsaGenerateKey(object, rand);
811             break;
812 #endif // ALG_RSA
813
814 #if     ALG_ECC
815         // Create ECC key
816         case ALG_ECC_VALUE:
817             result = CryptEccGenerateKey(&object->publicArea, &object->sensitive,
818                                          rand);
819             break;
820 #endif // ALG_ECC
821         case ALG_SYMCIPHER_VALUE:
822             result = CryptGenerateKeySymmetric(&object->publicArea,
823                                                &object->sensitive,
824                                                sensitiveCreate, rand);
825             break;
826         case ALG_KEYEDHASH_VALUE:
827             result = CryptGenerateKeyedHash(&object->publicArea, &object->sensitive,
828                                             sensitiveCreate, rand);
829             break;
830         default:
831             FAIL(FATAL_ERROR_INTERNAL);
832             break;
833     }
834     if(result != TPM_RC_SUCCESS)
835         return result;
836 // Create the sensitive seed value
837     // If this is a primary key in the endorsement hierarchy, stir the DRBG state
838     // This implementation uses both shProof and ehProof to make sure that there
839     // is no leakage of either.
840     if(object->attributes.primary && object->attributes.epsHierarchy)
841     {
842         DRBG_AdditionalData((DRBG_STATE *)rand, &gp.shProof.b);
843         DRBG_AdditionalData((DRBG_STATE *)rand, &gp.ehProof.b);
844     }
845     // Generate a seedValue that is the size of the digest produced by nameAlg
846     object->sensitive.seedValue.t.size =
847         DRBG_Generate(rand, object->sensitive.seedValue.t.buffer,
848                       CryptHashGetDigestSize(publicArea->nameAlg));
849     if(object->sensitive.seedValue.t.size == 0)
850         return TPM_RC_NO_RESULT;
```



```
851         // For symmetric objects, need to compute the unique value for the public area
852         if(publicArea->type == ALG_SYMCIPHER_VALUE
853            || publicArea->type == ALG_KEYEDHASH_VALUE)
854         {
855             CryptComputeSymmetricUnique(&object->publicArea, &object->sensitive,
856                                         &object->publicArea.unique.sym);
857         }
858         else
859         {
860             // if this is an asymmetric key and it isn't a parent, then
861             // get rid of the seed.
862             if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
863                || !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
864                 memset(&object->sensitive.seedValue, 0,
865                        sizeof(object->sensitive.seedValue));
866         }
867         // Compute the name
868         PublicMarshalAndComputeName(&object->publicArea, &object->name);
869         return result;
870     }
```

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

```
871     TPMI_ALG_HASH
872     CryptGetSignHashAlg(
873         TPMT_SIGNATURE  *auth           // IN: signature
874         )
875     {
876         if(auth->sigAlg == ALG_NULL_VALUE)
877             FAIL(FATAL_ERROR_INTERNAL);
878
879         // Get authHash algorithm based on signing scheme
880         switch(auth->sigAlg)
881         {
882     #if     ALG_RSA
883         // If RSA is supported, both RSASSA and RSAPSS are required
884     #   if !defined ALG_RSASSA_VALUE || !defined ALG_RSAPSS_VALUE
885     #       error "RSASSA and RSAPSS are required for RSA"
886     #   endif
887             case ALG_RSASSA_VALUE:
888                 return auth->signature.rsassa.hash;
889             case ALG_RSAPSS_VALUE:
890                 return auth->signature.rsapss.hash;
891     #endif // ALG_RSA
892
893     #if     ALG_ECC
894         // If ECC is defined, ECDSA is mandatory
895     #   if !ALG_ECDSA
896     #       error "ECDSA is requried for ECC"
897     #   endif
898             case ALG_ECDSA_VALUE:
899         // SM2 and ECSCHNORR are optional
900
901     #   if      ALG_SM2
902             case ALG_SM2_VALUE:
903     #   endif
904     #   if  ALG_ECSCHNORR
905             case ALG_ECSCHNORR_VALUE:
906     #   endif
907             //all ECC signatures look the same
908                 return auth->signature.ecdsa.hash;
```

```
#   if  ALG_ECDAA
        // Don't know how to verify an ECDAA signature
        case ALG_ECDAA_VALUE:
            break;
#   endif

#endif // ALG_ECC

        case ALG_HMAC_VALUE:
            return auth->signature.hmac.hashAlg;

        default:
            break;
    }
    return ALG_NULL_VALUE;
}
```

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)
    {
#   if  ALG_ECDAA
        case ALG_ECDAA_VALUE:
            return TRUE;
            break;
#   endif   // 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)
    {
#if     ALG_RSA
        case ALG_RSA_VALUE:
            switch(scheme)
            {
#   if !ALG_RSASSA  || !ALG_RSAPSS
#       error "RSASSA and PSAPSS required if RSA used."
#   endif
                case ALG_RSASSA_VALUE:
```



```
961                     case ALG_RSAPSS_VALUE:
962                         break;
963                     default:
964                         isSignScheme = FALSE;
965                         break;
966                 }
967                 break;
968     #endif // ALG_RSA
969
970     #if     ALG_ECC
971             // If ECC is implemented ECDSA is required
972             case ALG_ECC_VALUE:
973                 switch(scheme)
974                 {
975                     // Support for ECDSA is required for ECC
976                     case ALG_ECDSA_VALUE:
977     #if     ALG_ECDAA // ECDAA is optional
978                     case ALG_ECDAA_VALUE:
979     #endif
980     #if     ALG_ECSCHNORR // Schnorr is also optional
981                     case ALG_ECSCHNORR_VALUE:
982     #endif
983     #if     ALG_SM2 // SM2 is optional
984                     case ALG_SM2_VALUE:
985     #endif
986                         break;
987                     default:
988                         isSignScheme = FALSE;
989                         break;
990                 }
991                 break;
992     #endif // ALG_ECC
993             default:
994                 isSignScheme = FALSE;
995                 break;
996         }
997         return isSignScheme;
998     }
```

10.2.6.6.12 CryptIsAsymDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
999     BOOL
1000    CryptIsAsymDecryptScheme(
1001        TPMI_ALG_PUBLIC          publicType,         // IN: Type of the object
1002        TPMI_ALG_ASYM_SCHEME     scheme              // IN: the scheme
1003        )
1004    {
1005        BOOL        isDecryptScheme = TRUE;
1006
1007        switch(publicType)
1008        {
1009    #if     ALG_RSA
1010            case ALG_RSA_VALUE:
1011                switch(scheme)
1012                {
1013                    case ALG_RSAES_VALUE:
1014                    case ALG_OAEP_VALUE:
1015                        break;
1016                    default:
1017                        isDecryptScheme = FALSE;
1018                        break;
1019                }
```

```
1020                break;
1021    #endif // ALG_RSA
1022
1023    #if     ALG_ECC
1024            // If ECC is implemented ECDH is required
1025            case ALG_ECC_VALUE:
1026                switch(scheme)
1027                {
1028    #if  !ALG_ECDH
1029    #   error "ECDH is required for ECC"
1030    #endif
1031                    case ALG_ECDH_VALUE:
1032    #if     ALG_SM2
1033                    case ALG_SM2_VALUE:
1034    #endif
1035    #if     ALG_ECMQV
1036                    case ALG_ECMQV_VALUE:
1037    #endif
1038                        break;
1039                    default:
1040                        isDecryptScheme = FALSE;
1041                        break;
1042                }
1043                break;
1044    #endif // ALG_ECC
1045            default:
1046                isDecryptScheme = FALSE;
1047                break;
1048        }
1049        return isDecryptScheme;
1050    }
```

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' == ALG_SYMCIPHER.

Return Value	Meaning
TRUE(1)	scheme selected
FALSE(0)	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*

```
1051    BOOL
1052    CryptSelectSignScheme(
1053        OBJECT              *signObject,    // IN: signing key
1054        TPMT_SIG_SCHEME     *scheme         // IN/OUT: signing scheme
1055        )
1056    {
1057        TPMT_SIG_SCHEME     *objectScheme;
1058        TPMT_PUBLIC         *publicArea;
1059        BOOL                 OK;
1060
1061        // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
1062        // of the setting of scheme
```

```
1063        if(signObject == NULL)
1064        {
1065            OK = TRUE;
1066            scheme->scheme = ALG_NULL_VALUE;
1067            scheme->details.any.hashAlg = ALG_NULL_VALUE;
1068        }
1069        else
1070        {
1071            // assignment to save typing.
1072            publicArea = &signObject->publicArea;
1073
1074            // A symmetric cipher can be used to encrypt and decrypt but it can't
1075            // be used for signing
1076            if(publicArea->type == ALG_SYMCIPHER_VALUE)
1077                return FALSE;
1078            // Point to the scheme object
1079            if(CryptIsAsymAlgorithm(publicArea->type))
1080                objectScheme =
1081                (TPMT_SIG_SCHEME *)&publicArea->parameters.asymDetail.scheme;
1082            else
1083                objectScheme =
1084                (TPMT_SIG_SCHEME *)&publicArea->parameters.keyedHashDetail.scheme;
1085
1086        // If the object doesn't have a default scheme, then use the
1087        // input scheme.
1088            if(objectScheme->scheme == ALG_NULL_VALUE)
1089            {
1090                // Input and default can't both be NULL
1091                OK = (scheme->scheme != ALG_NULL_VALUE);
1092                // Assume that the scheme is compatible with the key. If not,
1093                // an error will be generated in the signing operation.
1094            }
1095            else if(scheme->scheme == ALG_NULL_VALUE)
1096            {
1097                // input scheme is NULL so use default
1098
1099                // First, check to see if the default requires that the caller
1100                // provided scheme data
1101                OK = !CryptIsSplitSign(objectScheme->scheme);
1102                if(OK)
1103                {
1104                    // The object has a scheme and the input is TPM_ALG_NULL so copy
1105                    // the object scheme as the final scheme. It is better to use a
1106                    // structure copy than a copy of the individual fields.
1107                    *scheme = *objectScheme;
1108                }
1109            }
1110            else
1111            {
1112                // Both input and object have scheme selectors
1113                // If the scheme and the hash are not the same then...
1114                // NOTE: the reason that there is no copy here is that the input
1115                // might contain extra data for a split signing scheme and that
1116                // data is not in the object so, it has to be preserved.
1117                OK = (objectScheme->scheme == scheme->scheme)
1118                    && (objectScheme->details.any.hashAlg
1119                        == scheme->details.any.hashAlg);
1120            }
1121        }
1122        return OK;
1123    }
```

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)

```
1124  TPM_RC
1125  CryptSign(
1126      OBJECT              *signKey,       // IN: signing key
1127      TPMT_SIG_SCHEME     *signScheme,    // IN: sign scheme.
1128      TPM2B_DIGEST        *digest,        // IN: The digest being signed
1129      TPMT_SIGNATURE      *signature      // OUT: signature
1130      )
1131  {
1132      TPM_RC               result = TPM_RC_SCHEME;
1133
1134      // Initialize signature scheme
1135      signature->sigAlg = signScheme->scheme;
1136
1137      // If the signature algorithm is TPM_ALG_NULL or the signing key is NULL,
1138      // then we are done
1139      if((signature->sigAlg == ALG_NULL_VALUE) || (signKey == NULL))
1140          return TPM_RC_SUCCESS;
1141
1142      // Initialize signature hash
1143      // Note: need to do the check for TPM_ALG_NULL first because the null scheme
1144      // doesn't have a hashAlg member.
1145      signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
1146
1147      // perform sign operation based on different key type
1148      switch(signKey->publicArea.type)
1149      {
1150  #if     ALG_RSA
1151          case ALG_RSA_VALUE:
1152              result = CryptRsaSign(signature, signKey, digest, NULL);
1153              break;
1154  #endif // ALG_RSA
1155  #if     ALG_ECC
1156          case ALG_ECC_VALUE:
1157              // The reason that signScheme is passed to CryptEccSign but not to the
1158              // other signing methods is that the signing for ECC may be split and
1159              // need the 'r' value that is in the scheme but not in the signature.
1160              result = CryptEccSign(signature, signKey, digest,
1161                                    (TPMT_ECC_SCHEME *)signScheme, NULL);
1162              break;
1163  #endif // ALG_ECC
1164          case ALG_KEYEDHASH_VALUE:
1165              result = CryptHmacSign(signature, signKey, digest);
1166              break;
1167          default:
1168              FAIL(FATAL_ERROR_INTERNAL);
1169              break;
1170      }
1171      return result;
1172  }
```


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

```
1173  TPM_RC
1174  CryptValidateSignature(
1175      TPMI_DH_OBJECT   keyHandle,     // IN: The handle of sign key
1176      TPM2B_DIGEST    *digest,        // IN: The digest being validated
1177      TPMT_SIGNATURE  *signature      // IN: signature
1178      )
1179  {
1180      // NOTE: HandleToObject will either return a pointer to a loaded object or
1181      // will assert. It will never return a non-valid value. This makes it save
1182      // to initialize 'publicArea' with the return value from HandleToObject()
1183      // without checking it first.
1184      OBJECT              *signObject = HandleToObject(keyHandle);
1185      TPMT_PUBLIC         *publicArea = &signObject->publicArea;
1186      TPM_RC               result = TPM_RC_SCHEME;
1187
1188      // The input unmarshaling should prevent any input signature from being
1189      // a NULL signature, but just in case
1190      if(signature->sigAlg == ALG_NULL_VALUE)
1191          return TPM_RC_SIGNATURE;
1192
1193      switch(publicArea->type)
1194      {
1195  #if     ALG_RSA
1196          case ALG_RSA_VALUE:
1197          {
1198      //
1199              // Call RSA code to verify signature
1200              result = CryptRsaValidateSignature(signature, signObject, digest);
1201              break;
1202          }
1203  #endif // ALG_RSA
1204
1205  #if     ALG_ECC
1206          case ALG_ECC_VALUE:
1207              result = CryptEccValidateSignature(signature, signObject, digest);
1208              break;
1209  #endif // ALG_ECC
1210
1211          case ALG_KEYEDHASH_VALUE:
1212              if(signObject->attributes.publicOnly)
1213                  result = TPM_RCS_HANDLE;
1214              else
1215                  result = CryptHMACVerifySignature(signObject, digest, signature);
1216              break;
1217          default:
1218              break;
1219      }
```

```
    return result;
}
```

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().

```
TPM_RC
CryptGetTestResult(
    TPM2B_MAX_BUFFER    *outData        // OUT: test result data
    )
{
    outData->t.size = 0;
    return TPM_RC_SUCCESS;
}
```

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(1)	sizes are consistent
FALSE(0)	sizes are not consistent

```
BOOL
CryptIsUniqueSizeValid(
    TPMT_PUBLIC     *publicArea     // IN: the public area to check
    )
{
    BOOL            consistent = FALSE;
    UINT16          keySizeInBytes;

    switch(publicArea->type)
    {
#if     ALG_RSA
        case ALG_RSA_VALUE:
            keySizeInBytes = BITS_TO_BYTES(
                                        publicArea->parameters.rsaDetail.keyBits);
            consistent = publicArea->unique.rsa.t.size == keySizeInBytes;
            break;
#endif // ALG_RSA
#if     ALG_ECC
        case ALG_ECC_VALUE:
        {
            keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(
                publicArea->parameters.eccDetail.curveID));
            consistent = keySizeInBytes > 0
                && publicArea->unique.ecc.x.t.size <= keySizeInBytes
                && publicArea->unique.ecc.y.t.size <= keySizeInBytes;
        }
        break;
#endif // ALG_ECC
        default:
            // For SYMCIPHER and KEYDEDHASH objects, the unique field is the size
```



```
1260                // of the nameAlg digest.
1261                consistent = publicArea->unique.sym.t.size
1262                    == CryptHashGetDigestSize(publicArea->nameAlg);
1263                break;
1264        }
1265        return consistent;
1266    }
```

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.

```
1267    BOOL
1268    CryptIsSensitiveSizeValid(
1269        TPMT_PUBLIC             *publicArea,        // IN: the object's public part
1270        TPMT_SENSITIVE          *sensitiveArea      // IN: the object's sensitive part
1271        )
1272    {
1273        BOOL                     consistent;
1274        UINT16                   keySizeInBytes;
1275
1276        switch(publicArea->type)
1277        {
1278    #if     ALG_RSA
1279            case ALG_RSA_VALUE:
1280                // sensitive prime value has to be half the size of the public modulus
1281                keySizeInBytes = BITS_TO_BYTES(publicArea->parameters.rsaDetail.keyBits);
1282                consistent =
1283                    ((sensitiveArea->sensitive.rsa.t.size * 2) == keySizeInBytes);
1284                break;
1285    #endif
1286    #if     ALG_ECC
1287            case ALG_ECC_VALUE:
1288                keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(
1289                    publicArea->parameters.eccDetail.curveID));
1290                consistent = (keySizeInBytes > 0)
1291                    && (sensitiveArea->sensitive.ecc.t.size == keySizeInBytes);
1292                break;
1293
1294    #endif
1295            case ALG_SYMCIPHER_VALUE:
1296                keySizeInBytes =
1297                    BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
1298                consistent = keySizeInBytes == sensitiveArea->sensitive.sym.t.size;
1299                break;
1300            case ALG_KEYEDHASH_VALUE:
1301                keySizeInBytes = CryptHashGetBlockSize(publicArea->nameAlg);
1302                // if the block size is 0, then the algorithm is TPM_ALG_NULL and the
1303                // size of the private part is limited to 128. If the algorithm block
1304                // size is over 128 bytes, then the size is limited to 128 bytes for
1305                // interoperability reasons.
1306                if((keySizeInBytes == 0) || (keySizeInBytes > 128))
1307                    keySizeInBytes = 128;
1308                consistent = sensitiveArea->sensitive.bits.t.size <= keySizeInBytes;
1309                break;
1310            default:
1311                consistent = TRUE;
1312                break;
1313        }
1314        return consistent;
1315    }
```

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

```
1316  TPM_RC
1317  CryptValidateKeys(
1318      TPMT_PUBLIC       *publicArea,
1319      TPMT_SENSITIVE    *sensitive,
1320      TPM_RC             blamePublic,
1321      TPM_RC             blameSensitive
1322      )
1323  {
1324      TPM_RC                result;
1325      UINT16                keySizeInBytes;
1326      UINT16                digestSize = CryptHashGetDigestSize(publicArea->nameAlg);
1327      TPMU_PUBLIC_PARMS    *params = &publicArea->parameters;
1328      TPMU_PUBLIC_ID       *unique = &publicArea->unique;
1329
1330      if(sensitive != NULL)
1331      {
1332          // Make sure that the types of the public and sensitive are compatible
1333          if(publicArea->type != sensitive->sensitiveType)
1334              return TPM_RCS_TYPE + blameSensitive;
1335          // Make sure that the authValue is not bigger than allowed
1336          // If there is no name algorithm, then the size just needs to be less than
1337          // the maximum size of the buffer used for authorization. That size check
1338          // was made during unmarshaling of the sensitive area
1339          if((sensitive->authValue.t.size) > digestSize && (digestSize > 0))
1340              return TPM_RCS_SIZE + blameSensitive;
1341      }
1342      switch(publicArea->type)
1343      {
1344  #if     ALG_RSA
1345          case ALG_RSA_VALUE:
1346              keySizeInBytes = BITS_TO_BYTES(params->rsaDetail.keyBits);
1347
1348              // Regardless of whether there is a sensitive area, the public modulus
1349              // needs to have the correct size. Otherwise, it can't be used for
1350              // any public key operation nor can it be used to compute the private
1351              // exponent.
1352              // NOTE: This implementation only supports key sizes that are multiples
1353              // of 1024 bits which means that the MSb of the 0th byte will always be
1354              // SET in either a prime or the public modulus.
1355              if((unique->rsa.t.size != keySizeInBytes)
1356                  || (unique->rsa.t.buffer[0] < 0x80))
1357                  return TPM_RCS_KEY + blamePublic;
1358              if(params->rsaDetail.exponent != 0
```



```
1359                 && params->rsaDetail.exponent < 7)
1360                  return TPM_RCS_VALUE + blamePublic;
1361             if(sensitive != NULL)
1362             {
1363                 // If there is a sensitive area, it has to be the correct size
1364                 // including having the correct high order bit SET.
1365                 if(((sensitive->sensitive.rsa.t.size * 2) != keySizeInBytes)
1366                    || (sensitive->sensitive.rsa.t.buffer[0] < 0x80))
1367                     return TPM_RCS_KEY_SIZE + blameSensitive;
1368             }
1369             break;
1370 #endif
1371 #if     ALG_ECC
1372         case ALG_ECC_VALUE:
1373         {
1374             TPMI_ECC_CURVE      curveId;
1375             curveId = params->eccDetail.curveID;
1376             keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(curveId));
1377             if(sensitive == NULL)
1378             {
1379                 // Validate the public key size
1380                 if(unique->ecc.x.t.size != keySizeInBytes
1381                    || unique->ecc.y.t.size != keySizeInBytes)
1382                     return TPM_RCS_KEY + blamePublic;
1383                 if(publicArea->nameAlg != ALG_NULL_VALUE)
1384                 {
1385                     if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
1386                         return TPM_RCS_ECC_POINT + blamePublic;
1387                 }
1388             }
1389             else
1390             {
1391                 // If the nameAlg is TPM_ALG_NULL, then only verify that the
1392                 // private part of the key is OK.
1393                 if(!CryptEccIsValidPrivateKey(&sensitive->sensitive.ecc,
1394                                               curveId))
1395                     return TPM_RCS_KEY_SIZE;
1396                 if(publicArea->nameAlg != ALG_NULL_VALUE)
1397                 {
1398                 // Full key load, verify that the public point belongs to the
1399                 // private key.
1400                     TPMS_ECC_POINT          toCompare;
1401                     result = CryptEccPointMultiply(&toCompare, curveId, NULL,
1402                                                    &sensitive->sensitive.ecc,
1403                                                    NULL, NULL);
1404                     if(result != TPM_RC_SUCCESS)
1405                         return TPM_RCS_BINDING;
1406                     else
1407                     {
1408                     // Make sure that the private key generated the public key.
1409                     // The input values and the values produced by the point
1410                     // multiply may not be the same size so adjust the computed
1411                     // value to match the size of the input value by adding or
1412                     // removing zeros.
1413                         AdjustNumberB(&toCompare.x.b, unique->ecc.x.t.size);
1414                         AdjustNumberB(&toCompare.y.b, unique->ecc.y.t.size);
1415                         if(!MemoryEqual2B(&unique->ecc.x.b, &toCompare.x.b)
1416                            || !MemoryEqual2B(&unique->ecc.y.b, &toCompare.y.b))
1417                             return TPM_RCS_BINDING;
1418                     }
1419                 }
1420             }
1421             break;
1422         }
1423 #endif
1424         default:
```

```
1425                // Checks for SYMCIPHER and KEYEDHASH are largely the same
1426                // If public area has a nameAlg, then validate the public area size
1427                // and if there is also a sensitive area, validate the binding
1428
1429                // For consistency, if the object is public-only just make sure that
1430                // the unique field is consistent with the name algorithm
1431                if(sensitive == NULL)
1432                {
1433                    if(unique->sym.t.size != digestSize)
1434                        return TPM_RCS_KEY + blamePublic;
1435                }
1436                else
1437                {
1438                    // Make sure that the key size in the sensitive area is consistent.
1439                    if(publicArea->type == ALG_SYMCIPHER_VALUE)
1440                    {
1441                        result = CryptSymKeyValidate(&params->symDetail.sym,
1442                                                     &sensitive->sensitive.sym);
1443                        if(result != TPM_RC_SUCCESS)
1444                            return result + blameSensitive;
1445                    }
1446                    else
1447                    {
1448                        // For a keyed hash object, the key has to be less than the
1449                        // smaller of the block size of the hash used in the scheme or
1450                        // 128 bytes. The worst case value is limited by the
1451                        // unmarshaling code so the only thing left to be checked is
1452                        // that it does not exceed the block size of the hash.
1453                        // by the hash algorithm of the scheme.
1454                        TPMT_KEYEDHASH_SCHEME       *scheme;
1455                        UINT16                       maxSize;
1456                        scheme = &params->keyedHashDetail.scheme;
1457                        if(scheme->scheme == ALG_XOR_VALUE)
1458                        {
1459                            maxSize = CryptHashGetBlockSize(scheme->details.xor.hashAlg);
1460                        }
1461                        else if(scheme->scheme == ALG_HMAC_VALUE)
1462                        {
1463                            maxSize = CryptHashGetBlockSize(scheme->details.hmac.hashAlg);
1464                        }
1465                        else if(scheme->scheme == ALG_NULL_VALUE)
1466                        {
1467                            // Not signing or xor so must be a data block
1468                            maxSize = 128;
1469                        }
1470                        else
1471                            return TPM_RCS_SCHEME + blamePublic;
1472                        if(sensitive->sensitive.bits.t.size > maxSize)
1473                            return TPM_RCS_KEY_SIZE + blameSensitive;
1474                    }
1475                    // If there is a nameAlg, check the binding
1476                    if(publicArea->nameAlg != ALG_NULL_VALUE)
1477                    {
1478                        TPM2B_DIGEST            compare;
1479                        if(sensitive->seedValue.t.size != digestSize)
1480                            return TPM_RCS_KEY_SIZE + blameSensitive;
1481
1482                        CryptComputeSymmetricUnique(publicArea, sensitive, &compare);
1483                        if(!MemoryEqual2B(&unique->sym.b, &compare.b))
1484                            return TPM_RC_BINDING;
1485                    }
1486                }
1487                break;
1488        }
1489        // For a parent, need to check that the seedValue is the correct size for
1490        // protections. It should be at least half the size of the nameAlg
```



```
1491        if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
1492           && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
1493           && sensitive != NULL
1494           && publicArea->nameAlg != ALG_NULL_VALUE)
1495        {
1496            if((sensitive->seedValue.t.size < (digestSize / 2))
1497               || (sensitive->seedValue.t.size > digestSize))
1498                return TPM_RCS_SIZE + blameSensitive;
1499        }
1500        return TPM_RC_SUCCESS;
1501    }
```

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

```
1502    void
1503    CryptAlgsSetImplemented(
1504        void
1505        )
1506    {
1507        AlgorithmGetImplementedVector(&g_implementedAlgorithms);
1508    }
```

10.2.6.6.21 CryptSelectMac()

This function is used to set the MAC scheme based on the key parameters and the input scheme.

Error Returns	Meaning
TPM_RC_SCHEME	the scheme is not a valid mac scheme
TPM_RC_TYPE	the input key is not a type that supports a mac
TPM_RC_VALUE	the input scheme and the key scheme are not compatible

```
1509    TPM_RC
1510    CryptSelectMac(
1511        TPMT_PUBLIC             *publicArea,
1512        TPMI_ALG_MAC_SCHEME     *inMac
1513    )
1514    {
1515        TPM_ALG_ID              macAlg = ALG_NULL_VALUE;
1516        switch(publicArea->type)
1517        {
1518            case ALG_KEYEDHASH_VALUE:
1519            {
1520                // Local value to keep lines from getting too long
1521                TPMT_KEYEDHASH_SCHEME   *scheme;
1522                scheme = &publicArea->parameters.keyedHashDetail.scheme;
1523                // Expect that the scheme is either HMAC or NULL
1524                if(scheme->scheme != ALG_NULL_VALUE)
1525                    macAlg = scheme->details.hmac.hashAlg;
1526                break;
1527            }
1528            case ALG_SYMCIPHER_VALUE:
1529            {
1530                TPMT_SYM_DEF_OBJECT     *scheme;
1531                scheme = &publicArea->parameters.symDetail.sym;
1532                // Expect that the scheme is either valid symmetric cipher or NULL
1533                if(scheme->algorithm != ALG_NULL_VALUE)
```

```
1534                    macAlg = scheme->mode.sym;
1535                break;
1536            }
1537            default:
1538                return TPM_RCS_TYPE;
1539        }
1540        // If the input value is not TPM_ALG_NULL ...
1541        if(*inMac != ALG_NULL_VALUE)
1542        {
1543            // ... then either the scheme in the key must be TPM_ALG_NULL or the input
1544            // value must match
1545            if((macAlg != ALG_NULL_VALUE) && (*inMac != macAlg))
1546                return TPM_RCS_VALUE;
1547        }
1548        else
1549        {
1550            // Since the input value is TPM_ALG_NULL, then the key value can't be
1551            // ALG_NULL
1552            if(macAlg == ALG_NULL_VALUE)
1553                return TPM_RCS_VALUE;
1554            *inMac = macAlg;
1555        }
1556        if(!CryptMacIsValidForKey(publicArea->type, *inMac, FALSE))
1557            return TPM_RCS_SCHEME;
1558        return TPM_RC_SUCCESS;
1559    }
```

10.2.6.6.22 CryptMacIsValidForKey()

Check to see if the key type is compatible with the mac type

```
1560    BOOL
1561    CryptMacIsValidForKey(
1562        TPM_ALG_ID          keyType,
1563        TPM_ALG_ID          macAlg,
1564        BOOL                flag
1565    )
1566    {
1567        switch(keyType)
1568        {
1569            case ALG_KEYEDHASH_VALUE:
1570                return CryptHashIsValidAlg(macAlg, flag);
1571                break;
1572            case ALG_SYMCIPHER_VALUE:
1573                return CryptSmacIsValidAlg(macAlg, flag);
1574                break;
1575            default:
1576                break;
1577        }
1578        return FALSE;
1579    }
```

10.2.6.6.23 CryptSmacIsValidAlg()

This function is used to test if an algorithm is a supported SMAC algorithm. It needs to be updated as new algorithms are added.

```
1580    BOOL
1581    CryptSmacIsValidAlg(
1582        TPM_ALG_ID      alg,
1583        BOOL            FLAG        // IN: Indicates if TPM_ALG_NULL is valid
1584    )
1585    {
```



```
1586        switch (alg)
1587        {
1588    #if     ALG_CMAC
1589            case ALG_CMAC_VALUE:
1590                return TRUE;
1591                break;
1592    #endif
1593            case ALG_NULL_VALUE:
1594                return FLAG;
1595                break;
1596            default:
1597                return FALSE;
1598        }
1599    }
```

10.2.6.6.24 CryptSymModeIsValid()

Function checks to see if an algorithm ID is a valid, symmetric block cipher mode for the TPM. If *flag* is SET, them TPM_ALG_NULL is a valid mode. not include the modes used for SMAC

```
1600    BOOL
1601    CryptSymModeIsValid(
1602        TPM_ALG_ID          mode,
1603        BOOL                flag
1604    )
1605    {
1606        switch(mode)
1607        {
1608    #if         ALG_CTR
1609            case ALG_CTR_VALUE:
1610    #endif // ALG_CTR
1611    #if         ALG_OFB
1612            case ALG_OFB_VALUE:
1613    #endif // ALG_OFB
1614    #if         ALG_CBC
1615            case ALG_CBC_VALUE:
1616    #endif // ALG_CBC
1617    #if         ALG_CFB
1618            case ALG_CFB_VALUE:
1619    #endif // ALG_CFB
1620    #if         ALG_ECB
1621            case ALG_ECB_VALUE:
1622    #endif // ALG_ECB
1623                return TRUE;
1624            case ALG_NULL_VALUE:
1625                return flag;
1626                break;
1627            default:
1628                break;
1629        }
1630        return FALSE;
1631    }
```

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
9       // For each of the algorithms that are in the toTestVecor, need to run a
10      // test
11      for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
12      {
13          if(TEST_BIT(alg, *toTest))
14          {
15              TPM_RC          result = CryptTestAlgorithm(alg, toTest);
16              if(result != TPM_RC_SUCCESS)
17                  return result;
18          }
19      }
20      return TPM_RC_SUCCESS;
21  }
```

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

```
LIB_EXPORT
TPM_RC
CryptSelfTest(
    TPMI_YES_NO      fullTest       // IN: if full test is required
    )
{
#if SIMULATION
    if(g_forceFailureMode)
        FAIL(FATAL_ERROR_FORCED);
#endif

    // If the caller requested a full test, then reset the to test vector so that
    // all the tests will be run
    if(fullTest == YES)
    {
        MemoryCopy(g_toTest,
                   g_implementedAlgorithms,
                   sizeof(g_toTest));
    }
    return CryptRunSelfTests(&g_toTest);
}
```

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

```
TPM_RC
CryptIncrementalSelfTest(
    TPML_ALG            *toTest,        // IN: list of algorithms to be tested
    TPML_ALG            *toDoList       // OUT: list of algorithms needing test
    )
{
    ALGORITHM_VECTOR     toTestVector = {0};
    TPM_ALG_ID           alg;
    UINT32               i;

    pAssert(toTest != NULL && toDoList != NULL);
    if(toTest->count > 0)
    {
        // Transcribe the toTest list into the toTestVector
        for(i = 0; i < toTest->count; i++)
        {
            alg = toTest->algorithms[i];

            // make sure that the algorithm value is not out of range
            if((alg > TPM_ALG_LAST) || !TEST_BIT(alg, g_implementedAlgorithms))
```

```
63                  return TPM_RC_VALUE;
64              SET_BIT(alg, toTestVector);
65          }
66          // Run the test
67          if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
68              return TPM_RC_CANCELED;
69      }
70      // Fill in the toDoList with the algorithms that are still untested
71      toDoList->count = 0;
72
73      for(alg = TPM_ALG_FIRST;
74      toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
75          alg++)
76      {
77          if(TEST_BIT(alg, g_toTest))
78              toDoList->algorithms[toDoList->count++] = alg;
79      }
80      return TPM_RC_SUCCESS;
81  }
```

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

```
82  void
83  CryptInitializeToTest(
84      void
85      )
86  {
87      // Indicate that nothing has been tested
88      memset(&g_cryptoSelfTestState, 0, sizeof(g_cryptoSelfTestState));
89
90      // Copy the implemented algorithm vector
91      MemoryCopy(g_toTest, g_implementedAlgorithms, sizeof(g_toTest));
92
93      // Setting the algorithm to null causes the test function to just clear
94      // out any algorithms for which there is no test.
95      CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);
96
97      return;
98  }
```

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 algorithm 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 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_CANCELED	test was canceled

```
99  LIB_EXPORT
100 TPM_RC
101 CryptTestAlgorithm(
102     TPM_ALG_ID            alg,
103     ALGORITHM_VECTOR     *toTest
```



```
104     )
105 {
106     TPM_RC                  result;
107 #if SELF_TEST
108     result = TestAlgorithm(alg, toTest);
109 #else
110     // If this is an attempt to determine the algorithms for which there is a
111     // self test, pretend that all of them do. We do that by not clearing any
112     // of the algorithm bits. When/if this function is called to run tests, it
113     // will over report. This can be changed so that any call to check on which
114     // algorithms have tests, 'toTest' can be cleared.
115     if(alg != TPM_ALG_ERROR)
116     {
117         CLEAR_BIT(alg, g_toTest);
118         if(toTest != NULL)
119             CLEAR_BIT(alg, *toTest);
120     }
121     result = TPM_RC_SUCCESS;
122 #endif
123     return result;
124 }
```

10.2.8 CryptEccData.c

```
1   #include    "Tpm.h"
```

This file contains the ECC curve data. The format of the data depends on the setting of USE_BN_ECC_DATA. If it is defined, then the TPM's BigNum() format is used. Otherwise, it is kept in TPM2B format. The purpose of having the data in BigNum() format is so that it does not have to be reformatted before being used by the crypto library.

```
2   #if ALG_ECC
3   #if USE_BN_ECC_DATA
4   #       define TO_ECC_64                       TO_CRYPT_WORD_64
5   #       define TO_ECC_56(a, b, c, d, e, f, g)  TO_ECC_64(0, a, b, c, d, e, f, g)
6   #       define TO_ECC_48(a, b, c, d, e, f)     TO_ECC_64(0, 0, a, b, c, d, e, f)
7   #       define TO_ECC_40(a, b, c, d, e)        TO_ECC_64(0, 0, 0, a, b, c, d, e)
8   #   if RADIX_BITS > 32
9   #       define TO_ECC_32(a, b, c, d)           TO_ECC_64(0, 0, 0, 0, a, b, c, d)
10  #       define TO_ECC_24(a, b, c)              TO_ECC_64(0, 0, 0, 0, 0, a, b, c)
11  #       define TO_ECC_16(a, b)                 TO_ECC_64(0, 0, 0, 0, 0, 0, a, b)
12  #       define TO_ECC_8(a)                     TO_ECC_64(0, 0, 0, 0, 0, 0, 0, a)
13  #   else // RADIX_BITS == 32
14  #       define TO_ECC_32                       BIG_ENDIAN_BYTES_TO_UINT32
15  #       define TO_ECC_24(a, b, c)              TO_ECC_32(0, a, b, c)
16  #       define TO_ECC_16(a, b)                 TO_ECC_32(0, 0, a, b)
17  #       define TO_ECC_8(a)                     TO_ECC_32(0, 0, 0, a)
18  #   endif
19  #else // TPM2B_
20  #   define TO_ECC_64(a, b, c, d, e, f, g, h) a, b, c, d, e, f, g, h
21  #   define TO_ECC_56(a, b, c, d, e, f, g)    a, b, c, d, e, f, g
22  #   define TO_ECC_48(a, b, c, d, e, f)       a, b, c, d, e, f
23  #   define TO_ECC_40(a, b, c, d, e)          a, b, c, d, e
24  #   define TO_ECC_32(a, b, c, d)             a, b, c, d
25  #   define TO_ECC_24(a, b, c)                a, b, c
26  #   define TO_ECC_16(a, b)                   a, b
27  #   define TO_ECC_8(a)                       a
28  #endif
29  #if USE_BN_ECC_DATA
30  #define BN_MIN_ALLOC(bytes)                                                 \
31          (BYTES_TO_CRYPT_WORDS(bytes) == 0) ? 1 : BYTES_TO_CRYPT_WORDS(bytes)
32  # define ECC_CONST(NAME, bytes, initializer)                                \
33      const struct {                                                          \
34          crypt_uword_t   allocate, size, d[BN_MIN_ALLOC(bytes)];             \
35          } NAME = {BN_MIN_ALLOC(bytes), BYTES_TO_CRYPT_WORDS(bytes),{initializer}}
```

define how to transform a curve parameter address into an entry into an ECC_CURVE_DATA structure.

```
36  # define ECC_ENTRY(val, x)    (bigNum)&val##_##x
37  ECC_CONST(ECC_ZERO, 0, 0);
38  #else
39  # define ECC_CONST(NAME, bytes, initializer)                                \
40         const TPM2B_##bytes##_BYTE_VALUE NAME = {bytes, {initializer}}
```

Have to special case ECC_ZERO

```
41  TPM2B_BYTE_VALUE(1);
42  TPM2B_1_BYTE_VALUE ECC_ZERO = {1, {0}};
43  # define ECC_ENTRY(val, x)    &val##_##x##.b
44  #endif
45  ECC_CONST(ECC_ONE, 1, 1);
46  #define CURVE_SPEC(C)                                                       \
47      const ECC_CURVE_DATA C = {                                              \
48              ECC_ENTRY(C, p), ECC_ENTRY(C, n),   ECC_ENTRY(C, h),  ECC_ENTRY(C, a), \
49              ECC_ENTRY(C, b), {ECC_ENTRY(C, gX), ECC_ENTRY(C, gY), ECC_ENTRY(C, gZ)}}
```



```
50   #if !USE_BN_ECC_DATA
51   TPM2B_BYTE_VALUE(24);
52   #define TO_ECC_192(a, b, c)  a, b, c
53   TPM2B_BYTE_VALUE(28);
54   #define TO_ECC_224(a, b, c, d)   a, b, c, d
55   TPM2B_BYTE_VALUE(32);
56   #define TO_ECC_256(a, b, c, d)   a, b, c, d
57   TPM2B_BYTE_VALUE(48);
58   #define TO_ECC_384(a, b, c, d, e, f)     a, b, c, d, e, f
59   TPM2B_BYTE_VALUE(66);
60   #define TO_ECC_528(a, b, c, d, e, f, g, h, i)    a, b, c, d, e, f, g, h, i
61   TPM2B_BYTE_VALUE(80);
62   #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j)     a, b, c, d, e, f, g, h, i, j
63   #else
64   #define TO_ECC_192(a, b, c)  c, b, a
65   #define TO_ECC_224(a, b, c, d)   d, c, b, a
66   #define TO_ECC_256(a, b, c, d)   d, c, b, a
67   #define TO_ECC_384(a, b, c, d, e, f)     f, e, d, c, b, a
68   #define TO_ECC_528(a, b, c, d, e, f, g, h, i)    i, h, g, f, e, d, c, b, a
69   #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j)     j, i, h, g, f, e, d, c, b, a
70   #endif // !USE_BN_ECC_DATA
71   #if ECC_NIST_P192
72   ECC_CONST(NIST_P192_p, 24, TO_ECC_192(
73       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
74       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
75       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
76   ECC_CONST(NIST_P192_a, 24, TO_ECC_192(
77       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
78       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
79       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
80   ECC_CONST(NIST_P192_b, 24, TO_ECC_192(
81       TO_ECC_64(0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7),
82       TO_ECC_64(0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49),
83       TO_ECC_64(0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1)));
84   ECC_CONST(NIST_P192_gX, 24, TO_ECC_192(
85       TO_ECC_64(0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6),
86       TO_ECC_64(0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00),
87       TO_ECC_64(0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12)));
88   ECC_CONST(NIST_P192_gY, 24, TO_ECC_192(
89       TO_ECC_64(0x07, 0x19, 0x2B, 0x95, 0xFF, 0xC8, 0xDA, 0x78),
90       TO_ECC_64(0x63, 0x10, 0x11, 0xED, 0x6B, 0x24, 0xCD, 0xD5),
91       TO_ECC_64(0x73, 0xF9, 0x77, 0xA1, 0x1E, 0x79, 0x48, 0x11)));
92   ECC_CONST(NIST_P192_n, 24, TO_ECC_192(
93       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
94       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x99, 0xDE, 0xF8, 0x36),
95       TO_ECC_64(0x14, 0x6B, 0xC9, 0xB1, 0xB4, 0xD2, 0x28, 0x31)));
96   #define NIST_P192_h         ECC_ONE
97   #define NIST_P192_gZ        ECC_ONE
98   CURVE_SPEC(NIST_P192);
99   #endif // ECC_NIST_P192
100  #if ECC_NIST_P224
101  ECC_CONST(NIST_P224_p, 28, TO_ECC_224(
102      TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
103      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
104      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
105      TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01)));
106  ECC_CONST(NIST_P224_a, 28, TO_ECC_224(
107      TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
108      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
109      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
110      TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE)));
111  ECC_CONST(NIST_P224_b, 28, TO_ECC_224(
112      TO_ECC_32(0xB4, 0x05, 0x0A, 0x85),
113      TO_ECC_64(0x0C, 0x04, 0xB3, 0xAB, 0xF5, 0x41, 0x32, 0x56),
114      TO_ECC_64(0x50, 0x44, 0xB0, 0xB7, 0xD7, 0xBF, 0xD8, 0xBA),
115      TO_ECC_64(0x27, 0x0B, 0x39, 0x43, 0x23, 0x55, 0xFF, 0xB4)));
```

```
116   ECC_CONST(NIST_P224_gX, 28, TO_ECC_224(
117       TO_ECC_32(0xB7, 0x0E, 0x0C, 0xBD),
118       TO_ECC_64(0x6B, 0xB4, 0xBF, 0x7F, 0x32, 0x13, 0x90, 0xB9),
119       TO_ECC_64(0x4A, 0x03, 0xC1, 0xD3, 0x56, 0xC2, 0x11, 0x22),
120       TO_ECC_64(0x34, 0x32, 0x80, 0xD6, 0x11, 0x5C, 0x1D, 0x21)));
121   ECC_CONST(NIST_P224_gY, 28, TO_ECC_224(
122       TO_ECC_32(0xBD, 0x37, 0x63, 0x88),
123       TO_ECC_64(0xB5, 0xF7, 0x23, 0xFB, 0x4C, 0x22, 0xDF, 0xE6),
124       TO_ECC_64(0xCD, 0x43, 0x75, 0xA0, 0x5A, 0x07, 0x47, 0x64),
125       TO_ECC_64(0x44, 0xD5, 0x81, 0x99, 0x85, 0x00, 0x7E, 0x34)));
126   ECC_CONST(NIST_P224_n, 28, TO_ECC_224(
127       TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
128       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
129       TO_ECC_64(0xFF, 0xFF, 0x16, 0xA2, 0xE0, 0xB8, 0xF0, 0x3E),
130       TO_ECC_64(0x13, 0xDD, 0x29, 0x45, 0x5C, 0x5C, 0x2A, 0x3D)));
131   #define NIST_P224_h         ECC_ONE
132   #define NIST_P224_gZ        ECC_ONE
133   CURVE_SPEC(NIST_P224);
134   #endif // ECC_NIST_P224
135   #if ECC_NIST_P256
136   ECC_CONST(NIST_P256_p, 32, TO_ECC_256(
137       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
138       TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
139       TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
140       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
141   ECC_CONST(NIST_P256_a, 32, TO_ECC_256(
142       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
143       TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
144       TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
145       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
146   ECC_CONST(NIST_P256_b, 32, TO_ECC_256(
147       TO_ECC_64(0x5A, 0xC6, 0x35, 0xD8, 0xAA, 0x3A, 0x93, 0xE7),
148       TO_ECC_64(0xB3, 0xEB, 0xBD, 0x55, 0x76, 0x98, 0x86, 0xBC),
149       TO_ECC_64(0x65, 0x1D, 0x06, 0xB0, 0xCC, 0x53, 0xB0, 0xF6),
150       TO_ECC_64(0x3B, 0xCE, 0x3C, 0x3E, 0x27, 0xD2, 0x60, 0x4B)));
151   ECC_CONST(NIST_P256_gX, 32, TO_ECC_256(
152       TO_ECC_64(0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47),
153       TO_ECC_64(0xF8, 0xBC, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2),
154       TO_ECC_64(0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0),
155       TO_ECC_64(0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96)));
156   ECC_CONST(NIST_P256_gY, 32, TO_ECC_256(
157       TO_ECC_64(0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B),
158       TO_ECC_64(0x8E, 0xE7, 0xEB, 0x4A, 0x7C, 0x0F, 0x9E, 0x16),
159       TO_ECC_64(0x2B, 0xCE, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE),
160       TO_ECC_64(0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5)));
161   ECC_CONST(NIST_P256_n, 32, TO_ECC_256(
162       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
163       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
164       TO_ECC_64(0xBC, 0xE6, 0xFA, 0xAD, 0xA7, 0x17, 0x9E, 0x84),
165       TO_ECC_64(0xF3, 0xB9, 0xCA, 0xC2, 0xFC, 0x63, 0x25, 0x51)));
166   #define NIST_P256_h         ECC_ONE
167   #define NIST_P256_gZ        ECC_ONE
168   CURVE_SPEC(NIST_P256);
169   #endif // ECC_NIST_P256
170   #if ECC_NIST_P384
171   ECC_CONST(NIST_P384_p, 48, TO_ECC_384(
172       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
173       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
174       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
175       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
176       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
177       TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF)));
178   ECC_CONST(NIST_P384_a, 48, TO_ECC_384(
179       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
180       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
181       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
```



```
182         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
183         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
184         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFC)));
185     ECC_CONST(NIST_P384_b, 48, TO_ECC_384(
186         TO_ECC_64(0xB3, 0x31, 0x2F, 0xA7, 0xE2, 0x3E, 0xE7, 0xE4),
187         TO_ECC_64(0x98, 0x8E, 0x05, 0x6B, 0xE3, 0xF8, 0x2D, 0x19),
188         TO_ECC_64(0x18, 0x1D, 0x9C, 0x6E, 0xFE, 0x81, 0x41, 0x12),
189         TO_ECC_64(0x03, 0x14, 0x08, 0x8F, 0x50, 0x13, 0x87, 0x5A),
190         TO_ECC_64(0xC6, 0x56, 0x39, 0x8D, 0x8A, 0x2E, 0xD1, 0x9D),
191         TO_ECC_64(0x2A, 0x85, 0xC8, 0xED, 0xD3, 0xEC, 0x2A, 0xEF)));
192     ECC_CONST(NIST_P384_gX, 48, TO_ECC_384(
193         TO_ECC_64(0xAA, 0x87, 0xCA, 0x22, 0xBE, 0x8B, 0x05, 0x37),
194         TO_ECC_64(0x8E, 0xB1, 0xC7, 0x1E, 0xF3, 0x20, 0xAD, 0x74),
195         TO_ECC_64(0x6E, 0x1D, 0x3B, 0x62, 0x8B, 0xA7, 0x9B, 0x98),
196         TO_ECC_64(0x59, 0xF7, 0x41, 0xE0, 0x82, 0x54, 0x2A, 0x38),
197         TO_ECC_64(0x55, 0x02, 0xF2, 0x5D, 0xBF, 0x55, 0x29, 0x6C),
198         TO_ECC_64(0x3A, 0x54, 0x5E, 0x38, 0x72, 0x76, 0x0A, 0xB7)));
199     ECC_CONST(NIST_P384_gY, 48, TO_ECC_384(
200         TO_ECC_64(0x36, 0x17, 0xDE, 0x4A, 0x96, 0x26, 0x2C, 0x6F),
201         TO_ECC_64(0x5D, 0x9E, 0x98, 0xBF, 0x92, 0x92, 0xDC, 0x29),
202         TO_ECC_64(0xF8, 0xF4, 0x1D, 0xBD, 0x28, 0x9A, 0x14, 0x7C),
203         TO_ECC_64(0xE9, 0xDA, 0x31, 0x13, 0xB5, 0xF0, 0xB8, 0xC0),
204         TO_ECC_64(0x0A, 0x60, 0xB1, 0xCE, 0x1D, 0x7E, 0x81, 0x9D),
205         TO_ECC_64(0x7A, 0x43, 0x1D, 0x7C, 0x90, 0xEA, 0x0E, 0x5F)));
206     ECC_CONST(NIST_P384_n, 48, TO_ECC_384(
207         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
208         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
209         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
210         TO_ECC_64(0xC7, 0x63, 0x4D, 0x81, 0xF4, 0x37, 0x2D, 0xDF),
211         TO_ECC_64(0x58, 0x1A, 0x0D, 0xB2, 0x48, 0xB0, 0xA7, 0x7A),
212         TO_ECC_64(0xEC, 0xEC, 0x19, 0x6A, 0xCC, 0xC5, 0x29, 0x73)));
213     #define NIST_P384_h         ECC_ONE
214     #define NIST_P384_gZ        ECC_ONE
215     CURVE_SPEC(NIST_P384);
216     #endif // ECC_NIST_P384
217     #if ECC_NIST_P521
218     ECC_CONST(NIST_P521_p, 66, TO_ECC_528(
219         TO_ECC_16(0x01, 0xFF),
220         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
221         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
222         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
223         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
224         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
225         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
226         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
227         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
228     ECC_CONST(NIST_P521_a, 66, TO_ECC_528(
229         TO_ECC_16(0x01, 0xFF),
230         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
231         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
232         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
233         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
234         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
235         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
236         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
237         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
238     ECC_CONST(NIST_P521_b, 66, TO_ECC_528(
239         TO_ECC_16(0x00, 0x51),
240         TO_ECC_64(0x95, 0x3E, 0xB9, 0x61, 0x8E, 0x1C, 0x9A, 0x1F),
241         TO_ECC_64(0x92, 0x9A, 0x21, 0xA0, 0xB6, 0x85, 0x40, 0xEE),
242         TO_ECC_64(0xA2, 0xDA, 0x72, 0x5B, 0x99, 0xB3, 0x15, 0xF3),
243         TO_ECC_64(0xB8, 0xB4, 0x89, 0x91, 0x8E, 0xF1, 0x09, 0xE1),
244         TO_ECC_64(0x56, 0x19, 0x39, 0x51, 0xEC, 0x7E, 0x93, 0x7B),
245         TO_ECC_64(0x16, 0x52, 0xC0, 0xBD, 0x3B, 0xB1, 0xBF, 0x07),
246         TO_ECC_64(0x35, 0x73, 0xDF, 0x88, 0x3D, 0x2C, 0x34, 0xF1),
247         TO_ECC_64(0xEF, 0x45, 0x1F, 0xD4, 0x6B, 0x50, 0x3F, 0x00)));
```

```
248   ECC_CONST(NIST_P521_gX, 66, TO_ECC_528(
249       TO_ECC_16(0x00, 0xC6),
250       TO_ECC_64(0x85, 0x8E, 0x06, 0xB7, 0x04, 0x04, 0xE9, 0xCD),
251       TO_ECC_64(0x9E, 0x3E, 0xCB, 0x66, 0x23, 0x95, 0xB4, 0x42),
252       TO_ECC_64(0x9C, 0x64, 0x81, 0x39, 0x05, 0x3F, 0xB5, 0x21),
253       TO_ECC_64(0xF8, 0x28, 0xAF, 0x60, 0x6B, 0x4D, 0x3D, 0xBA),
254       TO_ECC_64(0xA1, 0x4B, 0x5E, 0x77, 0xEF, 0xE7, 0x59, 0x28),
255       TO_ECC_64(0xFE, 0x1D, 0xC1, 0x27, 0xA2, 0xFF, 0xA8, 0xDE),
256       TO_ECC_64(0x33, 0x48, 0xB3, 0xC1, 0x85, 0x6A, 0x42, 0x9B),
257       TO_ECC_64(0xF9, 0x7E, 0x7E, 0x31, 0xC2, 0xE5, 0xBD, 0x66)));
258   ECC_CONST(NIST_P521_gY, 66, TO_ECC_528(
259       TO_ECC_16(0x01, 0x18),
260       TO_ECC_64(0x39, 0x29, 0x6A, 0x78, 0x9A, 0x3B, 0xC0, 0x04),
261       TO_ECC_64(0x5C, 0x8A, 0x5F, 0xB4, 0x2C, 0x7D, 0x1B, 0xD9),
262       TO_ECC_64(0x98, 0xF5, 0x44, 0x49, 0x57, 0x9B, 0x44, 0x68),
263       TO_ECC_64(0x17, 0xAF, 0xBD, 0x17, 0x27, 0x3E, 0x66, 0x2C),
264       TO_ECC_64(0x97, 0xEE, 0x72, 0x99, 0x5E, 0xF4, 0x26, 0x40),
265       TO_ECC_64(0xC5, 0x50, 0xB9, 0x01, 0x3F, 0xAD, 0x07, 0x61),
266       TO_ECC_64(0x35, 0x3C, 0x70, 0x86, 0xA2, 0x72, 0xC2, 0x40),
267       TO_ECC_64(0x88, 0xBE, 0x94, 0x76, 0x9F, 0xD1, 0x66, 0x50)));
268   ECC_CONST(NIST_P521_n, 66, TO_ECC_528(
269       TO_ECC_16(0x01, 0xFF),
270       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
271       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
272       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
273       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFA),
274       TO_ECC_64(0x51, 0x86, 0x87, 0x83, 0xBF, 0x2F, 0x96, 0x6B),
275       TO_ECC_64(0x7F, 0xCC, 0x01, 0x48, 0xF7, 0x09, 0xA5, 0xD0),
276       TO_ECC_64(0x3B, 0xB5, 0xC9, 0xB8, 0x89, 0x9C, 0x47, 0xAE),
277       TO_ECC_64(0xBB, 0x6F, 0xB7, 0x1E, 0x91, 0x38, 0x64, 0x09)));
278   #define NIST_P521_h         ECC_ONE
279   #define NIST_P521_gZ        ECC_ONE
280   CURVE_SPEC(NIST_P521);
281   #endif // ECC_NIST_P521
282   #if ECC_BN_P256
283   ECC_CONST(BN_P256_p, 32, TO_ECC_256(
284       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
285       TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9F),
286       TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x98, 0x0A, 0x82),
287       TO_ECC_64(0xD3, 0x29, 0x2D, 0xDB, 0xAE, 0xD3, 0x30, 0x13)));
288   #define BN_P256_a           ECC_ZERO
289   ECC_CONST(BN_P256_b, 1, TO_ECC_8(3));
290   #define BN_P256_gX          ECC_ONE
291   ECC_CONST(BN_P256_gY, 1, TO_ECC_8(2));
292   ECC_CONST(BN_P256_n, 32, TO_ECC_256(
293       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
294       TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9E),
295       TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x99, 0x92, 0x1A),
296       TO_ECC_64(0xF6, 0x2D, 0x53, 0x6C, 0xD1, 0x0B, 0x50, 0x0D)));
297   #define BN_P256_h           ECC_ONE
298   #define BN_P256_gZ          ECC_ONE
299   CURVE_SPEC(BN_P256);
300   #endif // ECC_BN_P256
301   #if ECC_BN_P638
302   ECC_CONST(BN_P638_p, 80, TO_ECC_640(
303       TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
304       TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
305       TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
306       TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
307       TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
308       TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
309       TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
310       TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
311       TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
312       TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67)));
313   #define BN_P638_a           ECC_ZERO
```



```
314    ECC_CONST(BN_P638_b, 2, TO_ECC_16(0x01,0x01));
315    ECC_CONST(BN_P638_gX, 80, TO_ECC_640(
316        TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
317        TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
318        TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
319        TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
320        TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
321        TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
322        TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
323        TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
324        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
325        TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66)));
326    ECC_CONST(BN_P638_gY, 1, TO_ECC_8(0x10));
327    ECC_CONST(BN_P638_n, 80, TO_ECC_640(
328        TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
329        TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
330        TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
331        TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
332        TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
333        TO_ECC_64(0x60, 0x00, 0x86, 0x55, 0x00, 0x21, 0xE5, 0x55),
334        TO_ECC_64(0xFF, 0xFF, 0xF5, 0x4F, 0xFF, 0xF4, 0xEA, 0xC0),
335        TO_ECC_64(0x00, 0x00, 0x00, 0x49, 0x80, 0x01, 0x54, 0xD9),
336        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xA0),
337        TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61)));
338    #define BN_P638_h           ECC_ONE
339    #define BN_P638_gZ          ECC_ONE
340    CURVE_SPEC(BN_P638);
341    #endif // ECC_BN_P638
342    #if ECC_SM2_P256
343    ECC_CONST(SM2_P256_p, 32, TO_ECC_256(
344        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
345        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
346        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
347        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
348    ECC_CONST(SM2_P256_a, 32, TO_ECC_256(
349        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
350        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
351        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
352        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
353    ECC_CONST(SM2_P256_b, 32, TO_ECC_256(
354        TO_ECC_64(0x28, 0xE9, 0xFA, 0x9E, 0x9D, 0x9F, 0x5E, 0x34),
355        TO_ECC_64(0x4D, 0x5A, 0x9E, 0x4B, 0xCF, 0x65, 0x09, 0xA7),
356        TO_ECC_64(0xF3, 0x97, 0x89, 0xF5, 0x15, 0xAB, 0x8F, 0x92),
357        TO_ECC_64(0xDD, 0xBC, 0xBD, 0x41, 0x4D, 0x94, 0x0E, 0x93)));
358    ECC_CONST(SM2_P256_gX, 32, TO_ECC_256(
359        TO_ECC_64(0x32, 0xC4, 0xAE, 0x2C, 0x1F, 0x19, 0x81, 0x19),
360        TO_ECC_64(0x5F, 0x99, 0x04, 0x46, 0x6A, 0x39, 0xC9, 0x94),
361        TO_ECC_64(0x8F, 0xE3, 0x0B, 0xBF, 0xF2, 0x66, 0x0B, 0xE1),
362        TO_ECC_64(0x71, 0x5A, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7)));
363    ECC_CONST(SM2_P256_gY, 32, TO_ECC_256(
364        TO_ECC_64(0xBC, 0x37, 0x36, 0xA2, 0xF4, 0xF6, 0x77, 0x9C),
365        TO_ECC_64(0x59, 0xBD, 0xCE, 0xE3, 0x6B, 0x69, 0x21, 0x53),
366        TO_ECC_64(0xD0, 0xA9, 0x87, 0x7C, 0xC6, 0x2A, 0x47, 0x40),
367        TO_ECC_64(0x02, 0xDF, 0x32, 0xE5, 0x21, 0x39, 0xF0, 0xA0)));
368    ECC_CONST(SM2_P256_n, 32, TO_ECC_256(
369        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
370        TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
371        TO_ECC_64(0x72, 0x03, 0xDF, 0x6B, 0x21, 0xC6, 0x05, 0x2B),
372        TO_ECC_64(0x53, 0xBB, 0xF4, 0x09, 0x39, 0xD5, 0x41, 0x23)));
373    #define SM2_P256_h          ECC_ONE
374    #define SM2_P256_gZ         ECC_ONE
375    CURVE_SPEC(SM2_P256);
376    #endif // ECC_SM2_P256
377    #define comma
378    const ECC_CURVE   eccCurves[] = {
379    #if ECC_NIST_P192
```

```
380         comma
381         {TPM_ECC_NIST_P192,
382         192,
383         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
384         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
385         &NIST_P192
386         CURVE_NAME("NIST_P192")}
387     #   undef comma
388     #   define comma ,
389     #endif // ECC_NIST_P192
390     #if ECC_NIST_P224
391         comma
392         {TPM_ECC_NIST_P224,
393         224,
394         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
395         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
396         &NIST_P224
397         CURVE_NAME("NIST_P224")}
398     #   undef comma
399     #   define comma ,
400     #endif // ECC_NIST_P224
401     #if ECC_NIST_P256
402         comma
403         {TPM_ECC_NIST_P256,
404         256,
405         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
406         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
407         &NIST_P256
408         CURVE_NAME("NIST_P256")}
409     #   undef comma
410     #   define comma ,
411     #endif // ECC_NIST_P256
412     #if ECC_NIST_P384
413         comma
414         {TPM_ECC_NIST_P384,
415         384,
416         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA384_VALUE}}},
417         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
418         &NIST_P384
419         CURVE_NAME("NIST_P384")}
420     #   undef comma
421     #   define comma ,
422     #endif // ECC_NIST_P384
423     #if ECC_NIST_P521
424         comma
425         {TPM_ECC_NIST_P521,
426         521,
427         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA512_VALUE}}},
428         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
429         &NIST_P521
430         CURVE_NAME("NIST_P521")}
431     #   undef comma
432     #   define comma ,
433     #endif // ECC_NIST_P521
434     #if ECC_BN_P256
435         comma
436         {TPM_ECC_BN_P256,
437         256,
438         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
439         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
440         &BN_P256
441         CURVE_NAME("BN_P256")}
442     #   undef comma
443     #   define comma ,
444     #endif // ECC_BN_P256
445     #if ECC_BN_P638
```



```
446         comma
447         {TPM_ECC_BN_P638,
448         638,
449         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
450         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
451         &BN_P638
452         CURVE_NAME("BN_P638")}
453     #   undef comma
454     #   define comma ,
455     #endif // ECC_BN_P638
456     #if ECC_SM2_P256
457         comma
458         {TPM_ECC_SM2_P256,
459         256,
460         {ALG_KDF1_SP800_56A_VALUE, {{ALG_SM3_256_VALUE}}},
461         {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
462         &SM2_P256
463         CURVE_NAME("SM2_P256")}
464     #   undef comma
465     #   define comma ,
466     #endif // ECC_SM2_P256
467     };
468     #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

```
1   #include "Tpm.h"
2   #if     ALG_TDES
3   #define DES_NUM_WEAK 64
4   const UINT64 DesWeakKeys[DES_NUM_WEAK] = {
5       0x0101010101010101, 0xFEFEFEFEFEFEFEFE, 0xE0E0E0E0F1F1F1F1, 0x1F1F1F1F0E0E0E0E,
6       0x011F011F010E010E, 0x1F011F010E010E01, 0x01E001E001F101F1, 0xE001E001F101F101,
7       0x01FE01FE01FE01FE, 0xFE01FE01FE01FE01, 0x1FE01FE00EF10EF1, 0xE01FE01FF10EF10E,
8       0x1FFE1FFE0EFE0EFE, 0xFE1FFE1FFE0EFE0E, 0xE0FEE0FEF1FEF1FE, 0xFEE0FEE0FEF1FEF1,
9       0x01011F1F01010E0E, 0x1F1F01010E0E0101, 0xE0E01F1FF1F10E0E, 0x0101E0E00101F1F1,
10      0x1F1FE0E00E0EF1F1, 0xE0E0FEFEF1F1FEFE, 0x0101FEFE0101FEFE, 0x1F1FFEFE0E0EFEFE,
11      0xE0FE011FF1FE010E, 0x011F1F01010E0E01, 0x1FE001FE0EF101FE, 0xE0FE1F01F1FE0E01,
12      0x011FE0FE010EF1FE, 0x1FE0E01F0EF1F10E, 0xE0FEFEE0F1FEFEF1, 0x011FFEE0010EFEF1,
13      0x1FE0FE010EF1FE01, 0xFE0101FEFE0101FE, 0x01E01FFE01F10EFE, 0x1FFE01E00EFE01F1,
14      0xFE011FE0FE010EF1, 0xFE01E01FFE01F10E, 0x1FFEE0010EFEF101, 0xFE1F01E0FE0E01F1,
15      0x01E0E00101F1F101, 0x1FFEFE1F0EFEFE0E, 0xFE1FE001FE0EF101, 0x01E0FE1F01F1FE0E,
16      0xE00101E0F10101F1, 0xFE1F1FFEFE0E0EFE, 0x01FE1FE001FE0EF1, 0xE0011FFEF1010EFE,
17      0xFEE0011FFEF1010E, 0x01FEE01F01FEF10E, 0xE001FE1FF101FE0E, 0xFEE01F01FEF10E01,
18      0x01FEFE0101FEFE01, 0xE01F01FEF10E01FE, 0xFEE0E0FEFEF1F1FE, 0x1F01011F0E01010E,
19      0xE01F1FE0F10E0EF1, 0xFEFE0101FEFE0101, 0x1F01E0FE0E01F1FE, 0xE01FFE01F10EFE01,
20      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.

```
21  UINT64
22  CryptSetOddByteParity(
23      UINT64          k
24      )
25  {
26  #define PMASK 0x0101010101010101ULL
27      UINT64          out;
28      k |= PMASK;     // set the parity bit
29      out = k;
30      k ^= k >> 4;
31      k ^= k >> 2;
32      k ^= k >> 1;
33      k &= PMASK;     // odd parity extracted
34      out ^= k;       // out is now even parity because parity bit was already set
35      out ^= PMASK;   // out is now even parity
36      return out;
37  }
```

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.

Return Value	Meaning
TRUE(1)	DES key is weak
FALSE(0)	DES key is not weak

```
38  static BOOL
39  CryptDesIsWeakKey(
40      UINT64              k
41      )
42  {
43      int                 i;
44  //
45      for(i = 0; i < DES_NUM_WEAK; i++)
46      {
47          if(k == DesWeakKeys[i])
48              return TRUE;
49      }
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
```

```
    )
{

    // Assume that the publicArea key size has been validated and is a supported
    // number of bits.
    sensitive->sensitive.sym.t.size =
            BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
    do
    {
        BYTE                    *pK = sensitive->sensitive.sym.t.buffer;
        int                      i = (sensitive->sensitive.sym.t.size + 7) / 8;
// Use the random number generator to generate the required number of bits
        if(DRBG_Generate(rand, pK, sensitive->sensitive.sym.t.size) == 0)
            return TPM_RC_NO_RESULT;
        for(; i > 0; pK += 8, i--)
        {
            UINT64      k = BYTE_ARRAY_TO_UINT64(pK);
            k = CryptSetOddByteParity(k);
            UINT64_TO_BYTE_ARRAY(k, pK);
        }
    } while(!CryptDesValidateKey(&sensitive->sensitive.sym));
    return TPM_RC_SUCCESS;
}
#endif
```

10.2.10 CryptEccKeyExchange.c

10.2.10.1 Introduction

This file contains the functions that are used for the two-phase, ECC, key-exchange protocols

```
#include "Tpm.h"
#if CC_ZGen_2Phase == YES
```

10.2.10.2 Functions

```
#if ALG_ECMQV
```

10.2.10.2.1 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) Always returns TRUE(1).

```
static BOOL
avf1(
    bigNum                  bnX,            // IN/OUT: the reduced value
    bigNum                  bnN             // IN: the order of the curve
    )
{
// compute f = 2^(ceil(ceil(log2(n)) / 2))
    int                         f = (BnSizeInBits(bnN) + 1) / 2;
// x' = 2^f + (x mod 2^f)
    BnMaskBits(bnX, f);    // This is mod 2*2^f but it doesn't matter because
                             // the next operation will SET the extra bit anyway
    BnSetBit(bnX, f);
    return TRUE;
}
```

10.2.10.2.2 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_NO_RESULT	the value for *dsA* does not give a valid point on the curve

```
static TPM_RC
C_2_2_MQV(
    TPMS_ECC_POINT          *outZ,          // OUT: the computed point
    TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
    TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
    TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
    TPMS_ECC_POINT          *QsB,           // IN: static public party B key
    TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
```

```
26      )
27  {
28      CURVE_INITIALIZED(E, curveId);
29      const ECC_CURVE_DATA    *C;
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      C = AccessCurveData(E);
46  // Process:
47  //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
48  //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
49  //  3. If P = O, output an error indicator.
50  //  4. Z=xP, where xP is the x-coordinate of P.
51
52      // Compute the public ephemeral key pQeA = [de,A]G
53      if((retVal = BnPointMult(pQeA, CurveGetG(C), bnDeA, NULL, NULL, E))
54         != TPM_RC_SUCCESS)
55          goto Exit;
56
57  //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.
58  //  tA := (ds,A + de,A  avf(Xe,A)) mod n    (3)
59  //  Compute 'tA' = ('deA' +  'dsA'  avf('XeA')) mod n
60      // Ta = avf(XeA);
61      BnCopy(bnTa, pQeA->x);
62      avf1(bnTa, bnN);
63      // do Ta = ds,A * Ta mod n = dsA * avf(XeA) mod n
64      BnModMult(bnTa, bnDsA, bnTa, bnN);
65      // now Ta = deA + Ta mod n =  deA + dsA * avf(XeA) mod n
66      BnAdd(bnTa, bnTa, bnDeA);
67      BnMod(bnTa, bnN);
68
69  //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
70  // Put this in because almost every case of h is == 1 so skip the call when
71      // not necessary.
72      if(!BnEqualWord(CurveGetCofactor(C), 1))
73          // Cofactor is not 1 so compute Ta := Ta * h mod n
74          BnModMult(bnTa, bnTa, CurveGetCofactor(C), CurveGetOrder(C));
75
76      // Now that 'tA' is (h * 'tA' mod n)
77      // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
78
79      // first, compute XeB = avf(XeB)
80      avf1(bnXeB, bnN);
81
82      // QsB := [XeB]QsB
83      BnPointMult(pQsB, pQsB, bnXeB, NULL, NULL, E);
84      BnEccAdd(pQeB, pQeB, pQsB, E);
85
86      // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
87      // If the result is not the point at infinity, return QeB
88      BnPointMult(pQeB, pQeB, bnTa, NULL, NULL, E);
89      if(BnEqualZero(pQeB->z))
90          ERROR_RETURN(TPM_RC_NO_RESULT);
91      // Convert BIGNUM E to TPM2B E
```



```
92      BnPointTo2B(outZ, pQeB, E);
93
94  Exit:
95      CURVE_FREE(E);
96      return retVal;
97  }
98  #endif // ALG_ECMQV
```

10.2.10.2.3 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).

```
99  static TPM_RC
100 C_2_2_ECDH(
101     TPMS_ECC_POINT          *outZs,         // OUT: Zs
102     TPMS_ECC_POINT          *outZe,         // OUT: Ze
103     TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
104     TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
105     TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
106     TPMS_ECC_POINT          *QsB,           // IN: static public party B key
107     TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
108     )
109 {
110     CURVE_INITIALIZED(E, curveId);
111     ECC_INITIALIZED(bnAs, dsA);
112     ECC_INITIALIZED(bnAe, deA);
113     POINT_INITIALIZED(ecBs, QsB);
114     POINT_INITIALIZED(ecBe, QeB);
115     POINT(ecZ);
116     TPM_RC              retVal;
117 //
118     // Parameter checks
119     if(E == NULL)
120         ERROR_RETURN(TPM_RC_CURVE);
121     pAssert(outZs != NULL && dsA != NULL && deA != NULL && QsB != NULL
122             && QeB != NULL);
123
124     // Do the point multiply for the Zs value ([dsA]QsB)
125     retVal = BnPointMult(ecZ, ecBs, bnAs, NULL, NULL, E);
126     if(retVal == TPM_RC_SUCCESS)
127     {
128         // Convert the Zs value.
129         BnPointTo2B(outZs, ecZ, E);
130         // Do the point multiply for the Ze value ([deA]QeB)
131         retVal = BnPointMult(ecZ, ecBe, bnAe, NULL, NULL, E);
132         if(retVal == TPM_RC_SUCCESS)
133             BnPointTo2B(outZe, ecZ, E);
134     }
135 Exit:
136     CURVE_FREE(E);
137     return retVal;
138 }
```

10.2.10.2.4 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

```
139 LIB_EXPORT TPM_RC
140 CryptEcc2PhaseKeyExchange(
141     TPMS_ECC_POINT          *outZ1,         // OUT: a computed point
142     TPMS_ECC_POINT          *outZ2,         // OUT: and optional second point
143     TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
144     TPM_ALG_ID               scheme,        // IN: the key exchange scheme
145     TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
146     TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
147     TPMS_ECC_POINT          *QsB,           // IN: static public party B key
148     TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
149     )
150 {
151     pAssert(outZ1 != NULL
152             && dsA != NULL && deA != NULL
153             && QsB != NULL && QeB != NULL);
154
155     // Initialize the output points so that they are empty until one of the
156     // functions decides otherwise
157     outZ1->x.b.size = 0;
158     outZ1->y.b.size = 0;
159     if(outZ2 != NULL)
160     {
161         outZ2->x.b.size = 0;
162         outZ2->y.b.size = 0;
163     }
164     switch(scheme)
165     {
166         case ALG_ECDH_VALUE:
167             return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
168             break;
169 #if     ALG_ECMQV
170         case ALG_ECMQV_VALUE:
171             return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
172             break;
173 #endif
174 #if     ALG_SM2
175         case ALG_SM2_VALUE:
176             return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
177             break;
178 #endif
179         default:
180             return TPM_RC_SCHEME;
181     }
182 }
183 #if     ALG_SM2
```

10.2.10.2.5 ComputeWForSM2()

Compute the value for w used by SM2

```
184 static UINT32
185 ComputeWForSM2(
186     bigCurve        E
187     )
188 {
189     //  w := ceil(ceil(log2(n)) / 2) - 1
190     return (BnMsb(CurveGetOrder(AccessCurveData(E))) / 2 - 1);
191 }
```


10.2.10.2.6 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.

```
192 static bigNum
193 avfSm2(
194     bigNum              bn,             // IN/OUT: the reduced value
195     UINT32              w               // IN: the value of w
196     )
197 {
198     // a)   set w := ceil(ceil(log2(n)) / 2) - 1
199     // b)   set x' := 2^w + ( x & (2^w - 1))
200     // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
201
202     BnMaskBits(bn, w);   // as with avf1, this is too big by a factor of 2 but
203                          // it doesn't matter because we SET the extra bit
204                          // anyway
205     BnSetBit(bn, w);
206     return bn;
207 }
```

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_NO_RESULT	the value for *dsA* does not give a valid point on the curve

```
208 LIB_EXPORT TPM_RC
209 SM2KeyExchange(
210     TPMS_ECC_POINT        *outZ,          // OUT: the computed point
211     TPM_ECC_CURVE          curveId,       // IN: the curve for the computations
212     TPM2B_ECC_PARAMETER   *dsAIn,         // IN: static private TPM key
213     TPM2B_ECC_PARAMETER   *deAIn,         // IN: ephemeral private TPM key
214     TPMS_ECC_POINT        *QsBIn,         // IN: static public party B key
215     TPMS_ECC_POINT        *QeBIn          // IN: ephemeral public party B key
216     )
217 {
218     CURVE_INITIALIZED(E, curveId);
219     const ECC_CURVE_DATA      *C;
220     ECC_INITIALIZED(dsA, dsAIn);
221     ECC_INITIALIZED(deA, deAIn);
222     POINT_INITIALIZED(QsB, QsBIn);
223     POINT_INITIALIZED(QeB, QeBIn);
224     BN_WORD_INITIALIZED(One, 1);
225     POINT(QeA);
226     ECC_NUM(XeB);
227     POINT(Z);
228     ECC_NUM(Ta);
229     UINT32                    w;
230     TPM_RC                  retVal = TPM_RC_NO_RESULT;
231 //
232     // Parameter checks
233     if(E == NULL)
234         ERROR_RETURN(TPM_RC_CURVE);
235     C = AccessCurveData(E);
```

```
236         pAssert(outZ != NULL && dsA != NULL && deA != NULL &&  QsB != NULL
237                 && QeB != NULL);
238
239         // Compute the value for w
240         w = ComputeWForSM2(E);
241
242         // Compute the public ephemeral key pQeA = [de,A]G
243         if(!BnEccModMult(QeA, CurveGetG(C), deA, E))
244             goto Exit;
245
246         //  tA := (ds,A + de,A  avf(Xe,A)) mod n    (3)
247         //  Compute 'tA' = ('dsA' +  'deA'  avf('XeA')) mod n
248         // Ta = avf(XeA);
249         // do Ta = de,A * Ta = deA * avf(XeA)
250         BnMult(Ta, deA, avfSm2(QeA->x, w));
251         // now Ta = dsA + Ta =  dsA + deA * avf(XeA)
252         BnAdd(Ta, dsA, Ta);
253         BnMod(Ta, CurveGetOrder(C));
254
255         //  outZ = [h  tA mod n] (Qs,B + [avf(Xe,B)](Qe,B)) (4)
256         // Put this in because almost every case of h is == 1 so skip the call when
257         // not necessary.
258         if(!BnEqualWord(CurveGetCofactor(C), 1))
259             // Cofactor is not 1 so compute Ta := Ta * h mod n
260             BnModMult(Ta, Ta, CurveGetCofactor(C), CurveGetOrder(C));
261         // Now that 'tA' is (h * 'tA' mod n)
262         // 'outZ' = ['tA'](QsB + [avf(QeB.x)](QeB)).
263         BnCopy(XeB, QeB->x);
264         if(!BnEccModMult2(Z, QsB, One, QeB, avfSm2(XeB, w), E))
265             goto Exit;
266         // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
267         if(!BnEccModMult(Z, Z, Ta, E))
268             goto Exit;
269         // Convert BIGNUM E to TPM2B E
270         BnPointTo2B(outZ, Z, E);
271         retVal = TPM_RC_SUCCESS;
272     Exit:
273         CURVE_FREE(E);
274         return retVal;
275     }
276     #endif
277     #endif // CC_ZGen_2Phase
```

10.2.11 CryptEccMain.c

10.2.11.1 Includes and Defines

```
#include "Tpm.h"
#if     ALG_ECC
```

This version requires that the new format for ECC data be used

```
#if !USE_BN_ECC_DATA
#error "Need to SET USE_BN_ECC_DATA to YES in Implementaion.h"
#endif
```

10.2.11.2 Functions

```
#if SIMULATION
void
EccSimulationEnd(
    void
    )
{
#if SIMULATION
// put things to be printed at the end of the simulation here
#endif
}
#endif // SIMULATION
```

10.2.11.2.1 CryptEccInit()

This function is called at _TPM_Init()

```
BOOL
CryptEccInit(
    void
    )
{
    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
```

```
34      )
35  {
36      if(p != NULL)
37      {
38          p->x.t.size = 0;
39          p->y.t.size = 0;
40      }
41  }
```

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 is not implemented
!= NULL	pointer to the curve data

```
42  LIB_EXPORT const ECC_CURVE *
43  CryptEccGetParametersByCurveId(
44      TPM_ECC_CURVE       curveId     // IN: the curveID
45      )
46  {
47      int           i;
48      for(i = 0; i < ECC_CURVE_COUNT; i++)
49      {
50          if(eccCurves[i].curveId == curveId)
51              return &eccCurves[i];
52      }
53      return NULL;
54  }
```

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);
```



```
    return (curve != NULL) ? curve->curveData : NULL;
}
```

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.

```
LIB_EXPORT TPM_ECC_CURVE
CryptEccGetCurveByIndex(
    UINT16                i
    )
{
    if(i >= ECC_CURVE_COUNT)
        return TPM_ECC_NONE;
    return eccCurves[i].curveId;
}
```

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(1)	curve exists and parameter returned
FALSE(0)	curve does not exist or parameter selector

```
LIB_EXPORT BOOL
CryptEccGetParameter(
    TPM2B_ECC_PARAMETER     *out,       // OUT: place to put parameter
    char                     p,         // IN: the parameter selector
    TPM_ECC_CURVE            curveId    // IN: the curve id
    )
{
    const ECC_CURVE_DATA    *curve = GetCurveData(curveId);
    bigConst                 parameter = NULL;

    if(curve != NULL)
    {
        switch(p)
        {
            case 'p':
                parameter = CurveGetPrime(curve);
                break;
            case 'n':
                parameter = CurveGetOrder(curve);
                break;
            case 'a':
                parameter = CurveGet_a(curve);
                break;
            case 'b':
                parameter = CurveGet_b(curve);
                break;
            case 'x':
                parameter = CurveGetGx(curve);
                break;
            case 'y':
                parameter = CurveGetGy(curve);
                break;
```

```
115                 case 'h':
116                     parameter = CurveGetCofactor(curve);
117                     break;
118                 default:
119                     FAIL(FATAL_ERROR_INTERNAL);
120                     break;
121             }
122         }
123         // If not debugging and we get here with parameter still NULL, had better
124         // not try to convert so just return FALSE instead.
125         return (parameter != NULL) ? BnTo2B(parameter, &out->b, 0) : 0;
126     }
```

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

```
127     TPMI_YES_NO
128     CryptCapGetECCCurve(
129         TPM_ECC_CURVE    curveID,       // IN: the starting ECC curve
130         UINT32           maxCount,      // IN: count of returned curves
131         TPML_ECC_CURVE  *curveList      // OUT: ECC curve list
132         )
133     {
134         TPMI_YES_NO       more = NO;
135         UINT16            i;
136         UINT32            count = ECC_CURVE_COUNT;
137         TPM_ECC_CURVE     curve;
138
139         // Initialize output property list
140         curveList->count = 0;
141
142         // The maximum count of curves we may return is MAX_ECC_CURVES
143         if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
144
145         // Scan the eccCurveValues array
146         for(i = 0; i < count; i++)
147         {
148             curve = CryptEccGetCurveByIndex(i);
149             // If curveID is less than the starting curveID, skip it
150             if(curve < curveID)
151                 continue;
152             if(curveList->count < maxCount)
153             {
154                 // If we have not filled up the return list, add more curves to
155                 // it
156                 curveList->eccCurves[curveList->count] = curve;
157                 curveList->count++;
158             }
159             else
160             {
161                 // If the return list is full but we still have curves
162                 // available, report this and stop iterating
163                 more = YES;
164                 break;
165             }
166         }
167         return more;
```



```
}
```

10.2.11.2.10 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```
const TPMT_ECC_SCHEME *
CryptGetCurveSignScheme(
    TPM_ECC_CURVE    curveId        // IN: The curve selector
    )
{
    const ECC_CURVE         *curve = CryptEccGetParametersByCurveId(curveId);

    if(curve != NULL)
        return &(curve->sign);
    else
        return NULL;
}
```

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(1)	r value computed
FALSE(0)	no r value computed

```
BOOL
CryptGenerateR(
    TPM2B_ECC_PARAMETER     *r,             // OUT: the generated random value
    UINT16                  *c,             // IN/OUT: count value.
    TPMI_ECC_CURVE           curveID,       // IN: the curve for the value
    TPM2B_NAME              *name           // IN: optional name of a key to
                                            //     associate with 'r'
    )
{
    // This holds the marshaled g_commitCounter.
    TPM2B_TYPE(8B, 8);
    TPM2B_8B                 cntr = {{8,{0}}};
    UINT32                   iterations;
    TPM2B_ECC_PARAMETER      n;
    UINT64                   currentCount = gr.commitCounter;
    UINT16                   t1;
//
    if(!CryptEccGetParameter(&n, 'n', curveID))
        return FALSE;

     // If this is the commit phase, use the current value of the commit counter
    if(c != NULL)
    {
        // if the array bit is not set, can't use the value.
        if(!TEST_BIT((*c & COMMIT_INDEX_MASK), gr.commitArray))
            return FALSE;

        // If it is the sign phase, figure out what the counter value was
        // when the commitment was made.
        //
```

```
211         // When gr.commitArray has less than 64K bits, the extra
212         // bits of 'c' are used as a check to make sure that the
213         // signing operation is not using an out of range count value
214         t1 = (UINT16)currentCount;
215
216         // If the lower bits of c are greater or equal to the lower bits of t1
217         // then the upper bits of t1 must be one more than the upper bits
218         // of c
219         if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
220             // Since the counter is behind, reduce the current count
221             currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
222
223         t1 = (UINT16)currentCount;
224         if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
225             return FALSE;
226         // set the counter to the value that was
227         // present when the commitment was made
228         currentCount = (currentCount & 0xffffffffffff0000) | *c;
229     }
230     // Marshal the count value to a TPM2B buffer for the KDF
231     cntr.t.size = sizeof(currentCount);
232     UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
233
234     // Now can do the KDF to create the random value for the signing operation
235     // During the creation process, we may generate an r that does not meet the
236     // requirements of the random value.
237     // want to generate a new r.
238     r->t.size = n.t.size;
239
240     for(iterations = 1; iterations < 1000000;)
241     {
242         int     i;
243         CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, COMMIT_STRING,
244                   &name->b, &cntr.b, n.t.size * 8, r->t.buffer, &iterations, FALSE);
245
246         // "random" value must be less than the prime
247         if(UnsignedCompareB(r->b.size, r->b.buffer, n.t.size, n.t.buffer) >= 0)
248             continue;
249
250         // in this implementation it is required that at least bit
251         // in the upper half of the number be set
252         for(i = n.t.size / 2; i >= 0; i--)
253             if(r->b.buffer[i] != 0)
254                 return TRUE;
255     }
256     return FALSE;
257 }
```

10.2.11.2.12 CryptCommit()

This function is called when the count value is committed. The gr.*commitArray* value associated with the current count value is SET and *g_commitCounter* is incremented. The low-order 16 bits of old value of the counter is returned.

```
258 UINT16
259 CryptCommit(
260     void
261     )
262 {
263     UINT16      oldCount = (UINT16)gr.commitCounter;
264     gr.commitCounter++;
265     SET_BIT(oldCount & COMMIT_INDEX_MASK, gr.commitArray);
266     return oldCount;
267 }
```

10.2.11.2.13 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the gr.*commitArray* bit associated with the count value so that it can't be used again.

```
268 void
269 CryptEndCommit(
270     UINT16          c               // IN: the counter value of the commitment
271     )
272 {
273     ClearBit((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
274 }
```

10.2.11.2.14 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve

Return Value	Meaning
TRUE(1)	success
FALSE(0)	unsupported ECC curve ID

```
275 BOOL
276 CryptEccGetParameters(
277     TPM_ECC_CURVE                 curveId,        // IN: ECC curve ID
278     TPMS_ALGORITHM_DETAIL_ECC    *parameters      // OUT: ECC parameters
279     )
280 {
281     const ECC_CURVE              *curve = CryptEccGetParametersByCurveId(curveId);
282     const ECC_CURVE_DATA         *data;
283     BOOL                          found = curve != NULL;
284
285     if(found)
286     {
287         data = curve->curveData;
288         parameters->curveID = curve->curveId;
289         parameters->keySize = curve->keySizeBits;
290         parameters->kdf = curve->kdf;
291         parameters->sign = curve->sign;
292         BnTo2B(data->prime, &parameters->p.b, 0);
293         BnTo2B(data->a, &parameters->a.b, 0);
294         BnTo2B(data->b, &parameters->b.b, 0);
295        BnTo2B(data->base.x, &parameters->gX.b, parameters->p.t.size);
296        BnTo2B(data->base.y, &parameters->gY.b, parameters->p.t.size);
297 //         BnTo2B(data->base.x, &parameters->gX.b, 0);
298 //         BnTo2B(data->base.y, &parameters->gY.b, 0);
299         BnTo2B(data->order, &parameters->n.b, 0);
300         BnTo2B(data->h, &parameters->h.b, 0);
301     }
302     return found;
303 }
```

10.2.11.2.15 BnGetCurvePrime()

This function is used to get just the prime modulus associated with a curve

```
304 const bignum_t *
305 BnGetCurvePrime(
306     TPM_ECC_CURVE            curveId
307     )
308 {
```

```
309     const ECC_CURVE_DATA    *C = GetCurveData(curveId);
310     return (C != NULL) ? CurveGetPrime(C) : NULL;
311 }
```

10.2.11.2.16 BnGetCurveOrder()

This function is used to get just the curve order

```
312 const bignum_t *
313 BnGetCurveOrder(
314     TPM_ECC_CURVE            curveId
315     )
316 {
317     const ECC_CURVE_DATA    *C = GetCurveData(curveId);
318     return (C != NULL) ? CurveGetOrder(C) : NULL;
319 }
```

10.2.11.2.17 BnIsOnCurve()

This function checks if a point is on the curve.

```
320 BOOL
321 BnIsOnCurve(
322     pointConst                  Q,
323     const ECC_CURVE_DATA        *C
324     )
325 {
326     BN_VAR(right, (MAX_ECC_KEY_BITS * 3));
327     BN_VAR(left, (MAX_ECC_KEY_BITS * 2));
328     bigConst                    prime = CurveGetPrime(C);
329 //
330     // Show that point is on the curve y^2 = x^3 + ax + b;
331     // Or y^2 = x(x^2 + a) + b
332     // y^2
333     BnMult(left, Q->y, Q->y);
334
335     BnMod(left, prime);
336 // x^2
337     BnMult(right, Q->x, Q->x);
338
339     // x^2 + a
340     BnAdd(right, right, CurveGet_a(C));
341
342 //    BnMod(right, CurveGetPrime(C));
343     // x(x^2 + a)
344     BnMult(right, right, Q->x);
345
346     // x(x^2 + a) + b
347     BnAdd(right, right, CurveGet_b(C));
348
349     BnMod(right, prime);
350     if(BnUnsignedCmp(left, right) == 0)
351         return TRUE;
352     else
353         return FALSE;
354 }
```

10.2.11.2.18 BnIsValidPrivateEcc()

Checks that 0 < x < q


```
355 BOOL
356 BnIsValidPrivateEcc(
357     bigConst                x,          // IN: private key to check
358     bigCurve                E           // IN: the curve to check
359     )
360 {
361     BOOL        retVal;
362     retVal = (!BnEqualZero(x)
363               && (BnUnsignedCmp(x, CurveGetOrder(AccessCurveData(E))) < 0));
364     return retVal;
365 }
366 LIB_EXPORT BOOL
367 CryptEccIsValidPrivateKey(
368     TPM2B_ECC_PARAMETER     *d,
369     TPM_ECC_CURVE            curveId
370     )
371 {
372     BN_INITIALIZED(bnD, MAX_ECC_PARAMETER_BYTES * 8, d);
373     return !BnEqualZero(bnD) && (BnUnsignedCmp(bnD, BnGetCurveOrder(curveId)) < 0);
374 }
```

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_RESULT if the resulting point is the point at infinity.

Error Returns	Meaning
TPM_RC_NO_RESULT	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 < n

```
375 TPM_RC
376 BnPointMult(
377     bigPoint            R,          // OUT: computed point
378     pointConst          S,          // IN: optional point to multiply by 'd'
379     bigConst            d,          // IN: scalar for [d]S or [d]G
380     pointConst          Q,          // IN: optional second point
381     bigConst            u,          // IN: optional second scalar
382     bigCurve            E           // IN: curve parameters
383     )
384 {
385     BOOL                OK;
386 //
387     TEST(TPM_ALG_ECDH);
388
389     // Need one scalar
390     OK = (d != NULL || u != NULL);
391
392     // If S is present, then d has to be present. If S is not
393     // present, then d may or may not be present
394     OK = OK && (((S == NULL) == (d == NULL)) || (d != NULL));
395
396     // either both u and Q have to be provided or neither can be provided (don't
397     // know what to do if only one is provided.
398     OK = OK && ((u == NULL) == (Q == NULL));
399
400     OK = OK && (E != NULL);
401     if(!OK)
```

```
402         return TPM_RC_VALUE;
403
404     OK = (S == NULL) || BnIsOnCurve(S, AccessCurveData(E));
405     OK = OK && ((Q == NULL) || BnIsOnCurve(Q, AccessCurveData(E)));
406     if(!OK)
407         return TPM_RC_ECC_POINT;
408
409     if((d != NULL) && (S == NULL))
410         S = CurveGetG(AccessCurveData(E));
411     // If only one scalar, don't need Shamir's trick
412     if((d == NULL) || (u == NULL))
413     {
414         if(d == NULL)
415             OK = BnEccModMult(R, Q, u, E);
416         else
417             OK = BnEccModMult(R, S, d, E);
418     }
419     else
420     {
421         OK = BnEccModMult2(R, S, d, Q, u, E);
422     }
423     return  (OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT);
424 }
```

10.2.11.2.20 BnEccGetPrivate()

This function gets random values that are the size of the key plus 64 bits. The value is reduced (mod *(q* - 1)) and incremented by 1 *(q* is the order of the curve. This produces a value *(d*) such that 1 <= *d* < *q*. This is the method of FIPS 186-4 Section B.4.1 'Key Pair Generation Using Extra Random Bits'.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure generating private key

```
425 BOOL
426 BnEccGetPrivate(
427     bigNum                  dOut,       // OUT: the qualified random value
428     const ECC_CURVE_DATA    *C,         // IN: curve for which the private key
429                                         //     needs to be appropriate
430     RAND_STATE              *rand       // IN: state for DRBG
431     )
432 {
433     bigConst                order = CurveGetOrder(C);
434     BOOL                    OK;
435     UINT32                  orderBits = BnSizeInBits(order);
436     UINT32                  orderBytes = BITS_TO_BYTES(orderBits);
437     BN_VAR(bnExtraBits, MAX_ECC_KEY_BITS + 64);
438     BN_VAR(nMinus1, MAX_ECC_KEY_BITS);
439 //
440     OK = BnGetRandomBits(bnExtraBits, (orderBytes * 8) + 64, rand);
441     OK = OK && BnSubWord(nMinus1, order, 1);
442     OK = OK && BnMod(bnExtraBits, nMinus1);
443     OK = OK && BnAddWord(dOut, bnExtraBits, 1);
444     return OK;
445 }
```

10.2.11.2.21 BnEccGenerateKeyPair()

This function gets a private scalar from the source of random bits and does the point multiply to get the public key.


```
446 BOOL
447 BnEccGenerateKeyPair(
448     bigNum                bnD,            // OUT: private scalar
449     bn_point_t           *ecQ,            // OUT: public point
450     bigCurve              E,              // IN: curve for the point
451     RAND_STATE           *rand            // IN: DRBG state to use
452     )
453 {
454     BOOL                  OK = FALSE;
455     // Get a private scalar
456     OK = BnEccGetPrivate(bnD, AccessCurveData(E), rand);
457
458     // Do a point multiply
459     OK = OK && BnEccModMult(ecQ, NULL, bnD, E);
460     if(!OK)
461         BnSetWord(ecQ->z, 0);
462     else
463         BnSetWord(ecQ->z, 1);
464     return OK;
465 }
```

10.2.11.2.22 CryptEccNewKeyPair

This function creates an ephemeral ECC. It is ephemeral in that is expected that the private part of the key will be discarded

```
466 LIB_EXPORT TPM_RC
467 CryptEccNewKeyPair(
468     TPMS_ECC_POINT           *Qout,      // OUT: the public point
469     TPM2B_ECC_PARAMETER      *dOut,      // OUT: the private scalar
470     TPM_ECC_CURVE             curveId    // IN: the curve for the key
471     )
472 {
473     CURVE_INITIALIZED(E, curveId);
474     POINT(ecQ);
475     ECC_NUM(bnD);
476     BOOL                     OK;
477
478     if(E == NULL)
479         return TPM_RC_CURVE;
480
481     TEST(TPM_ALG_ECDH);
482     OK = BnEccGenerateKeyPair(bnD, ecQ, E, NULL);
483     if(OK)
484     {
485         BnPointTo2B(Qout, ecQ, E);
486         BnTo2B(bnD, &dOut->b, Qout->x.t.size);
487     }
488     else
489     {
490         Qout->x.t.size = Qout->y.t.size = dOut->t.size = 0;
491     }
492     CURVE_FREE(E);
493     return OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
494 }
```

10.2.11.2.23 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_RESULT is returned.

The sizes of *xOut* and *yOut*' will be set to be the size of the degree of the curve

It is a fatal error if *dIn* and *uIn* are both unspecified (NULL) or if *Qin* or *Rout* is unspecified.

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

```
495  LIB_EXPORT TPM_RC
496  CryptEccPointMultiply(
497      TPMS_ECC_POINT      *Rout,              // OUT: the product point R
498      TPM_ECC_CURVE        curveId,           // IN: the curve to use
499      TPMS_ECC_POINT      *Pin,               // IN: first point (can be null)
500      TPM2B_ECC_PARAMETER *dIn,               // IN: scalar value for [dIn]Qin
501                                              //     the Pin
502      TPMS_ECC_POINT      *Qin,               // IN: point Q
503      TPM2B_ECC_PARAMETER *uIn                // IN: scalar value for the multiplier
504                                              //     of Q
505      )
506  {
507      CURVE_INITIALIZED(E, curveId);
508      POINT_INITIALIZED(ecP, Pin);
509      ECC_INITIALIZED(bnD, dIn);      // If dIn is null, then bnD is null
510      ECC_INITIALIZED(bnU, uIn);
511      POINT_INITIALIZED(ecQ, Qin);
512      POINT(ecR);
513      TPM_RC              retVal;
514  //
515      retVal = BnPointMult(ecR, ecP, bnD, ecQ, bnU, E);
516
517      if(retVal == TPM_RC_SUCCESS)
518          BnPointTo2B(Rout, ecR, E);
519      else
520          ClearPoint2B(Rout);
521      CURVE_FREE(E);
522      return retVal;
523  }
```

10.2.11.2.24 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(1)	point is on curve
FALSE(0)	point is not on curve or curve is not supported

```
524  LIB_EXPORT BOOL
525  CryptEccIsPointOnCurve(
526      TPM_ECC_CURVE            curveId,       // IN: the curve selector
```

```
527         TPMS_ECC_POINT          *Qin            // IN: the point.
528         )
529     {
530         const ECC_CURVE_DATA    *C = GetCurveData(curveId);
531         POINT_INITIALIZED(ecQ, Qin);
532         BOOL            OK;
533     //
534         pAssert(Qin != NULL);
535         OK = (C != NULL && (BnIsOnCurve(ecQ, C)));
536         return OK;
537     }
```

10.2.11.2.25 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_NO_RESULT	could not verify key with signature (FIPS only)

```
538     LIB_EXPORT TPM_RC
539     CryptEccGenerateKey(
540         TPMT_PUBLIC         *publicArea,        // IN/OUT: The public area template for
541                                                 //      the new key. The public key
542                                                 //      area will be replaced computed
543                                                 //      ECC public key
544         TPMT_SENSITIVE      *sensitive,         // OUT: the sensitive area will be
545                                                 //      updated to contain the private
546                                                 //      ECC key and the symmetric
547                                                 //      encryption key
548         RAND_STATE          *rand               // IN: if not NULL, the deterministic
549                                                 //     RNG state
550         )
551     {
552         CURVE_INITIALIZED(E, publicArea->parameters.eccDetail.curveID);
553         ECC_NUM(bnD);
554         POINT(ecQ);
555         BOOL                    OK;
556         TPM_RC                  retVal;
557     //
558         TEST(TPM_ALG_ECDSA); // ECDSA is used to verify each key
559
560         // Validate parameters
561         if(E == NULL)
562             ERROR_RETURN(TPM_RC_CURVE);
563
564         publicArea->unique.ecc.x.t.size = 0;
565         publicArea->unique.ecc.y.t.size = 0;
566         sensitive->sensitive.ecc.t.size = 0;
567
568         OK = BnEccGenerateKeyPair(bnD, ecQ, E, rand);
569         if(OK)
570         {
571             BnPointTo2B(&publicArea->unique.ecc, ecQ, E);
572             BnTo2B(bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
```

```
573         }
574     #if FIPS_COMPLIANT
575         // See if PWCT is required
576         if(OK && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
577         {
578             ECC_NUM(bnT);
579             ECC_NUM(bnS);
580             TPM2B_DIGEST            digest;
581     //
582             TEST(TPM_ALG_ECDSA);
583             digest.t.size = MIN(sensitive->sensitive.ecc.t.size, sizeof(digest.t.buffer));
584             // Get a random value to sign using the built in DRBG state
585             DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
586             BnSignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
587             // and make sure that we can validate the signature
588             OK = BnValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest) == TPM_RC_SUCCESS;
589         }
590     #endif
591         retVal = (OK) ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
592     Exit:
593         CURVE_FREE(E);
594         return retVal;
595     }
596     #endif  // ALG_ECC
```


10.2.12 CryptEccSignature.c

10.2.12.1 Includes and Defines

```
#include "Tpm.h"
#include "CryptEccSignature_fp.h"
#if     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

```
static TPM_RC
BnSchnorrSign(
    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(MAX_ECC_KEY_BYTES, 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
89      do // This implements the loop at step 6. If s is zero, start over.
90      {
91          for(; tries > 0; tries--)
92          {
93              // Step 1 and 2 -- generate an ephemeral key and the modular inverse
94              // of the private key.
95              if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
96                  continue;
97           // x coordinate is mod p.  Make it mod q
98              BnMod(ecR->x, order);
99              // Make sure that it is not zero;
100             if(BnEqualZero(ecR->x))
101                 continue;
102             // write the modular reduced version of r as part of the signature
103             BnCopy(bnR, ecR->x);
104             // Make sure that a modular inverse exists and try again if not
105             OK = (BnModInverse(bnIk, bnK, order));
106             if(OK)
107                 break;
108         }
109         if(!OK)
110             goto Exit;
111
112         EcdsaDigest(bnE, digest, order);
113
114         // now have inverse of K (bnIk), e (bnE), r (bnR),  d (bnD) and
115         // CurveGetOrder(E)
116         // Compute s = k^-1 (e + r*d)(mod q)
117         //  first do s = r*d mod q
118         BnModMult(bnS, bnR, bnD, order);
119         // s = e + s = e + r * d
120         BnAdd(bnS, bnE, bnS);
121         // s = k^(-1)s (mod n) = k^(-1)(e + r * d)(mod n)
122         BnModMult(bnS, bnIk, bnS, order);
123
124         // If S is zero, try again
125     } while(BnEqualZero(bnS));
126 Exit:
127     return retVal;
128 }
129 #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

Error Returns	Meaning
TPM_RC_SCHEME	unsupported hash algorithm
TPM_RC_NO_RESULT	cannot get values from random number generator

```
130 static TPM_RC
131 BnSignEcdaa(
132     TPM2B_ECC_PARAMETER     *nonceK,        // OUT: nonce component of the signature
133     bigNum                   bnS,           // OUT: s component of the signature
134     bigCurve                 E,             // IN: the curve used in signing
135     bigNum                   bnD,           // IN: the private key
136     const TPM2B_DIGEST      *digest,        // IN: the value to sign (mod q)
137     TPMT_ECC_SCHEME         *scheme,        // IN: signing scheme (contains the
138                                             //      commit count value).
139     OBJECT                  *eccKey,        // IN: The signing key
140     RAND_STATE              *rand           // IN: a random number state
141     )
142 {
143     TPM_RC                   retVal;
144     TPM2B_ECC_PARAMETER      r;
145     HASH_STATE               state;
146     TPM2B_DIGEST             T;
147     BN_MAX(bnT);
148 //
149     NOT_REFERENCED(rand);
150     if(!CryptGenerateR(&r, &scheme->details.ecdaa.count,
151                        eccKey->publicArea.parameters.eccDetail.curveID,
152                        &eccKey->name))
153         retVal = TPM_RC_VALUE;
154     else
155     {
156         // This allocation is here because 'r' doesn't have a value until
157         // CrypGenerateR() is done.
158         ECC_INITIALIZED(bnR, &r);
159         do
160         {
161             // generate nonceK such that 0 < nonceK < n
162             // use bnT as a temp.
163             if(!BnEccGetPrivate(bnT, AccessCurveData(E), rand))
164             {
165                 retVal = TPM_RC_NO_RESULT;
166                 break;
167             }
168             BnTo2B(bnT, &nonceK->b, 0);
169
170             T.t.size = CryptHashStart(&state, scheme->details.ecdaa.hashAlg);
171             if(T.t.size == 0)
172             {
173                 retVal = TPM_RC_SCHEME;
174             }
175             else
176             {
177                 CryptDigestUpdate2B(&state, &nonceK->b);
178                 CryptDigestUpdate2B(&state, &digest->b);
179                 CryptHashEnd2B(&state, &T.b);
180                 BnFrom2B(bnT, &T.b);
181                 // Watch out for the name collisions in this call!!
182                 retVal = BnSchnorrSign(bnS, bnR, bnT, bnD,
183                                        AccessCurveData(E)->order);
184             }
185         } while(retVal == TPM_RC_NO_RESULT);
186         // Because the rule is that internal state is not modified if the command
187         // fails, only end the commit if the command succeeds.
188         // NOTE that if the result of the Schnorr computation was zero
```



```
189         // it will probably not be worthwhile to run the same command again because
190         // the result will still be zero. This means that the Commit command will
191         // need to be run again to get a new commit value for the signature.
192         if(retVal == TPM_RC_SUCCESS)
193             CryptEndCommit(scheme->details.ecdaa.count);
194     }
195     return retVal;
196 }
197 #endif // ALG_ECDAA
198 #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.

```
199 static void
200 SchnorrReduce(
201     TPM2B       *number,        // IN/OUT: Value to reduce
202     bigConst     reference      // IN: the reference value
203     )
204 {
205     UINT16      maxBytes = (UINT16)BITS_TO_BYTES(BnSizeInBits(reference));
206     if(number->size > maxBytes)
207         number->size = maxBytes;
208 }
```

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) $r = \text{hash}(xR \,||\, P)(\text{mod } q)$

c) rT = truncated r

d) $s = k + rT * ds \ (\text{mod } q)$

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

```
209 static TPM_RC
210 BnSignEcSchnorr(
211     bigNum                  bnR,            // OUT: r component of the signature
212     bigNum                  bnS,            // OUT: s component of the signature
213     bigCurve                E,              // IN: the curve used in signing
214     bigNum                  bnD,            // IN: the signing key
215     const TPM2B_DIGEST     *digest,         // IN: the digest to sign
216     TPM_ALG_ID              hashAlg,        // IN: signing scheme (contains a hash)
217     RAND_STATE             *rand            // IN: non-NULL when testing
218     )
219 {
220     HASH_STATE              hashState;
221     UINT16                  digestSize = CryptHashGetDigestSize(hashAlg);
222     TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_KEY_BYTES));
```

```
223     TPM2B_T                 T2b;
224     TPM2B                  *e = &T2b.b;
225     TPM_RC                  retVal = TPM_RC_NO_RESULT;
226     const ECC_CURVE_DATA   *C;
227     bigConst                order;
228     bigConst                prime;
229     ECC_NUM(bnK);
230     POINT(ecR);
231 //
232     // Parameter checks
233     if(E == NULL)
234         ERROR_RETURN(TPM_RC_VALUE);
235     C = AccessCurveData(E);
236     order = CurveGetOrder(C);
237     prime = CurveGetOrder(C);
238
239     // If the digest does not produce a hash, then null the signature and return
240     // a failure.
241     if(digestSize == 0)
242     {
243         BnSetWord(bnR, 0);
244         BnSetWord(bnS, 0);
245         ERROR_RETURN(TPM_RC_SCHEME);
246     }
247     do
248     {
249         // Generate a random key pair
250         if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
251             break;
252         // Convert R.x to a string
253         BnTo2B(ecR->x, e, (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(prime)));
254
255         // f) compute r = Hash(e || P) (mod n)
256         CryptHashStart(&hashState, hashAlg);
257         CryptDigestUpdate2B(&hashState, e);
258         CryptDigestUpdate2B(&hashState, &digest->b);
259         e->size = CryptHashEnd(&hashState, digestSize, e->buffer);
260         // Reduce the hash size if it is larger than the curve order
261         SchnorrReduce(e, order);
262         // Convert hash to number
263         BnFrom2B(bnR, e);
264         // Do the Schnorr computation
265         retVal = BnSchnorrSign(bnS, bnK, bnR, bnD, CurveGetOrder(C));
266     } while(retVal == TPM_RC_NO_RESULT);
267 Exit:
268     return retVal;
269 }
270 #endif // ALG_ECSCHNORR
271 #if ALG_SM2
272 #ifdef  _SM2_SIGN_DEBUG
```

10.2.12.3.5 BnHexEqual()

This function compares a bignum value to a hex string.

Return Value	Meaning
TRUE(1)	values equal
FALSE(0)	values not equal

```
273 static BOOL
274 BnHexEqual(
275     bigNum          bn,         //IN: big number value
```

```
    const char      *c          //IN: character string number
    )
{
    ECC_NUM(bnC);
    BnFromHex(bnC, c);
    return (BnUnsignedCmp(bn, bnC) == 0);
}
#endif // _SM2_SIGN_DEBUG
```

10.2.12.3.6 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

```
static TPM_RC
BnSignEcSm2(
    bigNum                  bnR,        // OUT: r component of the signature
    bigNum                  bnS,        // OUT: s component of the signature
    bigCurve                E,          // IN: the curve used in signing
    bigNum                  bnD,        // IN: the private key
    const TPM2B_DIGEST     *digest,     // IN: the digest to sign
    RAND_STATE             *rand        // IN: random number generator (mostly for
                                        //     debug)
    )
{
    BN_MAX_INITIALIZED(bnE, digest);    // Don't know how big digest might be
    ECC_NUM(bnN);
    ECC_NUM(bnK);
    ECC_NUM(bnT);                       // temp
    POINT(Q1);
    bigConst                  order = (E != NULL)
        ? CurveGetOrder(AccessCurveData(E)) : NULL;
//
#ifdef _SM2_SIGN_DEBUG
    BnFromHex(bnE, "B524F552CD82B8B028476E005C377FB1"
                   "9A87E6FC682D48BB5D42E3D9B9EFFE76");
    BnFromHex(bnD, "128B2FA8BD433C6C068C8D803DFF7979"
                   "2A519A55171B1B650C23661D15897263");
#endif
    // A3: Use random number generator to generate random number 1 <= k <= n-1;
    // NOTE: Ax: numbers are from the SM2 standard
loop:
    {
        // Get a random number 0 < k < n
        BnGenerateRandomInRange(bnK, order, rand);
#ifdef _SM2_SIGN_DEBUG
        BnFromHex(bnK, "6CB28D99385C175C94F94E934817663F"
                       "C176D925DD72B727260DBAAE1FB2F96F");
#endif
        // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
        // to details specified in 4.2.7 in Part 1 of this document, transform the
        // data type of x1 into an integer;
        if(!BnEccModMult(Q1, NULL, bnK, E))
            goto loop;
        // A5: Figure out r = (e + x1) mod n,
        BnAdd(bnR, bnE, Q1->x);
        BnMod(bnR, order);
#ifdef _SM2_SIGN_DEBUG
        pAssert(BnHexEqual(bnR, "40F1EC59F793D9F49E09DCEF49130D41"
```

```
                                "94F79FB1EED2CAA55BACDB49C4E755D1"));
#endif
        // if r=0 or r+k=n, return to A3;
        if(BnEqualZero(bnR))
            goto loop;
        BnAdd(bnT, bnK, bnR);
        if(BnUnsignedCmp(bnT, bnN) == 0)
            goto loop;
        // A6: Figure out s = ((1 + dA)^-1  (k - r  dA)) mod n,
        // if s=0, return to A3;
        // compute t = (1+dA)^-1
        BnAddWord(bnT, bnD, 1);
        BnModInverse(bnT, bnT, order);
#ifdef _SM2_SIGN_DEBUG
        pAssert(BnHexEqual(bnT, "79BFCF3052C80DA7B939E0C6914A18CB"
                                "B2D96D8555256E83122743A7D4F5F956"));
#endif
        // compute s = t * (k - r * dA) mod n
        BnModMult(bnS, bnR, bnD, order);
        // k - r * dA mod n = k + n - ((r * dA) mod n)
        BnSub(bnS, order, bnS);
        BnAdd(bnS, bnK, bnS);
        BnModMult(bnS, bnS, bnT, order);
#ifdef _SM2_SIGN_DEBUG
        pAssert(BnHexEqual(bnS, "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
                                "67A457872FB09EC56327A67EC7DEEBE7"));
#endif
        if(BnEqualZero(bnS))
            goto loop;
    }
    // A7: According to details specified in 4.2.1 in Part 1 of this document,
    // transform the data type of r, s into bit strings, signature of message M
    // is (r, s).
    // This is handled by the common return code
#ifdef _SM2_SIGN_DEBUG
    pAssert(BnHexEqual(bnR, "40F1EC59F793D9F49E09DCEF49130D41"
                            "94F79FB1EED2CAA55BACDB49C4E755D1"));
    pAssert(BnHexEqual(bnS, "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
                            "67A457872FB09EC56327A67EC7DEEBE7"));
#endif
    return TPM_RC_SUCCESS;
}
#endif // ALG_SM2
```

10.2.12.3.7 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

```
LIB_EXPORT TPM_RC
CryptEccSign(
    TPMT_SIGNATURE          *signature,     // OUT: signature
    OBJECT                  *signKey,       // IN: ECC key to sign the hash
    const TPM2B_DIGEST      *digest,        // IN: digest to sign
```

```
377         TPMT_ECC_SCHEME          *scheme,        // IN: signing scheme
378         RAND_STATE               *rand
379         )
380     {
381         CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
382         ECC_INITIALIZED(bnD, &signKey->sensitive.sensitive.ecc.b);
383         ECC_NUM(bnR);
384         ECC_NUM(bnS);
385         const ECC_CURVE_DATA   *C;
386         TPM_RC                   retVal = TPM_RC_SCHEME;
387     //
388         NOT_REFERENCED(scheme);
389         if(E == NULL)
390             ERROR_RETURN(TPM_RC_VALUE);
391          C = AccessCurveData(E);
392         signature->signature.ecdaa.signatureR.t.size
393             = sizeof(signature->signature.ecdaa.signatureR.t.buffer);
394         signature->signature.ecdaa.signatureS.t.size
395             = sizeof(signature->signature.ecdaa.signatureS.t.buffer);
396         TEST(signature->sigAlg);
397         switch(signature->sigAlg)
398         {
399             case ALG_ECDSA_VALUE:
400                 retVal = BnSignEcdsa(bnR, bnS, E, bnD, digest, rand);
401                 break;
402     #if ALG_ECDAA
403             case ALG_ECDAA_VALUE:
404                 retVal = BnSignEcdaa(&signature->signature.ecdaa.signatureR, bnS, E,
405                                      bnD, digest, scheme, signKey, rand);
406                 bnR = NULL;
407                 break;
408     #endif
409     #if ALG_ECSCHNORR
410             case ALG_ECSCHNORR_VALUE:
411                 retVal = BnSignEcSchnorr(bnR, bnS, E, bnD, digest,
412                                          signature->signature.ecschnorr.hash,
413                                          rand);
414                 break;
415     #endif
416     #if ALG_SM2
417             case ALG_SM2_VALUE:
418                 retVal = BnSignEcSm2(bnR, bnS, E, bnD, digest, rand);
419                 break;
420     #endif
421             default:
422                 break;
423         }
424         // If signature generation worked, convert the results.
425         if(retVal == TPM_RC_SUCCESS)
426         {
427             NUMBYTES     orderBytes =
428                 (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(C)));
429             if(bnR != NULL)
430                 BnTo2B(bnR, &signature->signature.ecdaa.signatureR.b, orderBytes);
431             if(bnS != NULL)
432                 BnTo2B(bnS, &signature->signature.ecdaa.signatureS.b, orderBytes);
433         }
434     Exit:
435         CURVE_FREE(E);
436         return retVal;
437     }
438     #if ALG_ECDSA
```


10.2.12.3.8 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

```
439  TPM_RC
440  BnValidateSignatureEcdsa(
441      bigNum                  bnR,            // IN: r component of the signature
442      bigNum                  bnS,            // IN: s component of the signature
443      bigCurve                E,              // IN: the curve used in the signature
444                                              //     process
445      bn_point_t             *ecQ,            // IN: the public point of the key
446      const TPM2B_DIGEST     *digest          // IN: the digest that was signed
447      )
448  {
449      // Make sure that the allocation for the digest is big enough for a maximum
450      // digest
451      BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
452      POINT(ecR);
453      ECC_NUM(bnU1);
454      ECC_NUM(bnU2);
455      ECC_NUM(bnW);
456      bigConst                order = CurveGetOrder(AccessCurveData(E));
457      TPM_RC                  retVal = TPM_RC_SIGNATURE;
458  //
459      // Get adjusted digest
460      EcdsaDigest(bnE, digest, order);
461      // 1. If r and s are not both integers in the interval [1, n - 1], output
462      //    INVALID.
463      //  bnR  and bnS were validated by the caller
464      // 2. Use the selected hash function to compute H0 = Hash(M0).
465      // This is an input parameter
466      // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
467      // Done at entry
468      // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
469      if(!BnModInverse(bnW, bnS, order))
470          goto Exit;
471      // 5. Compute u1 = (e' *   w) mod n, and compute u2 = (r' *  w) mod n.
472      BnModMult(bnU1, bnE, bnW, order);
473      BnModMult(bnU2, bnR, bnW, order);
474      // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
475      //    scalar multiplication and EC addition (see [Routines]). If R is equal to
476      //    the point at infinity O, output INVALID.
477      if(BnPointMult(ecR, CurveGetG(AccessCurveData(E)), bnU1, ecQ, bnU2, E)
478         != TPM_RC_SUCCESS)
479          goto Exit;
480      // 7. Compute v = Rx mod n.
481      BnMod(ecR->x, order);
482      // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
483      if(BnUnsignedCmp(ecR->x, bnR) != 0)
484          goto Exit;
485
486      retVal = TPM_RC_SUCCESS;
487  Exit:
488      return retVal;
489  }
490  #endif      // ALG_ECDSA
491  #if ALG_SM2
```


10.2.12.3.9 BnValidateSignatureEcSm2()

This function is used to validate an SM2 signature.

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```
492  static TPM_RC
493  BnValidateSignatureEcSm2(
494      bigNum                  bnR,        // IN: r component of the signature
495      bigNum                  bnS,        // IN: s component of the signature
496      bigCurve                E,          // IN: the curve used in the signature
497                                          //     process
498      bigPoint                ecQ,        // IN: the public point of the key
499      const TPM2B_DIGEST     *digest      // IN: the digest that was signed
500      )
501  {
502      POINT(P);
503      ECC_NUM(bnRp);
504      ECC_NUM(bnT);
505      BN_MAX_INITIALIZED(bnE, digest);
506      BOOL                    OK;
507      bigConst                order = CurveGetOrder(AccessCurveData(E));
508
509  #ifdef _SM2_SIGN_DEBUG
510      // Make sure that the input signature is the test signature
511      pAssert(BnHexEqual(bnR,
512                         "40F1EC59F793D9F49E09DCEF49130D41"
513                         "94F79FB1EED2CAA55BACDB49C4E755D1"));
514      pAssert(BnHexEqual(bnS,
515                         "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
516                         "67A457872FB09EC56327A67EC7DEEBE7"));
517  #endif
518      // b)   compute t  := (r + s) mod n
519      BnAdd(bnT, bnR, bnS);
520      BnMod(bnT, order);
521  #ifdef _SM2_SIGN_DEBUG
522      pAssert(BnHexEqual(bnT,
523                         "2B75F07ED7ECE7CCC1C8986B991F441A"
524                         "D324D6D619FE06DD63ED32E0C997C801"));
525  #endif
526      // c)   verify that t > 0
527      OK = !BnEqualZero(bnT);
528      if(!OK)
529          // set T to a value that should allow rest of the computations to run
530          // without trouble
531          BnCopy(bnT, bnS);
532      // d)   compute (x, y) := [s]G + [t]Q
533      OK = BnEccModMult2(P, NULL, bnS, ecQ, bnT, E);
534  #ifdef  _SM2_SIGN_DEBUG
535      pAssert(OK && BnHexEqual(P->x,
536                               "110FCDA57615705D5E7B9324AC4B856D"
537                               "23E6D9188B2AE47759514657CE25D112"));
538  #endif
539      // e)   compute r' := (e + x) mod n (the x coordinate is in bnT)
540      OK = OK && BnAdd(bnRp, bnE, P->x);
541      OK = OK && BnMod(bnRp, order);
542
543      // f)   verify that r' = r
544      OK = OK && (BnUnsignedCmp(bnR, bnRp) == 0);
545
546      if(!OK)
547          return TPM_RC_SIGNATURE;
548      else
```



```
549         return TPM_RC_SUCCESS;
550 }
551 #endif  // ALG_SM2
552 #if ALG_ECSCHNORR
```

10.2.12.3.10 BnValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```
553 static TPM_RC
554 BnValidateSignatureEcSchnorr(
555     bigNum                bnR,        // IN: r component of the signature
556     bigNum                bnS,        // IN: s component of the signature
557     TPM_ALG_ID            hashAlg,    // IN: hash algorithm of the signature
558     bigCurve              E,          // IN: the curve used in the signature
559                                       //     process
560     bigPoint              ecQ,        // IN: the public point of the key
561     const TPM2B_DIGEST  *digest       // IN: the digest that was signed
562     )
563 {
564     BN_MAX(bnRn);
565     POINT(ecE);
566     BN_MAX(bnEx);
567     const ECC_CURVE_DATA    *C = AccessCurveData(E);
568     bigConst                 order = CurveGetOrder(C);
569     UINT16                   digestSize = CryptHashGetDigestSize(hashAlg);
570     HASH_STATE               hashState;
571     TPM2B_TYPE(BUFFER, MAX(MAX_ECC_PARAMETER_BYTES, MAX_DIGEST_SIZE));
572     TPM2B_BUFFER             Ex2 = {{sizeof(Ex2.t.buffer),{ 0 }}};
573     BOOL                     OK;
574 //
575     // E = [s]G - [r]Q
576     BnMod(bnR, order);
577     // Make -r = n - r
578     BnSub(bnRn, order, bnR);
579     // E = [s]G + [-r]Q
580     OK = BnPointMult(ecE, CurveGetG(C), bnS, ecQ, bnRn, E) == TPM_RC_SUCCESS;
581 //   // reduce the x portion of E mod q
582 //     OK = OK && BnMod(ecE->x, order);
583     // Convert to byte string
584     OK = OK && BnTo2B(ecE->x, &Ex2.b,
585                       (NUMBYTES)(BITS_TO_BYTES(BnSizeInBits(order))));
586     if(OK)
587     {
588 // Ex = h(pE.x || digest)
589         CryptHashStart(&hashState, hashAlg);
590         CryptDigestUpdate(&hashState, Ex2.t.size, Ex2.t.buffer);
591         CryptDigestUpdate(&hashState, digest->t.size, digest->t.buffer);
592         Ex2.t.size = CryptHashEnd(&hashState, digestSize, Ex2.t.buffer);
593         SchnorrReduce(&Ex2.b, order);
594         BnFrom2B(bnEx, &Ex2.b);
595         // see if Ex matches R
596         OK = BnUnsignedCmp(bnEx, bnR) == 0;
597     }
598     return (OK) ? TPM_RC_SUCCESS : TPM_RC_SIGNATURE;
599 }
600 #endif  // ALG_ECSCHNORR
```


10.2.12.3.11 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

```
601  LIB_EXPORT TPM_RC
602  CryptEccValidateSignature(
603      TPMT_SIGNATURE          *signature,     // IN: signature to be verified
604      OBJECT                  *signKey,       // IN: ECC key signed the hash
605      const TPM2B_DIGEST      *digest         // IN: digest that was signed
606      )
607  {
608      CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
609      ECC_NUM(bnR);
610      ECC_NUM(bnS);
611      POINT_INITIALIZED(ecQ, &signKey->publicArea.unique.ecc);
612      bigConst                 order;
613      TPM_RC                   retVal;
614
615      if(E == NULL)
616          ERROR_RETURN(TPM_RC_VALUE);
617
618      order = CurveGetOrder(AccessCurveData(E));
619
620  //    // Make sure that the scheme is valid
621      switch(signature->sigAlg)
622      {
623          case ALG_ECDSA_VALUE:
624  #if ALG_ECSCHNORR
625          case ALG_ECSCHNORR_VALUE:
626  #endif
627  #if ALG_SM2
628          case ALG_SM2_VALUE:
629  #endif
630              break;
631          default:
632              ERROR_RETURN(TPM_RC_SCHEME);
633              break;
634      }
635      // Can convert r and s after determining that the scheme is an ECC scheme. If
636      // this conversion doesn't work, it means that the unmarshaling code for
637      // an ECC signature is broken.
638      BnFrom2B(bnR, &signature->signature.ecdsa.signatureR.b);
639      BnFrom2B(bnS, &signature->signature.ecdsa.signatureS.b);
640
641      // r and s have to be greater than 0 but less than the curve order
642      if(BnEqualZero(bnR) || BnEqualZero(bnS))
643          ERROR_RETURN(TPM_RC_SIGNATURE);
644      if((BnUnsignedCmp(bnS, order) >= 0)
645         || (BnUnsignedCmp(bnR, order) >= 0))
646          ERROR_RETURN(TPM_RC_SIGNATURE);
647
648      switch(signature->sigAlg)
649      {
650          case ALG_ECDSA_VALUE:
651              retVal = BnValidateSignatureEcdsa(bnR, bnS, E, ecQ, digest);
652              break;
653
654  #if ALG_ECSCHNORR
655          case ALG_ECSCHNORR_VALUE:
656              retVal = BnValidateSignatureEcSchnorr(bnR, bnS,
```

```
                                                signature->signature.any.hashAlg,
                                                E, ecQ, digest);
            break;
#endif
#if ALG_SM2
        case ALG_SM2_VALUE:
            retVal = BnValidateSignatureEcSm2(bnR, bnS, E, ecQ, digest);
            break;
#endif
        default:
            FAIL(FATAL_ERROR_INTERNAL);
    }
Exit:
    CURVE_FREE(E);
    return retVal;
}
```

10.2.12.3.12 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_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

```
LIB_EXPORT TPM_RC
CryptEccCommitCompute(
    TPMS_ECC_POINT          *K,             // OUT: [d]B or [r]Q
    TPMS_ECC_POINT          *L,             // OUT: [r]B
    TPMS_ECC_POINT          *E,             // OUT: [r]M
    TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
    TPMS_ECC_POINT          *M,             // IN: M (optional)
    TPMS_ECC_POINT          *B,             // IN: B (optional)
    TPM2B_ECC_PARAMETER     *d,             // IN: d (optional)
    TPM2B_ECC_PARAMETER     *r              // IN: the computed r value (required)
    )
{
    CURVE_INITIALIZED(curve, curveId);  // Normally initialize E as the curve, but
                                        // E means something else in this function
    ECC_INITIALIZED(bnR, r);
    TPM_RC                retVal = TPM_RC_SUCCESS;
//
    // Validate that the required parameters are provided.
    // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
    // E := [r]Q if both M and B are NULL.
    pAssert(r != NULL && E != NULL);

    // Initialize the output points in case they are not computed
    ClearPoint2B(K);
    ClearPoint2B(L);
    ClearPoint2B(E);

    // Sizes of the r parameter may not be zero
    pAssert(r->t.size > 0);

    // If B is provided, compute K=[d]B and L=[r]B
    if(B != NULL)
```



```
705         {
706             ECC_INITIALIZED(bnD, d);
707             POINT_INITIALIZED(pB, B);
708             POINT(pK);
709             POINT(pL);
710     //
711             pAssert(d != NULL && K != NULL && L != NULL);
712
713             if(!BnIsOnCurve(pB, AccessCurveData(curve)))
714                 ERROR_RETURN(TPM_RC_VALUE);
715              // do the math for K = [d]B
716             if((retVal = BnPointMult(pK, pB, bnD, NULL, NULL, curve)) != TPM_RC_SUCCESS)
717                 goto Exit;
718             // Convert BN K to TPM2B K
719             BnPointTo2B(K, pK, curve);
720             //  compute L= [r]B after checking for cancel
721             if(_plat__IsCanceled())
722                 ERROR_RETURN(TPM_RC_CANCELED);
723             // compute L = [r]B
724             if(!BnIsValidPrivateEcc(bnR, curve))
725                 ERROR_RETURN(TPM_RC_VALUE);
726             if((retVal = BnPointMult(pL, pB, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
727                 goto Exit;
728             // Convert BN L to TPM2B L
729             BnPointTo2B(L, pL, curve);
730         }
731         if((M != NULL) || (B == NULL))
732         {
733             POINT_INITIALIZED(pM, M);
734             POINT(pE);
735     //
736             // Make sure that a place was provided for the result
737             pAssert(E != NULL);
738
739             // if this is the third point multiply, check for cancel first
740             if((B != NULL) && _plat__IsCanceled())
741                 ERROR_RETURN(TPM_RC_CANCELED);
742
743             // If M provided, then pM will not be NULL and will compute E = [r]M.
744             // However, if M was not provided, then pM will be NULL and E = [r]G
745             // will be computed
746             if((retVal = BnPointMult(pE, pM, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
747                 goto Exit;
748             // Convert E to 2B format
749             BnPointTo2B(E, pE, curve);
750         }
751     Exit:
752         CURVE_FREE(curve);
753         return retVal;
754     }
755     #endif  // 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   #if     ALG_SHA1
6   HASH_DEF_TEMPLATE(SHA1);
7   #endif
8   #if     ALG_SHA256
9   HASH_DEF_TEMPLATE(SHA256);
10  #endif
11  #if     ALG_SHA384
12  HASH_DEF_TEMPLATE(SHA384);
13  #endif
14  #if     ALG_SHA512
15  HASH_DEF_TEMPLATE(SHA512);
16  #endif
17  HASH_DEF nullDef = {{0}};
```

10.2.13.3 Obligatory Initialization Functions

10.2.13.3.1 CryptHashInit()

This function is called by _TPM_Init() do perform the initialization operations for the library.

```
18  BOOL
19  CryptHashInit(
20      void
21      )
22  {
23      LibHashInit();
24      return TRUE;
25  }
```

10.2.13.3.2 CryptHashStartup()

This function is called by TPM2_Startup() in case there is work to do at startup. Currently, this is a placeholder.

```
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  PHASH_DEF
34  CryptGetHashDef(
35      TPM_ALG_ID        hashAlg
36      )
37  {
38      PHASH_DEF        retVal;
39      switch(hashAlg)
40      {
41  #if     ALG_SHA1
42          case ALG_SHA1_VALUE:
43              return &SHA1_Def;
44              break;
45  #endif
46  #if     ALG_SHA256
47          case ALG_SHA256_VALUE:
48              retVal = &SHA256_Def;
49              break;
50  #endif
51  #if     ALG_SHA384
52          case ALG_SHA384_VALUE:
53              retVal = &SHA384_Def;
54              break;
55  #endif
56  #if     ALG_SHA512
57          case ALG_SHA512_VALUE:
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 CryptHashIsValidAlg()

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(1)	*hashAlg* is a valid, implemented hash on this TPM
FALSE(0)	*hashAlg* is not valid for this TPM

```
67  BOOL
68  CryptHashIsValidAlg(
69      TPM_ALG_ID        hashAlg,
70      BOOL              flag
```

```
71      )
72  {
73      switch(hashAlg)
74      {
75  #if     ALG_SHA1
76          case ALG_SHA1_VALUE:
77  #endif
78  #if     ALG_SHA256
79          case ALG_SHA256_VALUE:
80  #endif
81  #if     ALG_SHA384
82          case ALG_SHA384_VALUE:
83  #endif
84  #if     ALG_SHA512
85          case ALG_SHA512_VALUE:
86  #endif
87  #if     ALG_SM3_256
88          case ALG_SHA256_VALUE:
89  #endif
90              return TRUE;
91              break;
92          case ALG_NULL_VALUE:
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     // 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
ALG_xxx()	a hash algorithm
ALG_NULL	this can be used as a stop value

```
LIB_EXPORT TPM_ALG_ID
CryptHashGetAlgByIndex(
    UINT32          index           // IN: the index
    )
{
    if(index >= HASH_COUNT)
        return TPM_ALG_NULL;
    return g_hashData[index].alg;
}
```

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

```
LIB_EXPORT UINT16
CryptHashGetDigestSize(
    TPM_ALG_ID      hashAlg         // IN: hash algorithm to look up
    )
{
    return CryptGetHashDef(hashAlg)->digestSize;
}
```

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

```
LIB_EXPORT UINT16
CryptHashGetBlockSize(
    TPM_ALG_ID      hashAlg         // IN: hash algorithm to look up
    )
{
    return CryptGetHashDef(hashAlg)->blockSize;
}
```

10.2.13.4.8 CryptHashGetDer

This function returns a pointer to the DER string for the algorithm and indicates its size.

```
LIB_EXPORT UINT16
CryptHashGetDer(
    TPM_ALG_ID      hashAlg,        // IN: the algorithm to look up
    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

10.2.13.5.1 CryptHashCopyState

This function is used to **clone** a HASH_STATE.

```
159 LIB_EXPORT void
160 CryptHashCopyState(
161     HASH_STATE          *out,           // OUT: destination of the state
162     const HASH_STATE    *in             // IN: source of the state
163     )
164 {
165     pAssert(out->type == in->type);
166     out->hashAlg = in->hashAlg;
167     out->def = in->def;
168     if(in->hashAlg != TPM_ALG_NULL)
169     {
170         // Just verify that the hashAlg makes sense (is implemented)
171         CryptGetHashDef(in->hashAlg);
172         // ... and copy.
173          HASH_STATE_COPY(out, in);
174     }
175     if(in->type == HASH_STATE_HMAC)
176     {
177         const HMAC_STATE    *hIn = (HMAC_STATE *)in;
178         HMAC_STATE          *hOut = (HMAC_STATE *)out;
179         hOut->hmacKey = hIn->hmacKey;
180     }
181     return;
182 }
```

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.

```
183 void
184 CryptHashExportState(
185     PCHASH_STATE         internalFmt,   // IN: the hash state formatted for use by
186                                         //     library
```



```
187         PEXPORT_HASH_STATE   externalFmt    // OUT: the exported hash state
188         )
189     {
190         BYTE                    *outBuf = (BYTE *)externalFmt;
191     //
192         cAssert(sizeof(HASH_STATE) <= sizeof(EXPORT_HASH_STATE));
193         // the following #define is used to move data from an aligned internal data
194         // structure to a byte buffer (external format data.
195     #define CopyToOffset(value)                                                    \
196             memcpy(&outBuf[offsetof(HASH_STATE,value)], &internalFmt->value,       \
197                     sizeof(internalFmt->value))
198         // Copy the hashAlg
199         CopyToOffset(hashAlg);
200         CopyToOffset(type);
201     #ifdef HASH_STATE_SMAC
202         if(internalFmt->type == HASH_STATE_SMAC)
203         {
204             memcpy(outBuf, internalFmt, sizeof(HASH_STATE));
205             return;
206
207         }
208     #endif
209         if(internalFmt->type == HASH_STATE_HMAC)
210         {
211             HMAC_STATE              *from = (HMAC_STATE *)internalFmt;
212             memcpy(&outBuf[offsetof(HMAC_STATE, hmacKey)], &from->hmacKey,
213                    sizeof(from->hmacKey));
214         }
215         if(internalFmt->hashAlg != TPM_ALG_NULL)
216              HASH_STATE_EXPORT(externalFmt, internalFmt);
217     }
```

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.

```
218     void
219     CryptHashImportState(
220         PHASH_STATE          internalFmt,   // OUT: the hash state formatted for use by
221                                             //      the library
222         PCEXPORT_HASH_STATE  externalFmt    // IN: the exported hash state
223         )
224     {
225         BYTE                    *inBuf = (BYTE *)externalFmt;
226     //
227     #define CopyFromOffset(value)                                                  \
228         memcpy(&internalFmt->value, &inBuf[offsetof(HASH_STATE,value)],            \
229                sizeof(internalFmt->value))
230
231         // Copy the hashAlg of the byte-aligned input structure to the structure-aligned
232         // internal structure.
233         CopyFromOffset(hashAlg);
234         CopyFromOffset(type);
235         if(internalFmt->hashAlg != TPM_ALG_NULL)
236         {
237     #ifdef HASH_STATE_SMAC
238             if(internalFmt->type == HASH_STATE_SMAC)
239             {
240                 memcpy(internalFmt, inBuf, sizeof(HASH_STATE));
241                 return;
242             }
243     #endif
244             internalFmt->def = CryptGetHashDef(internalFmt->hashAlg);
```

```
245         HASH_STATE_IMPORT(internalFmt, inBuf);
246         if(internalFmt->type == HASH_STATE_HMAC)
247         {
248             HMAC_STATE              *to = (HMAC_STATE *)internalFmt;
249             memcpy(&to->hmacKey, &inBuf[offsetof(HMAC_STATE, hmacKey)],
250                    sizeof(to->hmacKey));
251         }
252     }
253 }
```

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.

```
254 static UINT16
255 HashEnd(
256     PHASH_STATE      hashState,      // IN: the hash state
257     UINT32           dOutSize,       // IN: the size of receive buffer
258     PBYTE            dOut            // OUT: the receive buffer
259     )
260 {
261     BYTE                temp[MAX_DIGEST_SIZE];
262     if((hashState->hashAlg == TPM_ALG_NULL)
263        || (hashState->type != HASH_STATE_HASH))
264         dOutSize = 0;
265     if(dOutSize > 0)
266     {
267         hashState->def = CryptGetHashDef(hashState->hashAlg);
268         // Set the final size
269         dOutSize = MIN(dOutSize, hashState->def->digestSize);
270         // Complete into the temp buffer and then copy
271         HASH_END(hashState, temp);
272         // Don't want any other functions calling the HASH_END method
273         // directly.
274 #undef HASH_END
275         memcpy(dOut, &temp, dOutSize);
276     }
277     hashState->type = HASH_STATE_EMPTY;
278     return (UINT16)dOutSize;
279 }
```

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

```
LIB_EXPORT UINT16
CryptHashStart(
    PHASH_STATE      hashState,     // OUT: the running hash state
    TPM_ALG_ID       hashAlg        // IN: hash algorithm
    )
{
    UINT16               retVal;

    TEST(hashAlg);

    hashState->hashAlg = hashAlg;
    if(hashAlg == TPM_ALG_NULL)
    {
        retVal = 0;
    }
    else
    {
        hashState->def = CryptGetHashDef(hashAlg);
        HASH_START(hashState);
        retVal = hashState->def->digestSize;
    }
#undef HASH_START
    hashState->type = HASH_STATE_HASH;
    return retVal;
}
```

10.2.13.6.3 CryptDigestUpdate()

Add data to a hash or HMAC, SMAC stack.

```
void
CryptDigestUpdate(
    PHASH_STATE      hashState,     // IN: the hash context information
    UINT32           dataSize,      // IN: the size of data to be added
    const BYTE      *data           // IN: data to be hashed
    )
{
    if(hashState->hashAlg != TPM_ALG_NULL)
    {

        if((hashState->type == HASH_STATE_HASH)
           || (hashState->type == HASH_STATE_HMAC))
            HASH_DATA(hashState, dataSize, (BYTE *)data);
#if SMAC_IMPLEMENTED
        else if(hashState->type == HASH_STATE_SMAC)
            (hashState->state.smac.smacMethods.data)(&hashState->state.smac.state,
                                                     dataSize, data);
#endif // SMAC_IMPLEMENTED
        else
            FAIL(FATAL_ERROR_INTERNAL);
    }
    return;
}
```

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

```
LIB_EXPORT UINT16
CryptHashEnd(
    PHASH_STATE      hashState,      // IN: the state of hash stack
    UINT32           dOutSize,       // IN: size of digest buffer
    BYTE            *dOut            // OUT: hash digest
    )
{
    pAssert(hashState->type == HASH_STATE_HASH);
    return HashEnd(hashState, dOutSize, dOut);
}
```

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*

```
LIB_EXPORT UINT16
CryptHashBlock(
    TPM_ALG_ID       hashAlg,        // IN: The hash algorithm
    UINT32           dataSize,       // IN: size of buffer to hash
    const BYTE      *data,           // IN: the buffer to hash
    UINT32           dOutSize,       // IN: size of the digest buffer
    BYTE            *dOut            // OUT: digest buffer
    )
{
    HASH_STATE          state;
    CryptHashStart(&state, hashAlg);
    CryptDigestUpdate(&state, dataSize, data);
    return HashEnd(&state, dOutSize, dOut);
}
```

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.

```
LIB_EXPORT void
CryptDigestUpdate2B(
    PHASH_STATE      state,          // IN: the digest state
    const TPM2B     *bIn             // IN: 2B containing the data
    )
{
```



```
358         // Only compute the digest if a pointer to the 2B is provided.
359         // In CryptDigestUpdate(), if size is zero or buffer is NULL, then no change
360         // to the digest occurs. This function should not provide a buffer if bIn is
361         // not provided.
362         pAssert(bIn != NULL);
363         CryptDigestUpdate(state, bIn->size, bIn->buffer);
364         return;
365     }
```

10.2.13.6.7 CryptHashEnd2B()

This function is the same as CryptCompleteHash() but the digest is placed in a TPM2B. This is the most common use and this is provided for specification clarity. 'digest.size' should be set to indicate the number of bytes to place in the buffer

Return Value	Meaning
>=0	the number of bytes placed in 'digest.buffer'

```
366     LIB_EXPORT UINT16
367     CryptHashEnd2B(
368         PHASH_STATE      state,          // IN: the hash state
369         P2B              digest          // IN: the size of the buffer Out: requested
370                                          //     number of bytes
371         )
372     {
373         return CryptHashEnd(state, digest->size, digest->buffer);
374     }
```

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().

```
375     LIB_EXPORT void
376     CryptDigestUpdateInt(
377         void            *state,          // IN: the state of hash stack
378         UINT32           intSize,        // IN: the size of 'intValue' in bytes
379         UINT64           intValue        // IN: integer value to be hashed
380         )
381     {
382     #if LITTLE_ENDIAN_TPM
383         intValue = REVERSE_ENDIAN_64(intValue);
384     #endif
385         CryptDigestUpdate(state, intSize, &((BYTE *)&intValue)[8 - intSize]);
386     }
```

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)

```
387  LIB_EXPORT UINT16
388  CryptHmacStart(
389      PHMAC_STATE      state,         // IN/OUT: the state buffer
390      TPM_ALG_ID       hashAlg,       // IN: the algorithm to use
391      UINT16           keySize,       // IN: the size of the HMAC key
392      const BYTE      *key            // IN: the HMAC key
393      )
394  {
395      PHASH_DEF                hashDef;
396      BYTE *                   pb;
397      UINT32                   i;
398  //
399      hashDef = CryptGetHashDef(hashAlg);
400      if(hashDef->digestSize != 0)
401      {
402      // If the HMAC key is larger than the hash block size, it has to be reduced
403      // to fit. The reduction is a digest of the hashKey.
404          if(keySize > hashDef->blockSize)
405          {
406              // if the key is too big, reduce it to a digest of itself
407              state->hmacKey.t.size = CryptHashBlock(hashAlg, keySize, key,
408                                                     hashDef->digestSize,
409                                                     state->hmacKey.t.buffer);
410          }
411          else
412          {
413              memcpy(state->hmacKey.t.buffer, key, keySize);
414              state->hmacKey.t.size = keySize;
415          }
416          // XOR the key with iPad (0x36)
417          pb = state->hmacKey.t.buffer;
418          for(i = state->hmacKey.t.size; i > 0; i--)
419              *pb++ ^= 0x36;
420
421          // if the keySize is smaller than a block, fill the rest with 0x36
422          for(i = hashDef->blockSize - state->hmacKey.t.size; i > 0; i--)
423              *pb++ = 0x36;
424
425          // Increase the oPadSize to a full block
426          state->hmacKey.t.size = hashDef->blockSize;
427
428          // Start a new hash with the HMAC key
429          // This will go in the caller's state structure and may be a sequence or not
430          CryptHashStart((PHASH_STATE)state, hashAlg);
431          CryptDigestUpdate((PHASH_STATE)state, state->hmacKey.t.size,
432                            state->hmacKey.t.buffer);
433          // XOR the key block with 0x5c ^ 0x36
434          for(pb = state->hmacKey.t.buffer, i = hashDef->blockSize; i > 0; i--)
435              *pb++ ^= (0x5c ^ 0x36);
436      }
437      // Set the hash algorithm
438      state->hashState.hashAlg = hashAlg;
439      // Set the hash state type
440      state->hashState.type = HASH_STATE_HMAC;
441
442      return hashDef->digestSize;
443  }
```


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)

```
444  LIB_EXPORT UINT16
445  CryptHmacEnd(
446      PHMAC_STATE      state,          // IN: the hash state buffer
447      UINT32           dOutSize,       // IN: size of digest buffer
448      BYTE            *dOut            // OUT: hash digest
449      )
450  {
451      BYTE                 temp[MAX_DIGEST_SIZE];
452      PHASH_STATE          hState = (PHASH_STATE)&state->hashState;
453
454  #if SMAC_IMPLEMENTED
455      if(hState->type == HASH_STATE_SMAC)
456          return (state->hashState.state.smac.smacMethods.end)
457                      (&state->hashState.state.smac.state,
458                       dOutSize,
459                       dOut);
460  #endif
461      pAssert(hState->type == HASH_STATE_HMAC);
462      hState->def = CryptGetHashDef(hState->hashAlg);
463      // Change the state type for completion processing
464      hState->type = HASH_STATE_HASH;
465      if(hState->hashAlg == TPM_ALG_NULL)
466          dOutSize = 0;
467      else
468      {
469
470      // Complete the current hash
471          HashEnd(hState, hState->def->digestSize, temp);
472          // Do another hash starting with the oPad
473          CryptHashStart(hState, hState->hashAlg);
474          CryptDigestUpdate(hState, state->hmacKey.t.size, state->hmacKey.t.buffer);
475          CryptDigestUpdate(hState, hState->def->digestSize, temp);
476      }
477      return HashEnd(hState, dOutSize, dOut);
478  }
```

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

```
479  LIB_EXPORT UINT16
480  CryptHmacStart2B(
```

```
481     PHMAC_STATE      hmacState,     // OUT: the state of HMAC stack. It will be used
482                                     //      in HMAC update and completion
483     TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
484     P2B              key            // IN: HMAC key
485     )
486 {
487     return CryptHmacStart(hmacState, hashAlg, key->size, key->buffer);
488 }
```

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*

```
489 LIB_EXPORT UINT16
490 CryptHmacEnd2B(
491     PHMAC_STATE      hmacState,     // IN: the state of HMAC stack
492     P2B              digest         // OUT: HMAC
493     )
494 {
495     return CryptHmacEnd(hmacState, digest->size, digest->buffer);
496 }
```

10.2.13.8 Mask and Key Generation Functions

10.2.13.8.1 CryptMGF1()

This function performs MGF1 using the selected hash. MGF1 is T(n) = T(n-1) || H(seed || counter). This function returns the length of the mask produced which could be zero if the digest algorithm is not supported

Return Value	Meaning
0	hash algorithm was TPM_ALG_NULL
> 0	should be the same as *mSize*

```
497 LIB_EXPORT UINT16
498 CryptMGF1(
499     UINT32           mSize,         // IN: length of the mask to be produced
500     BYTE            *mask,          // OUT: buffer to receive the mask
501     TPM_ALG_ID       hashAlg,       // IN: hash to use
502     UINT32           seedSize,      // IN: size of the seed
503     BYTE            *seed           // IN: seed size
504     )
505 {
506     HASH_STATE           hashState;
507     PHASH_DEF            hDef = CryptGetHashDef(hashAlg);
508     UINT32               remaining;
509     UINT32               counter = 0;
510     BYTE                 swappedCounter[4];
511
512     // If there is no digest to compute return
513     if((hashAlg == TPM_ALG_NULL) || (mSize == 0))
514         return 0;
515
516     for(remaining = mSize; ; remaining -= hDef->digestSize)
```



```
517         {
518             // Because the system may be either Endian...
519             UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
520
521             // Start the hash and include the seed and counter
522             CryptHashStart(&hashState, hashAlg);
523             CryptDigestUpdate(&hashState, seedSize, seed);
524             CryptDigestUpdate(&hashState, 4, swappedCounter);
525
526             // Handling the completion depends on how much space remains in the mask
527             // buffer. If it can hold the entire digest, put it there. If not
528             // put the digest in a temp buffer and only copy the amount that
529             // will fit into the mask buffer.
530             HashEnd(&hashState, remaining, mask);
531             if(remaining <= hDef->digestSize)
532                 break;
533             mask = &mask[hDef->digestSize];
534             counter++;
535         }
536         return (UINT16)mSize;
537     }
```

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.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
538 LIB_EXPORT UINT16
539 CryptKDFa(
540     TPM_ALG_ID       hashAlg,        // IN: hash algorithm used in HMAC
541     const TPM2B     *key,            // IN: HMAC key
542     const TPM2B     *label,          // IN: a label for the KDF
543     const TPM2B     *contextU,       // IN: context U
544     const TPM2B     *contextV,       // IN: context V
545     UINT32           sizeInBits,     // IN: size of generated key in bits
546     BYTE            *keyStream,      // OUT: key buffer
547     UINT32          *counterInOut,   // IN/OUT: caller may provide the iteration
548                                      //     counter for incremental operations to
549                                      //     avoid large intermediate buffers.
550     UINT16           blocks          // IN: If non-zero, this is the maximum number
551                                      //     of blocks to be returned, regardless
552                                      //     of sizeInBits
553     )
554 {
555     UINT32                  counter = 0;        // counter value
556     INT16                   bytes;              // number of bytes to produce
```

```
557     UINT16                  generated;         // number of bytes generated
558     BYTE                   *stream = keyStream;
559     HMAC_STATE              hState;
560     UINT16                  digestSize = CryptHashGetDigestSize(hashAlg);
561
562     pAssert(key != NULL && keyStream != NULL);
563
564     TEST(TPM_ALG_KDF1_SP800_108);
565
566     if(digestSize == 0)
567         return 0;
568
569     if(counterInOut != NULL)
570         counter = *counterInOut;
571
572     // If the size of the request is larger than the numbers will handle,
573     // it is a fatal error.
574     pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);
575
576     // The number of bytes to be generated is the smaller of the sizeInBits bytes or
577     // the number of requested blocks. The number of blocks is the smaller of the
578     // number requested or the number allowed by sizeInBits. A partial block is
579     // a full block.
580     bytes = (blocks > 0) ? blocks * digestSize : (UINT16)BITS_TO_BYTES(sizeInBits);
581     generated = bytes;
582
583     // Generate required bytes
584     for(; bytes > 0; bytes -= digestSize)
585     {
586         counter++;
587         // Start HMAC
588         if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
589             return 0;
590         // Adding counter
591         CryptDigestUpdateInt(&hState.hashState, 4, counter);
592
593         // Adding label
594         if(label != NULL)
595             HASH_DATA(&hState.hashState, label->size, (BYTE *)label->buffer);
596         // Add a null. SP108 is not very clear about when the 0 is needed but to
597         // make this like the previous version that did not add an 0x00 after
598         // a null-terminated string, this version will only add a null byte
599         // if the label parameter did not end in a null byte, or if no label
600         // is present.
601         if((label == NULL)
602            || (label->size == 0)
603            || (label->buffer[label->size - 1] != 0))
604             CryptDigestUpdateInt(&hState.hashState, 1, 0);
605         // Adding contextU
606         if(contextU != NULL)
607             HASH_DATA(&hState.hashState, contextU->size, contextU->buffer);
608         // Adding contextV
609         if(contextV != NULL)
610             HASH_DATA(&hState.hashState, contextV->size, contextV->buffer);
611         // Adding size in bits
612         CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);
613
614         // Complete and put the data in the buffer
615         CryptHmacEnd(&hState, bytes, stream);
616         stream = &stream[digestSize];
617     }
618
619     // Masking in the KDF is disabled. If the calling function wants something
620     // less than even number of bytes, then the caller should do the masking
621     // because there is no universal way to do it here
622     if(counterInOut != NULL)
```



```
        *counterInOut = counter;
    return generated;
}
```

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.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
LIB_EXPORT UINT16
CryptKDFe(
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
    TPM2B           *Z,             // IN: Z
    const TPM2B     *label,         // IN: a label value for the KDF
    TPM2B           *partyUInfo,    // IN: PartyUInfo
    TPM2B           *partyVInfo,    // IN: PartyVInfo
    UINT32           sizeInBits,    // IN: size of generated key in bits
    BYTE            *keyStream      // OUT: key buffer
    )
{
    HASH_STATE       hashState;
    PHASH_DEF        hashDef = CryptGetHashDef(hashAlg);

    UINT32           counter = 0;       // counter value
    UINT16           hLen;
    BYTE            *stream = keyStream;
    INT16            bytes;             // number of bytes to generate

    pAssert(keyStream != NULL && Z != NULL && ((sizeInBits + 7) / 8) < INT16_MAX);
//
    hLen = hashDef->digestSize;
    bytes = (INT16)((sizeInBits + 7) / 8);
    if(hashAlg == TPM_ALG_NULL || bytes == 0)
        return 0;

    // Generate required bytes
    //The inner loop of that KDF uses:
    //  Hash[i] := H(counter | Z | OtherInfo) (5)
    // Where:
    //  Hash[i]   the hash generated on the i-th iteration of the loop.
    //  H()     an approved hash function
    //  counter a 32-bit counter that is initialized to 1 and incremented
    //          on each iteration
    //  Z       the X coordinate of the product of a public ECC key and a
    //          different private ECC key.
    //  OtherInfo   a collection of qualifying data for the KDF defined below.
    //  In this specification, OtherInfo will be constructed by:
    //      OtherInfo := Use | PartyUInfo  | PartyVInfo
    for(; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
    {
        if(bytes < hLen)
            hLen = bytes;
        counter++;
        // Do the hash
```

```
671         CryptHashStart(&hashState, hashAlg);
672         // Add counter
673         CryptDigestUpdateInt(&hashState, 4, counter);
674
675         // Add Z
676         if(Z != NULL)
677             CryptDigestUpdate2B(&hashState, Z);
678         // Add label
679         if(label != NULL)
680             CryptDigestUpdate2B(&hashState, label);
681         // Add a null. SP108 is not very clear about when the 0 is needed but to
682         // make this like the previous version that did not add an 0x00 after
683         // a null-terminated string, this version will only add a null byte
684         // if the label parameter did not end in a null byte, or if no label
685         // is present.
686         if((label == NULL)
687            || (label->size == 0)
688            || (label->buffer[label->size - 1] != 0))
689             CryptDigestUpdateInt(&hashState, 1, 0);
690         // Add PartyUInfo
691         if(partyUInfo != NULL)
692             CryptDigestUpdate2B(&hashState, partyUInfo);
693
694         // Add PartyVInfo
695         if(partyVInfo != NULL)
696             CryptDigestUpdate2B(&hashState, partyVInfo);
697
698         // Compute Hash. hLen was changed to be the smaller of bytes or hLen
699         // at the start of each iteration.
700         CryptHashEnd(&hashState, hLen, stream);
701     }
702
703     // Mask off bits if the required bits is not a multiple of byte size
704     if((sizeInBits % 8) != 0)
705         keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
706
707     return (UINT16)((sizeInBits + 7) / 8);
708 }
```


10.2.14 CryptHashData.c

```
1   #include "Tpm.h"
2   const HASH_INFO   g_hashData[HASH_COUNT + 1] = {
3   #if  ALG_SHA1
4       {TPM_ALG_SHA1,    SHA1_DIGEST_SIZE,   SHA1_BLOCK_SIZE,
5       SHA1_DER_SIZE,   {SHA1_DER}},
6   #endif
7   #if  ALG_SHA256
8       {TPM_ALG_SHA256,    SHA256_DIGEST_SIZE,   SHA256_BLOCK_SIZE,
9       SHA256_DER_SIZE,   {SHA256_DER}},
10  #endif
11  #if  ALG_SHA384
12      {TPM_ALG_SHA384,    SHA384_DIGEST_SIZE,   SHA384_BLOCK_SIZE,
13      SHA384_DER_SIZE,   {SHA384_DER}},
14  #endif
15  #if  ALG_SHA512
16      {TPM_ALG_SHA512,    SHA512_DIGEST_SIZE,   SHA512_BLOCK_SIZE,
17      SHA512_DER_SIZE,   {SHA512_DER}},
18  #endif
19  #if  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      };
```

10.2.15 CryptPrime.c

10.2.15.1 Introduction

This file contains the code for prime validation.

```
1   #include "Tpm.h"
2   #include "CryptPrime_fp.h"
3   //#define CPRI_PRIME
4   //#include "PrimeTable.h"
5   #include "CryptPrimeSieve_fp.h"
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[];
10  extern bigConst            s_CompositeOfSmallPrimes;
```

10.2.15.2 Functions

10.2.15.2.1 Root2()

This finds ceil(sqrt(n)) to use as a stopping point for searching the prime table.

```
11  static uint32_t
12  Root2(
13      uint32_t             n
14      )
15  {
16      int32_t              last = (int32_t)(n >> 2);
17      int32_t              next = (int32_t)(n >> 1);
18      int32_t              diff;
19      int32_t              stop = 10;
20  //
21      // get a starting point
22      for(; next != 0; last >>= 1, next >>= 2);
23      last++;
24      do
25      {
26          next = (last + (n / last)) >> 1;
27          diff = next - last;
28          last = next;
29          if(stop-- == 0)
30              FAIL(FATAL_ERROR_INTERNAL);
31      } while(diff < -1 || diff > 1);
32      if((n / next) > (unsigned)next)
33          next++;
34      pAssert(next != 0);
35      pAssert(((n / next) <= (unsigned)next) && (n / (next + 1) < (unsigned)next));
36      return next;
37  }
```

10.2.15.2.2 IsPrimeInt()

This will do a test of a word of 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  }
```

10.2.15.2.3 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
78      if(BnIsEven(prime))
79          return FALSE;
80      if(BnUnsignedCmpWord(prime, s_LastPrimeInTable) <= 0)
81      {
82          crypt_uword_t       temp = prime->d[0] >> 1;
83          return ((s_PrimeTable[temp >> 3] >> (temp & 7)) & 1);
84      }
85      {
86          BN_VAR(n, LARGEST_NUMBER_BITS);
87          BnGcd(n, prime, s_CompositeOfSmallPrimes);
88          if(!BnEqualWord(n, 1))
89              return FALSE;
90      }
91      return MillerRabin(prime, rand);
92  }
```

10.2.15.2.4 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.

```
93  UINT32
```

```
94  MillerRabinRounds(
95      UINT32          bits            // IN: Number of bits in the RSA prime
96      )
97  {
98      if(bits < 511) return 8;     // don't really expect this
99      if(bits < 1536) return 5;    // for 512 and 1K primes
100     return 4;                    // for 3K public modulus and greater
101 }
```

10.2.15.2.5 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(1)	probably prime
FALSE(0)	composite

```
102 BOOL
103 MillerRabin(
104     bigNum          bnW,
105     RAND_STATE      *rand
106     )
107 {
108     BN_MAX(bnWm1);
109     BN_PRIME(bnM);
110     BN_PRIME(bnB);
111     BN_PRIME(bnZ);
112     BOOL            ret = FALSE;    // Assumed composite for easy exit
113     unsigned int    a;
114     unsigned int    j;
115     int             wLen;
116     int             i;
117     int             iterations = MillerRabinRounds(BnSizeInBits(bnW));
118 //
119    INSTRUMENT_INC(MillerRabinTrials[PrimeIndex]);
120
121     pAssert(bnW->size > 1);
122     // Let a be the largest integer such that 2^a divides w1.
123     BnSubWord(bnWm1, bnW, 1);
124     pAssert(bnWm1->size != 0);
125
126     // Since w is odd (w-1) is even so start at bit number 1 rather than 0
127     // Get the number of bits in bnWm1 so that it doesn't have to be recomputed
128     // on each iteration.
129     i = bnWm1->size * RADIX_BITS;
130   // Now find the largest power of 2 that divides w1
131     for(a = 1;
132     (a < (bnWm1->size * RADIX_BITS)) &&
133         (BnTestBit(bnWm1, a) == 0);
134         a++);
135  // 2. m = (w1) / 2^a
136         BnShiftRight(bnM, bnWm1, a);
137       // 3. wlen = len (w).
138     wLen = BnSizeInBits(bnW);
139   // 4. For i = 1 to iterations do
140     for(i = 0; i < iterations; i++)
141     {
142       // 4.1 Obtain a string b of wlen bits from an RBG.
143       // Ensure that 1 < b < w1.
144         do
145         {
```

```
146                 BnGetRandomBits(bnB, wLen, rand);
147                 // 4.2 If ((b <= 1) or (b >= w1)), then go to step 4.1.
148             } while((BnUnsignedCmpWord(bnB, 1) <= 0)
149                     || (BnUnsignedCmp(bnB, bnWm1) >= 0));
150         // 4.3 z = b^m mod w.
151         // if ModExp fails, then say this is not
152         // prime and bail out.
153             BnModExp(bnZ, bnB, bnM, bnW);
154
155          // 4.4 If ((z == 1) or (z = w == 1)), then go to step 4.7.
156             if((BnUnsignedCmpWord(bnZ, 1) == 0)
157                || (BnUnsignedCmp(bnZ, bnWm1) == 0))
158                 goto step4point7;
159          // 4.5 For j = 1 to a  1 do.
160             for(j = 1; j < a; j++)
161             {
162               // 4.5.1 z = z^2 mod w.
163                 BnModMult(bnZ, bnZ, bnZ, bnW);
164               // 4.5.2 If (z = w1), then go to step 4.7.
165                 if(BnUnsignedCmp(bnZ, bnWm1) == 0)
166                     goto step4point7;
167               // 4.5.3 If (z = 1), then go to step 4.6.
168                 if(BnEqualWord(bnZ, 1))
169                     goto step4point6;
170             }
171             // 4.6 Return COMPOSITE.
172     step4point6:
173             INSTRUMENT_INC(failedAtIteration[i]);
174             goto end;
175          // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
176     step4point7:
177             continue;
178         }
179       // 5. Return PROBABLY PRIME
180         ret = TRUE;
181     end:
182         return ret;
183     }
184     #if     ALG_RSA
```

10.2.15.2.6 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.

```
185     TPM_RC
186     RsaCheckPrime(
187         bigNum           prime,
188         UINT32           exponent,
189         RAND_STATE      *rand
190         )
191     {
192     #if !RSA_KEY_SIEVE
193         TPM_RC          retVal = TPM_RC_SUCCESS;
194         UINT32          modE = BnModWord(prime, exponent);
195
196         NOT_REFERENCED(rand);
197
198         if(modE == 0)
```

```
199             // evenly divisible so add two keeping the number odd
200             BnAddWord(prime, prime, 2);
201         // want 0 != (p - 1) mod e
202         // which is 1 != p mod e
203         else if(modE == 1)
204             // subtract 2 keeping number odd and insuring that
205             // 0 != (p - 1) mod e
206             BnSubWord(prime, prime, 2);
207
208         if(BnIsProbablyPrime(prime, rand) == 0)
209             ERROR_RETURN(TPM_RC_VALUE);
210     Exit:
211         return retVal;
212     #else
213         return PrimeSelectWithSieve(prime, exponent, rand);
214     #endif
215     }
```

10.2.15.2.7 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.

```
216     LIB_EXPORT void
217     RsaAdjustPrimeCandidate(
218         bigNum          prime
219         )
220     {
221         UINT16  highBytes;
222         crypt_uword_t       *msw = &prime->d[prime->size - 1];
223     #define MASK (MAX_CRYPT_UWORD >> (RADIX_BITS - 16))
224
225         highBytes = *msw >> (RADIX_BITS - 16);
226         // This is fixed point arithmetic on 16-bit values
227         highBytes = ((UINT32)highBytes * (UINT32)0x4AFB) >> 16;
228         highBytes += 0xB505;
229         *msw = ((crypt_uword_t)(highBytes) << (RADIX_BITS - 16)) + (*msw & MASK);
230         prime->d[0] |= 1;
231     }
```

10.2.15.2.8 BnGeneratePrimeForRSA()

Function to generate a prime of the desired size with the proper attributes for an RSA prime.

```
232     void
233     BnGeneratePrimeForRSA(
234         bigNum          prime,
235         UINT32          bits,
236         UINT32          exponent,
237         RAND_STATE      *rand
238         )
239     {
240         BOOL            found = FALSE;
241     //
242         // Make sure that the prime is large enough
243         pAssert(prime->allocated >= BITS_TO_CRYPT_WORDS(bits));
```



```
244         // Only try to handle specific sizes of keys in order to save overhead
245         pAssert((bits % 32) == 0);
246
247         prime->size = BITS_TO_CRYPT_WORDS(bits);
248
249         while(!found)
250         {
251             DRBG_Generate(rand, (BYTE *)prime->d, (UINT16)BITS_TO_BYTES(bits));
252             RsaAdjustPrimeCandidate(prime);
253             found = RsaCheckPrime(prime, exponent, rand) == TPM_RC_SUCCESS;
254         }
255     }
256     #endif // ALG_RSA
```

10.2.16 CryptPrimeSieve.c

10.2.16.1 Includes and defines

```
#include "Tpm.h"
#if RSA_KEY_SIEVE
#include "CryptPrimeSieve_fp.h"
```

This determines the number of bits in the largest sieve field.

```
#define MAX_FIELD_SIZE  2048
extern const uint32_t      s_LastPrimeInTable;
extern const uint32_t      s_PrimeTableSize;
extern const uint32_t      s_PrimesInTable;
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.

```
const uint32_t      s_PrimeMarkersCount = 6;
const uint32_t      s_PrimeMarkers[] = {
    8167, 17881, 28183, 38891, 49871, 60961 };
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.

```
LIB_EXPORT void
RsaAdjustPrimeLimit(
    uint32_t        requestedPrimes
    )
{

    if(requestedPrimes == 0 || requestedPrimes > s_PrimesInTable)
        requestedPrimes = s_PrimesInTable;
    requestedPrimes = (requestedPrimes - 1) / 1024;
    if(requestedPrimes < s_PrimeMarkersCount)
        primeLimit = s_PrimeMarkers[requestedPrimes];
    else
        primeLimit = s_LastPrimeInTable;
    primeLimit >>= 1;

}
```

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.

```
LIB_EXPORT uint32_t
```



```
30  RsaNextPrime(
31      uint32_t    lastPrime
32      )
33  {
34      if(lastPrime == 0)
35          return 0;
36      lastPrime >>= 1;
37      for(lastPrime += 1; lastPrime <= primeLimit; lastPrime++)
38      {
39          if(((s_PrimeTable[lastPrime >> 3] >> (lastPrime & 0x7)) & 1) == 1)
40              return ((lastPrime << 1) + 1);
41      }
42      return 0;
43  }
```

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.

```
44  const BYTE   seedValues[] = {
45      0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c,
46      0x99, 0x96, 0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96,
47      0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
48      0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
49      0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
50      0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
51      0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
52      0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0x69, 0xc9, 0x34,
53      0x25, 0xda, 0x22, 0x65, 0x99, 0xb4, 0x4c, 0x1b,
54      0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
55      0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6,
56      0x25, 0xd3, 0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2,
57      0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
58      0xd1};
59  #define USE_NIBBLE
60  #ifndef USE_NIBBLE
61  static const BYTE bitsInByte[256] = {
62      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
63      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
64      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
65      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
66      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
67      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
68      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
69      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
70      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
71      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
72      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
73      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
74      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
75      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
76      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
77      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
78      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
79      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
80      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
81      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
82      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
83      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
84      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
85      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
86      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
87      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
88      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
89      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
90      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
```

```
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08
};
#define BitsInByte(x)   bitsInByte[(unsigned char)x]
#else
const BYTE bitsInNibble[16] = {
    0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04};
#define BitsInByte(x)                                          \
            (bitsInNibble[(unsigned char)(x) & 0xf]            \
        +   bitsInNibble[((unsigned char)(x) >> 4) & 0xf])
#endif
```

10.2.16.2.3 BitsInArry()

This function counts the number of bits set in an array of bytes.

```
static int
BitsInArray(
    const unsigned char     *a,             // IN: A pointer to an array of bytes
    unsigned int             aSize          // IN: the number of bytes to sum
    )
{
    int     j = 0;
    for(; aSize; a++, aSize--)
        j += BitsInByte(*a);
    return j;
}
```

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

```
LIB_EXPORT int
FindNthSetBit(
    const UINT16     aSize,         // IN: the size of the array to check
    const BYTE      *a,             // IN: the array to check
    const UINT32     n              // IN, the number of the SET bit
    )
{
    UINT16       i;
    int          retValue;
    UINT32       sum = 0;
    BYTE         sel;

    //find the bit
    for(i = 0; (i < (int)aSize) && (sum < n); i++)
        sum += BitsInByte(a[i]);
    i--;
    // The chosen bit is in the byte that was just accessed
    // Compute the offset to the start of that byte
    retValue = i * 8 - 1;
    sel = a[i];
    // Subtract the bits in the last byte added.
    sum -= BitsInByte(sel);
```



```
137         // Now process the byte, one bit at a time.
138         for(; (sel != 0) && (sum != n); retValue++, sel = sel >> 1)
139             sum += (sel & 1) != 0;
140         return (sum == n) ? retValue : -1;
141     }
142     typedef struct
143     {
144         UINT16      prime;
145         UINT16      count;
146     } SIEVE_MARKS;
147     const SIEVE_MARKS sieveMarks[5] = {
148         {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.

```
149     LIB_EXPORT UINT32
150     PrimeSieve(
151         bigNum          bnN,        // IN/OUT: number to sieve
152         UINT32          fieldSize,  // IN: size of the field area in bytes
153         BYTE           *field       // IN: field
154         )
155     {
156         INT32           i;
157         UINT32          j;
158         UINT32          fieldBits = fieldSize * 8;
159         UINT32          r;
160         BYTE           *pField;
161         INT32           iter;
162         UINT32          adjust;
163         UINT32          mark = 0;
164         UINT32          count = sieveMarks[0].count;
165         UINT32          stop = sieveMarks[0].prime;
166         UINT32          composite;
167         UINT32          pList[8];
168         UINT32          next;
169
170         pAssert(field != NULL && bnN != NULL);
171
172         // If the remainder is odd, then subtracting the value will give an even number,
173         // but we want an odd number, so subtract the 105+rem. Otherwise, just subtract
174         // the even remainder.
175         adjust = BnModWord(bnN, 105);
176         if(adjust & 1)
177             adjust += 105;
178
179         // Adjust the input number so that it points to the first number in a
180         // aligned field.
181         BnSubWord(bnN, bnN, adjust);
182     //    pAssert(BnModWord(bnN, 105) == 0);
183         pField = field;
184         for(i = fieldSize; i >= sizeof(seedValues);
185             pField += sizeof(seedValues), i -= sizeof(seedValues))
```

```
186         {
187             memcpy(pField, seedValues, sizeof(seedValues));
188         }
189         if(i != 0)
190             memcpy(pField, seedValues, i);
191
192         // Cycle through the primes, clearing bits
193         // Have already done 3, 5, and 7
194         iter = 7;
195
196     #define NEXT_PRIME(iter)    (iter = RsaNextPrime(iter))
197         // Get the next N primes where N is determined by the mark in the sieveMarks
198         while((composite = NEXT_PRIME(iter)) != 0)
199         {
200             next = 0;
201             i = count;
202             pList[i--] = composite;
203             for(; i > 0; i--)
204             {
205                 next = NEXT_PRIME(iter);
206                 pList[i] = next;
207                 if(next != 0)
208                     composite *= next;
209             }
210             // Get the remainder when dividing the base field address
211             // by the composite
212             composite = BnModWord(bnN, composite);
213             // 'composite' is divisible by the composite components. for each of the
214             // composite components, divide 'composite'. That remainder (r) is used to
215             // pick a starting point for clearing the array. The stride is equal to the
216             // composite component. Note, the field only contains odd numbers. If the
217             // field were expanded to contain all numbers, then half of the bits would
218             // have already been cleared. We can save the trouble of clearing them a
219             // second time by having a stride of 2*next. Or we can take all of the even
220             // numbers out of the field and use a stride of 'next'
221             for(i = count; i > 0; i--)
222             {
223                 next = pList[i];
224                 if(next == 0)
225                     goto done;
226                 r = composite % next;
227             // these computations deal with the fact that the field starts at some
228             // arbitrary offset within the number space. If the field were all numbers,
229             // then we would have gone through some number of bit clearings before we
230             // got to the start of this range. We don't know how many there were before,
231             // but we can tell from the remainder whether we are on an even or odd
232             // stride when we hit the beginning of the table. If we are on an odd stride
233             // (r & 1), we would start half a stride in (next - r)/2. If we are on an
234             // even stride, we need 1.5 strides (next + r/2) because the table only has
235             // odd numbers. If the remainder happens to be zero, then the start of the
236             // table is on stride so no adjustment is necessary.
237                 if(r & 1)           j = (next - r) / 2;
238                 else if(r == 0)     j = 0;
239                 else                 j = next - r / 2;
240                 for(; j < fieldBits; j += next)
241                     ClearBit(j, field, fieldSize);
242             }
243             if(next >= stop)
244             {
245                 mark++;
246                 count = sieveMarks[mark].count;
247                 stop = sieveMarks[mark].prime;
248             }
249         }
250     done:
251         INSTRUMENT_INC(totalFieldsSieved[PrimeIndex]);
```



```
252         i = BitsInArray(field, fieldSize);
253         INSTRUMENT_ADD(bitsInFieldAfterSieve[PrimeIndex], i);
254         INSTRUMENT_ADD(emptyFieldsSieved[PrimeIndex], (i == 0));
255         return i;
256     }
257     #ifdef SIEVE_DEBUG
258     static uint32_t fieldSize = 210;
```

10.2.16.2.6 SetFieldSize()

Function to set the field size used for prime generation. Used for tuning.

```
259     LIB_EXPORT uint32_t
260     SetFieldSize(
261         uint32_t          newFieldSize
262         )
263     {
264         if(newFieldSize == 0 || newFieldSize > MAX_FIELD_SIZE)
265             fieldSize = MAX_FIELD_SIZE;
266         else
267             fieldSize = newFieldSize;
268         return fieldSize;
269     }
270     #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 the field

```
271     LIB_EXPORT TPM_RC
272     PrimeSelectWithSieve(
273         bigNum            candidate,          // IN/OUT: The candidate to filter
274         UINT32            e,                  // IN: the exponent
275         RAND_STATE       *rand                // IN: the random number generator state
276         )
277     {
278         BYTE              field[MAX_FIELD_SIZE];
279         UINT32            first;
280         UINT32            ones;
281         INT32             chosen;
282         BN_PRIME(test);
283         UINT32            modE;
284     #ifndef SIEVE_DEBUG
285         UINT32            fieldSize = MAX_FIELD_SIZE;
286     #endif
287         UINT32            primeSize;
288     //
289         // Adjust the field size and prime table list to fit the size of the prime
290         // being tested. This is done to try to optimize the trade-off between the
291         // dividing done for sieving and the time for Miller-Rabin. When the size
292         // of the prime is large, the cost of Miller-Rabin is fairly high, as is the
```

```
293         // cost of the sieving. However, the time for Miller-Rabin goes up considerably
294         // faster than the cost of dividing by a number of primes.
295         primeSize = BnSizeInBits(candidate);
296
297         if(primeSize <= 512)
298         {
299             RsaAdjustPrimeLimit(1024); // Use just the first 1024 primes
300         }
301         else if(primeSize <= 1024)
302         {
303             RsaAdjustPrimeLimit(4096); // Use just the first 4K primes
304         }
305         else
306         {
307             RsaAdjustPrimeLimit(0);     // Use all available
308         }
309
310         // Save the low-order word to use as a search generator and make sure that
311         // it has some interesting range to it
312         first = candidate->d[0] | 0x80000000;
313
314         // Sieve the field
315         ones = PrimeSieve(candidate, fieldSize, field);
316         pAssert(ones > 0 && ones < (fieldSize * 8));
317         for(; ones > 0; ones--)
318         {
319             // Decide which bit to look at and find its offset
320             chosen = FindNthSetBit((UINT16)fieldSize, field, ((first % ones) + 1));
321
322             if((chosen < 0) || (chosen >= (INT32)(fieldSize * 8)))
323                 FAIL(FATAL_ERROR_INTERNAL);
324
325             // Set this as the trial prime
326             BnAddWord(test, candidate, (crypt_uword_t)(chosen * 2));
327
328             // The exponent might not have been one of the tested primes so
329             // make sure that it isn't divisible and make sure that 0 != (p-1) mod e
330             // Note: This is the same as 1 != p mod e
331             modE = BnModWord(test, e);
332             if((modE != 0) && (modE != 1) && MillerRabin(test, rand))
333             {
334                 BnCopy(candidate, test);
335                 return TPM_RC_SUCCESS;
336             }
337             // Clear the bit just tested
338             ClearBit(chosen, field, fieldSize);
339         }
340         // Ran out of bits and couldn't find a prime in this field
341         INSTRUMENT_INC(noPrimeFields[PrimeIndex]);
342         return TPM_RC_NO_RESULT;
343     }
344     #if RSA_INSTRUMENT
345     static char             a[256];
346     char *
347     PrintTuple(
348         UINT32      *i
349         )
350     {
351         sprintf(a, "{%d, %d, %d}", i[0], i[1], i[2]);
352         return a;
353     }
354
355     #define CLEAR_VALUE(x)    memset(x, 0, sizeof(x))
356
357     void
358     RsaSimulationEnd(
```



```
359         void
360         )
361     {
362         int         i;
363         UINT32      averages[3];
364         UINT32      nonFirst = 0;
365         if((PrimeCounts[0] + PrimeCounts[1] + PrimeCounts[2]) != 0)
366         {
367             printf("Primes generated = %s\n", PrintTuple(PrimeCounts));
368             printf("Fields sieved = %s\n", PrintTuple(totalFieldsSieved));
369             printf("Fields with no primes = %s\n", PrintTuple(noPrimeFields));
370             printf("Primes checked with Miller-Rabin = %s\n",
371                    PrintTuple(MillerRabinTrials));
372             for(i = 0; i < 3; i++)
373                 averages[i] = (totalFieldsSieved[i]
374                                != 0 ? bitsInFieldAfterSieve[i] / totalFieldsSieved[i]
375                                : 0);
376             printf("Average candidates in field %s\n", PrintTuple(averages));
377             for(i = 1; i < (sizeof(failedAtIteration) / sizeof(failedAtIteration[0]));
378             i++)
379                 nonFirst += failedAtIteration[i];
380             printf("Miller-Rabin failures not in first round = %d\n", nonFirst);
381
382         }
383         CLEAR_VALUE(PrimeCounts);
384         CLEAR_VALUE(totalFieldsSieved);
385         CLEAR_VALUE(noPrimeFields);
386         CLEAR_VALUE(MillerRabinTrials);
387         CLEAR_VALUE(bitsInFieldAfterSieve);
388     }
389
390     LIB_EXPORT void
391     GetSieveStats(
392         uint32_t        *trials,
393         uint32_t        *emptyFields,
394         uint32_t        *averageBits
395         )
396     {
397         uint32_t        totalBits;
398         uint32_t        fields;
399         *trials = MillerRabinTrials[0] + MillerRabinTrials[1] + MillerRabinTrials[2];
400         *emptyFields = noPrimeFields[0] + noPrimeFields[1] + noPrimeFields[2];
401         fields = totalFieldsSieved[0] + totalFieldsSieved[1]
402             + totalFieldsSieved[2];
403         totalBits = bitsInFieldAfterSieve[0] + bitsInFieldAfterSieve[1]
404             + bitsInFieldAfterSieve[2];
405         if(fields != 0)
406             *averageBits = totalBits / fields;
407         else
408             *averageBits = 0;
409         CLEAR_VALUE(PrimeCounts);
410         CLEAR_VALUE(totalFieldsSieved);
411         CLEAR_VALUE(noPrimeFields);
412         CLEAR_VALUE(MillerRabinTrials);
413         CLEAR_VALUE(bitsInFieldAfterSieve);
414
415     }
416     #endif
417
418     #endif // RSA_KEY_SIEVE
419
420     #if !RSA_INSTRUMENT
421     void
422     RsaSimulationEnd(
423         void
424         )
```

```
425    {
426    }
427    #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.

This is the implementation layer of CTR DRGB mechanism as defined in SP800-90A and the functions are organized as closely as practical to the organization in SP800-90A. 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 the 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.

```
#include "Tpm.h"
```

Pull in the test vector definitions and define the space

```
#include    "PRNG_TestVectors.h"
const BYTE DRBG_NistTestVector_Entropy[] = {DRBG_TEST_INITIATE_ENTROPY};
const BYTE DRBG_NistTestVector_GeneratedInterm[] =
                                {DRBG_TEST_GENERATED_INTERM};
const BYTE DRBG_NistTestVector_EntropyReseed[] =
                                {DRBG_TEST_RESEED_ENTROPY};
const BYTE DRBG_NistTestVector_Generated[] = {DRBG_TEST_GENERATED};
```

10.2.17.2 Derivation Functions

10.2.17.2.1 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.

10.2.17.2.2 Derivation Function Defines and Structures

```
9   #define     DF_COUNT (DRBG_KEY_SIZE_WORDS / DRBG_IV_SIZE_WORDS + 1)
10  #if DRBG_KEY_SIZE_BITS != 128 && DRBG_KEY_SIZE_BITS != 256
11  #   error "CryptRand.c only written for AES with 128- or 256-bit keys."
12  #endif
13  typedef struct
14  {
15      DRBG_KEY_SCHEDULE   keySchedule;
16      DRBG_IV             iv[DF_COUNT];
17      DRBG_IV             out1;
18      DRBG_IV             buf;
19      int                 contents;
20  } DF_STATE, *PDF_STATE;
```

10.2.17.2.3 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.

```
21  static void
22  DfCompute(
23      PDF_STATE        dfState
24      )
25  {
26      int              i;
27      int              iv;
28      crypt_uword_t   *pIv;
29      crypt_uword_t    temp[DRBG_IV_SIZE_WORDS] = {0};
30  //
31      for(iv = 0; iv < DF_COUNT; iv++)
32      {
33          pIv = (crypt_uword_t *)&dfState->iv[iv].words[0];
34          for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
35          {
36              temp[i] ^= pIv[i] ^ dfState->buf.words[i];
37          }
38          DRBG_ENCRYPT(&dfState->keySchedule, &temp, pIv);
39      }
40      for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
41          dfState->buf.words[i] = 0;
42      dfState->contents = 0;
43  }
```

10.2.17.2.4 DfStart()

This initializes the output blocks with an encrypted counter value and initializes the key schedule.

```
44  static void
45  DfStart(
46      PDF_STATE        dfState,
47      uint32_t         inputLength
48      )
49  {
50      BYTE            init[8];
51      int             i;
52      UINT32          drbgSeedSize = sizeof(DRBG_SEED);
53
54      const BYTE dfKey[DRBG_KEY_SIZE_BYTES] = {
55          0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
56          0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
```



```
57      #if DRBG_KEY_SIZE_BYTES > 16
58          ,0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
59          0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
60      #endif
61      };
62      memset(dfState, 0, sizeof(DF_STATE));
63      DRBG_ENCRYPT_SETUP(&dfKey[0], DRBG_KEY_SIZE_BITS, &dfState->keySchedule);
64      // Create the first chaining values
65      for(i = 0; i < DF_COUNT; i++)
66          ((BYTE *)&dfState->iv[i])[3] = (BYTE)i;
67      DfCompute(dfState);
68      // initialize the first 64 bits of the IV in a way that doesn't depend
69      // on the size of the words used.
70      UINT32_TO_BYTE_ARRAY(inputLength, init);
71      UINT32_TO_BYTE_ARRAY(drbgSeedSize, &init[4]);
72      memcpy(&dfState->iv[0], init, 8);
73      dfState->contents = 4;
74  }
```

10.2.17.2.5 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().

```
75  static void
76  DfUpdate(
77      PDF_STATE        dfState,
78      int              size,
79      const BYTE      *data
80      )
81  {
82      while(size > 0)
83      {
84          int         toFill = DRBG_IV_SIZE_BYTES - dfState->contents;
85          if(size < toFill)
86              toFill = size;
87          // Copy as many bytes as there are or until the state buffer is full
88          memcpy(&dfState->buf.bytes[dfState->contents], data, toFill);
89          // Reduce the size left by the amount copied
90          size -= toFill;
91          // Advance the data pointer by the amount copied
92          data += toFill;
93          // increase the buffer contents count by the amount copied
94          dfState->contents += toFill;
95          pAssert(dfState->contents <= DRBG_IV_SIZE_BYTES);
96          // If we have a full buffer, do a computation pass.
97          if(dfState->contents == DRBG_IV_SIZE_BYTES)
98              DfCompute(dfState);
99      }
100 }
```

10.2.17.2.6 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.

```
101 static DRBG_SEED *
102 DfEnd(
103     PDF_STATE        dfState
104     )
105 {
```

```
106         // Since DfCompute is always called when a buffer is full, there is always
107         // space in the buffer for the terminator
108         dfState->buf.bytes[dfState->contents++] = 0x80;
109         // If the buffer is not full, pad with zeros
110         while(dfState->contents < DRBG_IV_SIZE_BYTES)
111             dfState->buf.bytes[dfState->contents++] = 0;
112         // Do a final state update
113         DfCompute(dfState);
114         return (DRBG_SEED *)&dfState->iv;
115     }
```

10.2.17.2.7 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();

```
116     static DRBG_SEED *
117     DfBuffer(
118         DRBG_SEED       *output,        // OUT: receives the result
119         int              size,          // IN: size of the buffer to add
120         BYTE            *buf            // IN: address of the buffer
121         )
122     {
123         DF_STATE        dfState;
124         if(size == 0 || buf == NULL)
125             return NULL;
126         // Initialize the derivation function
127         DfStart(&dfState, size);
128         DfUpdate(&dfState, size, buf);
129         DfEnd(&dfState);
130         memcpy(output, &dfState.iv[0], sizeof(DRBG_SEED));
131         return output;
132     }
```

10.2.17.2.8 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(1)	requested entropy returned
FALSE(0)	entropy Failure

```
133     BOOL
134     DRBG_GetEntropy(
135         UINT32           requiredEntropy,   // IN: requested number of bytes of full
136                                             //     entropy
137         BYTE            *entropy            // OUT: buffer to return collected entropy
138         )
139     {
140     #if !USE_DEBUG_RNG
141
142         UINT32       obtainedEntropy;
143         INT32        returnedEntropy;
144
145     // If in debug mode, always use the self-test values for initialization
146         if(IsSelfTest())
147         {
148     #endif
```



```
149         // If doing simulated DRBG, then check to see if the
150         // entropyFailure condition is being tested
151         if(!IsEntropyBad())
152         {
153             // In self-test, the caller should be asking for exactly the seed
154             // size of entropy.
155             pAssert(requiredEntropy == sizeof(DRBG_NistTestVector_Entropy));
156             memcpy(entropy, DRBG_NistTestVector_Entropy,
157                    sizeof(DRBG_NistTestVector_Entropy));
158         }
159 #if !USE_DEBUG_RNG
160     }
161     else if(!IsEntropyBad())
162     {
163         // Collect entropy
164         // Note: In debug mode, the only "entropy" value ever returned
165         // is the value of the self-test vector.
166         for(returnedEntropy = 1, obtainedEntropy = 0;
167             obtainedEntropy < requiredEntropy && !IsEntropyBad();
168             obtainedEntropy += returnedEntropy)
169         {
170             returnedEntropy = _plat__GetEntropy(&entropy[obtainedEntropy],
171                                                 requiredEntropy - obtainedEntropy);
172             if(returnedEntropy <= 0)
173                 SetEntropyBad();
174         }
175     }
176 #endif
177     return !IsEntropyBad();
178 }
```

10.2.17.2.9 IncrementIv()

Used by EncryptDRBG()

```
179 void
180 IncrementIv(
181     DRBG_IV         *iv
182     )
183 {
184     BYTE      *ivP = ((BYTE *)iv) + DRBG_IV_SIZE_BYTES;
185     while((--ivP >= (BYTE *)iv) && ((*ivP = ((*ivP + 1) & 0xFF)) == 0));
186 }
```

10.2.17.2.10 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.

```
187 void
188 EncryptDRBG(
189     BYTE                *dOut,
190     UINT32               dOutBytes,
191     DRBG_KEY_SCHEDULE   *keySchedule,
192     DRBG_IV             *iv,
193     UINT32              *lastValue      // Points to the last output value
194     )
195 {
196 #if FIPS_COMPLIANT
197 // For FIPS compliance, the DRBG has to do a continuous self-test to make sure that
198 // no two consecutive values are the same. This overhead is not incurred if the TPM
199 // is not required to be FIPS compliant
200 //
```

```
201     UINT32            temp[DRBG_IV_SIZE_BYTES / sizeof(UINT32)];
202     int               i;
203     BYTE             *p;
204
205     for(; dOutBytes > 0;)
206     {
207         // Increment the IV before each encryption (this is what makes this
208         // different from normal counter-mode encryption
209         IncrementIv(iv);
210         DRBG_ENCRYPT(keySchedule, iv, temp);
211 // Expect a 16 byte block
212 #if DRBG_IV_SIZE_BITS != 128
213 #error  "Unsuppored IV size in DRBG"
214 #endif
215         if((lastValue[0] == temp[0])
216            && (lastValue[1] == temp[1])
217            && (lastValue[2] == temp[2])
218            && (lastValue[3] == temp[3])
219            )
220             FAIL(FATAL_ERROR_DRBG);
221         lastValue[0] = temp[0];
222         lastValue[1] = temp[1];
223         lastValue[2] = temp[2];
224         lastValue[3] = temp[3];
225         i = MIN(dOutBytes, DRBG_IV_SIZE_BYTES);
226         dOutBytes -= i;
227         for(p = (BYTE *)temp; i > 0; i--)
228             *dOut++ = *p++;
229     }
230 #else // version without continuous self-test
231     NOT_REFERENCED(lastValue);
232     for(; dOutBytes >= DRBG_IV_SIZE_BYTES;
233     dOut = &dOut[DRBG_IV_SIZE_BYTES], dOutBytes -= DRBG_IV_SIZE_BYTES)
234     {
235         // Increment the IV
236         IncrementIv(iv);
237         DRBG_ENCRYPT(keySchedule, iv, dOut);
238     }
239     // If there is a partial, generate into a block-sized
240     // temp buffer and copy to the output.
241     if(dOutBytes != 0)
242     {
243         BYTE        temp[DRBG_IV_SIZE_BYTES];
244         // Increment the IV
245         IncrementIv(iv);
246         DRBG_ENCRYPT(keySchedule, iv, temp);
247         memcpy(dOut, temp, dOutBytes);
248     }
249 #endif
250 }
```

10.2.17.2.11 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.

```
251 void
252 DRBG_Update(
```

```
253     DRBG_STATE          *drbgState,     // IN:OUT state to update
254     DRBG_KEY_SCHEDULE   *keySchedule,   // IN: the key schedule (optional)
255     DRBG_SEED           *providedData   // IN: additional data
256     )
257 {
258     UINT32               i;
259     BYTE                *temp = (BYTE *)&drbgState->seed;
260     DRBG_KEY            *key = pDRBG_KEY(&drbgState->seed);
261     DRBG_IV             *iv = pDRBG_IV(&drbgState->seed);
262     DRBG_KEY_SCHEDULE    localKeySchedule;
263 //
264     pAssert(drbgState->magic == DRBG_MAGIC);
265
266     // If an key schedule was not provided, make one
267     if(keySchedule == NULL)
268     {
269         if(DRBG_ENCRYPT_SETUP((BYTE *)key,
270                               DRBG_KEY_SIZE_BITS, &localKeySchedule) != 0)
271             FAIL(FATAL_ERROR_INTERNAL);
272         keySchedule = &localKeySchedule;
273     }
274     // Encrypt the temp value
275
276     EncryptDRBG(temp, sizeof(DRBG_SEED), keySchedule, iv,
277                 drbgState->lastValue);
278     if(providedData != NULL)
279     {
280         BYTE        *pP = (BYTE *)providedData;
281         for(i = DRBG_SEED_SIZE_BYTES; i != 0; i--)
282             *temp++ ^= *pP++;
283     }
284     // Since temp points to the input key and IV, we are done and
285     // don't need to copy the resulting 'temp' to drbgState->seed
286 }
```

10.2.17.2.12 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(1)	reseed succeeded
FALSE(0)	reseed failed, probably due to the entropy generation

```
287 BOOL
288 DRBG_Reseed(
289     DRBG_STATE          *drbgState,         // IN: the state to update
290     DRBG_SEED           *providedEntropy,   // IN: entropy
291     DRBG_SEED           *additionalData     // IN:
292     )
293 {
294     DRBG_SEED            seed;
295
296     pAssert((drbgState != NULL) && (drbgState->magic == DRBG_MAGIC));
297
298     if(providedEntropy == NULL)
299     {
300         providedEntropy = &seed;
301         if(!DRBG_GetEntropy(sizeof(DRBG_SEED), (BYTE *)providedEntropy))
302             return FALSE;
```

```
303         }
304         if(additionalData != NULL)
305         {
306             unsigned int          i;
307
308             // XOR the provided data into the provided entropy
309             for(i = 0; i < sizeof(DRBG_SEED); i++)
310                 ((BYTE *)providedEntropy)[i] ^= ((BYTE *)additionalData)[i];
311         }
312         DRBG_Update(drbgState, NULL, providedEntropy);
313
314         drbgState->reseedCounter = 1;
315
316         return TRUE;
317     }
```

10.2.17.2.13 DRBG_SelfTest()

This is run when the DRBG is instantiated and at startup

Return Value	Meaning
TRUE(1)	test OK
FALSE(0)	test failed

```
318     BOOL
319     DRBG_SelfTest(
320         void
321         )
322     {
323         BYTE             buf[sizeof(DRBG_NistTestVector_Generated)];
324         DRBG_SEED        seed;
325         UINT32           i;
326         BYTE            *p;
327         DRBG_STATE       testState;
328     //
329         pAssert(!IsSelfTest());
330
331         SetSelfTest();
332         SetDrbgTested();
333         // Do an instantiate
334         if(!DRBG_Instantiate(&testState, 0, NULL))
335             return FALSE;
336     #if DRBG_DEBUG_PRINT
337         dbgDumpMemBlock(pDRBG_KEY(&testState), DRBG_KEY_SIZE_BYTES,
338                         "Key after Instantiate");
339         dbgDumpMemBlock(pDRBG_IV(&testState), DRBG_IV_SIZE_BYTES,
340                         "Value after Instantiate");
341     #endif
342         if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
343             return FALSE;
344     #if DRBG_DEBUG_PRINT
345         dbgDumpMemBlock(pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
346                         "Key after 1st Generate");
347         dbgDumpMemBlock(pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
348                         "Value after 1st Generate");
349     #endif
350         if(memcmp(buf, DRBG_NistTestVector_GeneratedInterm, sizeof(buf)) != 0)
351             return FALSE;
352         memcpy(seed.bytes, DRBG_NistTestVector_EntropyReseed, sizeof(seed));
353         DRBG_Reseed(&testState, &seed, NULL);
354     #if DRBG_DEBUG_PRINT
355         dbgDumpMemBlock((BYTE *)pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
```



```
356                         "Key after 2nd Generate");
357     dbgDumpMemBlock((BYTE *)pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
358                         "Value after 2nd Generate");
359     dbgDumpMemBlock(buf, sizeof(buf), "2nd Generated");
360 #endif
361     if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
362         return FALSE;
363     if(memcmp(buf, DRBG_NistTestVector_Generated, sizeof(buf)) != 0)
364         return FALSE;
365     ClearSelfTest();
366
367     DRBG_Uninstantiate(&testState);
368     for(p = (BYTE *)&testState, i = 0; i < sizeof(DRBG_STATE); i++)
369     {
370         if(*p++)
371             return FALSE;
372     }
373     // Simulate hardware failure to make sure that we get an error when
374     // trying to instantiate
375     SetEntropyBad();
376     if(DRBG_Instantiate(&testState, 0, NULL))
377        return FALSE;
378     ClearEntropyBad();
379
380     return TRUE;
381 }
```

10.2.17.3 Public Interface

10.2.17.3.1 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().

10.2.17.3.2 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	failure of the entropy generator

```
382 LIB_EXPORT TPM_RC
383 CryptRandomStir(
384     UINT16          additionalDataSize,
385     BYTE           *additionalData
386     )
387 {
388 #if !USE_DEBUG_RNG
389     DRBG_SEED        tmpBuf;
390     DRBG_SEED        dfResult;
391 //
392     // All reseed with outside data starts with a buffer full of entropy
393     if(!DRBG_GetEntropy(sizeof(tmpBuf), (BYTE *)&tmpBuf))
394         return TPM_RC_NO_RESULT;
395
396     DRBG_Reseed(&drbgDefault, &tmpBuf,
397                 DfBuffer(&dfResult, additionalDataSize, additionalData));
398     drbgDefault.reseedCounter = 1;
399
400     return TPM_RC_SUCCESS;
```

```
401
402 #else
403     // If doing debug, use the input data as the initial setting for the RNG state
404     // so that the test can be reset at any time.
405     // Note: If this is called with a data size of 0 or less, nothing happens. The
406     // presumption is that, in a debug environment, the caller will have specific
407     // values for initialization, so this check is just a simple way to prevent
408     // inadvertent programming errors from screwing things up. This doesn't use an
409     // pAssert() because the non-debug version of this function will accept these
410     // parameters as meaning that there is no additionalData and only hardware
411     // entropy is used.
412     if((additionalDataSize > 0) && (additionalData != NULL))
413     {
414         memset(drbgDefault.seed.bytes, 0, sizeof(drbgDefault.seed.bytes));
415         memcpy(drbgDefault.seed.bytes, additionalData,
416                MIN(additionalDataSize, sizeof(drbgDefault.seed.bytes)));
417     }
418     drbgDefault.reseedCounter = 1;
419
420     return TPM_RC_SUCCESS;
421 #endif
422 }
```

10.2.17.3.3 CryptRandomGenerate()

Generate a *randomSize* number or random bytes.

```
423 LIB_EXPORT UINT16
424 CryptRandomGenerate(
425     INT32            randomSize,
426     BYTE            *buffer
427     )
428 {
429     if(randomSize > UINT16_MAX)
430         randomSize = UINT16_MAX;
431     return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, (UINT16)randomSize);
432 }
```


10.2.17.3.3.1 DRBG_InstantiateSeededKdf()

This function is used to instantiate a KDF-based RNG. This is used for derivations. This function always returns TRUE.

```
433 LIB_EXPORT BOOL
434 DRBG_InstantiateSeededKdf(
435     KDF_STATE       *state,         // OUT: buffer to hold the state
436     TPM_ALG_ID       hashAlg,       // IN: hash algorithm
437     TPM_ALG_ID       kdf,           // IN: the KDF to use
438     TPM2B           *seed,          // IN: the seed to use
439     const TPM2B     *label,         // IN: a label for the generation process.
440     TPM2B           *context,       // IN: the context value
441     UINT32           limit          // IN: Maximum number of bits from the KDF
442     )
443 {
444     state->magic = KDF_MAGIC;
445     state->limit = limit;
446     state->seed = seed;
447     state->hash = hashAlg;
448     state->kdf = kdf;
449     state->label = label;
450     state->context = context;
451     state->digestSize = CryptHashGetDigestSize(hashAlg);
452     state->counter = 0;
453     state->residual.t.size = 0;
454     return TRUE;
455 }
```

10.2.17.3.3.2 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.

```
456 LIB_EXPORT void
457 DRBG_AdditionalData(
458     DRBG_STATE      *drbgState,     // IN:OUT state to update
459     TPM2B           *additionalData // IN: value to incorporate
460     )
461 {
462     DRBG_SEED        dfResult;
463     if(drbgState->magic == DRBG_MAGIC)
464     {
465         DfBuffer(&dfResult, additionalData->size, additionalData->buffer);
466         DRBG_Reseed(drbgState, &dfResult, NULL);
467     }
468 }
```


10.2.17.3.3.3 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. This function always returns TRUE.

```
469 LIB_EXPORT BOOL
470 DRBG_InstantiateSeeded(
471     DRBG_STATE      *drbgState,     // IN/OUT: buffer to hold the state
472     const TPM2B     *seed,          // IN: the seed to use
473     const TPM2B     *purpose,       // IN: a label for the generation process.
474     const TPM2B     *name,          // IN: name of the object
475     const TPM2B     *additional     // IN: additional data
476     )
477 {
478     DF_STATE         dfState;
479     int              totalInputSize;
480     // DRBG should have been tested, but...
481     if(!IsDrbgTested() && !DRBG_SelfTest())
482         FAIL(FATAL_ERROR_SELF_TEST);
483     // Initialize the DRBG state
484     memset(drbgState, 0, sizeof(DRBG_STATE));
485     drbgState->magic = DRBG_MAGIC;
486
487     // Size all of the values
488     totalInputSize = (seed != NULL) ? seed->size : 0;
489     totalInputSize += (purpose != NULL) ? purpose->size : 0;
490     totalInputSize += (name != NULL) ? name->size : 0;
491     totalInputSize += (additional != NULL) ? additional->size : 0;
492
493     // Initialize the derivation
494     DfStart(&dfState, totalInputSize);
495
496     // Run all the input strings through the derivation function
497     if(seed != NULL)
498         DfUpdate(&dfState, seed->size, seed->buffer);
499     if(purpose != NULL)
500         DfUpdate(&dfState, purpose->size, purpose->buffer);
501     if(name != NULL)
502         DfUpdate(&dfState, name->size, name->buffer);
503     if(additional != NULL)
504         DfUpdate(&dfState, additional->size, additional->buffer);
505
506     // Used the derivation function output as the "entropy" input. This is not
507     // how it is described in SP800-90A but this is the equivalent function
508     DRBG_Reseed(((DRBG_STATE *)drbgState), DfEnd(&dfState), NULL);
509
510     return TRUE;
511 }
```

10.2.17.3.3.4 CryptRandStartup()

This function is called when TPM_Startup() is executed. This function always returns TRUE.

```
512 LIB_EXPORT BOOL
513 CryptRandStartup(
514     void
515     )
516 {
517 #if ! _DRBG_STATE_SAVE
518     // If not saved in NV, re-instantiate on each startup
519     DRBG_Instantiate(&drbgDefault, 0, NULL);
520 #else
521     // If the running state is saved in NV, NV has to be loaded before it can
522     // be updated
523     if(go.drbgState.magic == DRBG_MAGIC)
524         DRBG_Reseed(&go.drbgState, NULL, NULL);
525     else
526         DRBG_Instantiate(&go.drbgState, 0, NULL);
527 #endif
528     return TRUE;
529 }
```

10.2.17.3.3.5 CryptRandInit()

This function is called when _TPM_Init() is being processed.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
530 LIB_EXPORT BOOL
531 CryptRandInit(
532     void
533     )
534 {
535 #if !USE_DEBUG_RNG
536     _plat__GetEntropy(NULL, 0);
537 #endif
538     return DRBG_SelfTest();
539 }
```

10.2.17.3.4 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.

```
540 LIB_EXPORT UINT16
541 DRBG_Generate(
542     RAND_STATE      *state,
543     BYTE            *random,        // OUT: buffer to receive the random values
544     UINT16           randomSize     // IN: the number of bytes to generate
545     )
546 {
547     if(state == NULL)
548         state = (RAND_STATE *)&drbgDefault;
549
550     // If the caller used a KDF state, generate a sequence from the KDF not to
551     // exceed the limit.
552     if(state->kdf.magic == KDF_MAGIC)
553     {
554         KDF_STATE        *kdf = (KDF_STATE *)state;
555         UINT32            counter = (UINT32)kdf->counter;
556         INT32             bytesLeft = randomSize;
557
558         if(random == NULL)
559             return 0;
560         // If the number of bytes to be returned would put the generator
561         // over the limit, then return 0
562         if((((kdf->counter * kdf->digestSize) + randomSize) * 8) > kdf->limit)
563             return 0;
564         // Process partial and full blocks until all requested bytes provided
565         while(bytesLeft > 0)
566         {
567             // If there is any residual data in the buffer, copy it to the output
568             // buffer
569             if(kdf->residual.t.size > 0)
570             {
571                 INT32      size;
572 //
```

```
573                     // Don't use more of the residual than will fit or more than are
574                     // available
575                     size = MIN(kdf->residual.t.size, bytesLeft);
576
577                     // Copy some or all of the residual to the output. The residual is
578                     // at the end of the buffer. The residual might be a full buffer.
579                     MemoryCopy(random,
580                                &kdf->residual.t.buffer
581                                [kdf->digestSize - kdf->residual.t.size], size);
582
583                     // Advance the buffer pointer
584                     random += size;
585
586                     // Reduce the number of bytes left to get
587                     bytesLeft -= size;
588
589                     // And reduce the residual size appropriately
590                     kdf->residual.t.size -= (UINT16)size;
591                 }
592                 else
593                 {
594                     UINT16           blocks = (UINT16)(bytesLeft / kdf->digestSize);
595     //
596                     // Get the number of required full blocks
597                     if(blocks > 0)
598                     {
599                         UINT16      size = blocks * kdf->digestSize;
600     // Get some number of full blocks and put them in the return buffer
601                         CryptKDFa(kdf->hash, kdf->seed, kdf->label, kdf->context, NULL,
602                                   kdf->limit, random, &counter, blocks);
603
604                         // reduce the size remaining to be moved and advance the pointer
605                         bytesLeft -= size;
606                         random += size;
607                     }
608                     else
609                     {
610                         // Fill the residual buffer with a full block and then loop to
611                         // top to get part of it copied to the output.
612                         kdf->residual.t.size = CryptKDFa(kdf->hash, kdf->seed,
613                                                          kdf->label, kdf->context, NULL,
614                                                          kdf->limit,
615                                                          kdf->residual.t.buffer,
616                                                          &counter, 1);
617                     }
618                 }
619             }
620             kdf->counter = counter;
621             return randomSize;
622         }
623         else if(state->drbg.magic == DRBG_MAGIC)
624         {
625             DRBG_STATE          *drbgState = (DRBG_STATE *)state;
626             DRBG_KEY_SCHEDULE    keySchedule;
627             DRBG_SEED           *seed = &drbgState->seed;
628
629             if(drbgState->reseedCounter >= CTR_DRBG_MAX_REQUESTS_PER_RESEED)
630             {
631                 if(drbgState == &drbgDefault)
632                 {
633                     DRBG_Reseed(drbgState, NULL, NULL);
634                     if(IsEntropyBad() && !IsSelfTest())
635                         return 0;
636                 }
637                 else
638                     // If this is a PRNG then the only way to get
```



```
639                     // here is if the SW has run away.
640                     FAIL(FATAL_ERROR_INTERNAL);
641             }
642             // if the allowed number of bytes in a request is larger than the
643             // less than the number of bytes that can be requested, then check
644     #if UINT16_MAX >=  CTR_DRBG_MAX_BYTES_PER_REQUEST
645             if(randomSize > CTR_DRBG_MAX_BYTES_PER_REQUEST)
646                 randomSize = CTR_DRBG_MAX_BYTES_PER_REQUEST;
647     #endif
648             // Create  encryption schedule
649             if(DRBG_ENCRYPT_SETUP((BYTE *)pDRBG_KEY(seed),
650                                   DRBG_KEY_SIZE_BITS, &keySchedule) != 0)
651                 FAIL(FATAL_ERROR_INTERNAL);
652             // Generate the random data
653             EncryptDRBG(random, randomSize, &keySchedule, pDRBG_IV(seed),
654                         drbgState->lastValue);
655             // Do a key update
656             DRBG_Update(drbgState, &keySchedule, NULL);
657
658             // Increment the reseed counter
659             drbgState->reseedCounter += 1;
660         }
661         else
662         {
663             FAIL(FATAL_ERROR_INTERNAL);
664         }
665         return randomSize;
666     }
```

10.2.17.3.5 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(1)	instantiation succeeded
FALSE(0)	instantiation failed

```
667     LIB_EXPORT BOOL
668     DRBG_Instantiate(
669         DRBG_STATE      *drbgState,         // OUT: the instantiated value
670         UINT16           pSize,             // IN: Size of personalization string
671         BYTE            *personalization    // IN: The personalization string
672         )
673     {
674         DRBG_SEED        seed;
675         DRBG_SEED        dfResult;
676     //
677         pAssert((pSize == 0) || (pSize <= sizeof(seed)) || (personalization != NULL));
678         // If the DRBG has not been tested, test when doing an instantiation. Since
679         // Instantiation is called during self test, make sure we don't get stuck in a
680         // loop.
681         if(!IsDrbgTested() && !IsSelfTest() && !DRBG_SelfTest())
682             return FALSE;
683         // If doing a self test, DRBG_GetEntropy will return the NIST
684         // test vector value.
685         if(!DRBG_GetEntropy(sizeof(seed), (BYTE *)&seed))
686             return FALSE;
687         // set everything to zero
688         memset(drbgState, 0, sizeof(DRBG_STATE));
689         drbgState->magic = DRBG_MAGIC;
690
```

```
691         // Steps 1, 2, 3, 6, 7 of SP 800-90A 10.2.1.3.1 are exactly what
692         // reseeding does. So, do a reduction on the personalization value (if any)
693         // and do a reseed.
694         DRBG_Reseed(drbgState, &seed, DfBuffer(&dfResult, pSize, personalization));
695
696         return TRUE;
697     }
```

10.2.17.3.6 DRBG_Uninstantiate()

This is Uninstantiate_function() from [SP 800-90A 9.4].

Error Returns	Meaning
TPM_RC_VALUE	not a valid

```
698     LIB_EXPORT TPM_RC
699     DRBG_Uninstantiate(
700         DRBG_STATE      *drbgState      // IN/OUT: working state to erase
701         )
702     {
703         if((drbgState == NULL) || (drbgState->magic != DRBG_MAGIC))
704             return TPM_RC_VALUE;
705         memset(drbgState, 0, sizeof(DRBG_STATE));
706         return TPM_RC_SUCCESS;
707     }
```

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"
3   #if     ALG_RSA
```

10.2.18.3 Obligatory Initialization Functions

10.2.18.3.1 CryptRsaInit()

Function called at _TPM_Init().

```
4   BOOL
5   CryptRsaInit(
6       void
7       )
8   {
9       return TRUE;
10  }
```

10.2.18.3.2 CryptRsaStartup()

Function called at TPM2_Startup()

```
11  BOOL
12  CryptRsaStartup(
13      void
14      )
15  {
16      return TRUE;
17  }
```

10.2.18.4 Internal Functions

```
18  void
19  RsaInitializeExponent(
20      privateExponent_t       *pExp
21      )
22  {
23  #if CRT_FORMAT_RSA == NO
24      BN_INIT(pExp->D);
25  #else
26      BN_INIT(pExp->Q);
27      BN_INIT(pExp->dP);
28      BN_INIT(pExp->dQ);
29      BN_INIT(pExp->qInv);
30  #endif
31  }
```


10.2.18.4.1 ComputePrivateExponent()

This function computes the private exponent from the primes.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
32  static BOOL
33  ComputePrivateExponent(
34      bigNum                P,              // IN: first prime (size is 1/2 of bnN)
35      bigNum                Q,              // IN: second prime (size is 1/2 of bnN)
36      bigNum                E,              // IN: the public exponent
37      bigNum                N,              // IN: the public modulus
38      privateExponent_t    *pExp            // OUT:
39      )
40  {
41      BOOL                 pOK;
42      BOOL                 qOK;
43  #if CRT_FORMAT_RSA == NO
44      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
63      // make p the larger value so that m2 is always less than p
64      if(BnUnsignedCmp(P, Q) < 0)
65      {
66          pT = P;
67          P = Q;
68          Q = pT;
69      }
70      //dP = (1/e) mod (p-1) = d mod (p-1)
71      pOK = BnSubWord(temp, P, 1);
72      pOK = pOK && BnModInverse((bigNum)&pExp->dP, E, temp);
73      //dQ = (1/e) mod (q-1) = d mod (q-1)
74      qOK = BnSubWord(temp, Q, 1);
75      qOK = qOK && BnModInverse((bigNum)&pExp->dQ, E, temp);
76      // qInv = (1/q) mod p
77      if(pOK && qOK)
78          pOK = qOK = BnModInverse((bigNum)&pExp->qInv, Q, P);
79  #endif
80      if(!pOK)
81          BnSetWord(P, 0);
82      if(!qOK)
83          BnSetWord(Q, 0);
84
85      return pOK && qOK;
86
```



```
87  }
```

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.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
88  static BOOL
89  RsaPrivateKeyOp(
90      bigNum                inOut, // IN/OUT: number to be exponentiated
91      bigNum                N,     // IN: public modulus (can be NULL if CRT)
92      bigNum                P,     // IN: one of the primes (can be NULL if not CRT)
93      privateExponent_t    *pExp
94      )
95  {
96      BOOL                  OK;
97  #if CRT_FORMAT_RSA == NO
98      (P);
99      OK = BnModExp(inOut, inOut, (bigNum)&pExp->D, N);
100 #else
101     BN_RSA(M1);
102     BN_RSA(M2);
103     BN_RSA(M);
104     BN_RSA(H);
105     bigNum               Q = (bigNum)&pExp->Q;
106     NOT_REFERENCED(N);
107     // Make P the larger prime.
108     // NOTE that when the CRT form of the private key is created, dP will always
109     // be computed using the larger of p and q so the only thing needed here is that
110     // the primes be selected so that they agree with dP.
111     if(BnUnsignedCmp(P, Q) < 0)
112     {
113         bigNum       T = P;
114         P = Q;
115         Q = T;
116     }
117     // m1 = cdP mod p
118     OK = BnModExp(M1, inOut, (bigNum)&pExp->dP, P);
119     // m2 = cdQ mod q
120     OK = OK && BnModExp(M2, inOut, (bigNum)&pExp->dQ, Q);
121     // h = qInv * (m1 - m2) mod p = qInv * (m1 + P - m2) mod P because Q < P
122     // so m2 < P
123     OK = OK && BnSub(H, P, M2);
124     OK = OK && BnAdd(H, H, M1);
125     OK = OK && BnModMult(H, H, (bigNum)&pExp->qInv, P);
126     // m = m2 + h * q
127     OK = OK && BnMult(M, H, Q);
128     OK = OK && BnAdd(inOut, M2, M);
129 #endif
130     return OK;
131 }
```

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

```
132 static TPM_RC
133 RSAEP(
134     TPM2B       *dInOut,        // IN: size of the encrypted block and the size of
135                                 //     the encrypted value. It must be the size of
136                                 //     the modulus.
137                                 // OUT: the encrypted data. Will receive the
138                                 //      decrypted value
139     OBJECT      *key            // IN: the key to use
140     )
141 {
142     TPM2B_TYPE(4BYTES, 4);
143     TPM2B_4BYTES(e) = {{4, {(BYTE)((RSA_DEFAULT_PUBLIC_EXPONENT >> 24) & 0xff),
144                            (BYTE)((RSA_DEFAULT_PUBLIC_EXPONENT >> 16) & 0xff),
145                            (BYTE)((RSA_DEFAULT_PUBLIC_EXPONENT >> 8) & 0xff),
146                            (BYTE)((RSA_DEFAULT_PUBLIC_EXPONENT)& 0xff)}}};
147 //
148     if(key->publicArea.parameters.rsaDetail.exponent != 0)
149         UINT32_TO_BYTE_ARRAY(key->publicArea.parameters.rsaDetail.exponent,
150                              e.t.buffer);
151
152     return ModExpB(dInOut->size, dInOut->buffer, dInOut->size, dInOut->buffer,
153                    e.t.size, e.t.buffer, key->publicArea.unique.rsa.t.size,
154                    key->publicArea.unique.rsa.t.buffer);
155 }
```

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

```
156 static TPM_RC
157 RSADP(
158     TPM2B           *inOut,         // IN/OUT: the value to encrypt
159     OBJECT          *key            // IN: the key
160     )
161 {
162     BN_RSA_INITIALIZED(bnM, inOut);
163     BN_RSA_INITIALIZED(bnN, &key->publicArea.unique.rsa);
164     BN_RSA_INITIALIZED(bnP, &key->sensitive.sensitive.rsa);
165     if(BnUnsignedCmp(bnM, bnN) >= 0)
166         return TPM_RC_SIZE;
167     // private key operation requires that private exponent be loaded
168     // During self-test, this might not be the case so load it up if it hasn't
169     // already done
170     // been done
171     if(!key->attributes.privateExp)
172         CryptRsaLoadPrivateExponent(key);
173
174     if(!RsaPrivateKeyOp(bnM, bnN, bnP, &key->privateExponent))
175         FAIL(FATAL_ERROR_INTERNAL);
176     BnTo2B(bnM, inOut, inOut->size);
177     return TPM_RC_SUCCESS;
```



```
178 }
```

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

```
179 static TPM_RC
180 OaepEncode(
181     TPM2B       *padded,        // OUT: the pad data
182     TPM_ALG_ID   hashAlg,       // IN: algorithm to use for padding
183     const TPM2B *label,         // IN: null-terminated string (may be NULL)
184     TPM2B       *message,       // IN: the message being padded
185     RAND_STATE  *rand           // IN: the random number generator to use
186     )
187 {
188     INT32        padLen;
189     INT32        dbSize;
190     INT32        i;
191     BYTE         mySeed[MAX_DIGEST_SIZE];
192     BYTE        *seed = mySeed;
193     INT32        hLen = CryptHashGetDigestSize(hashAlg);
194     BYTE         mask[MAX_RSA_KEY_BYTES];
195     BYTE        *pp;
196     BYTE        *pm;
197     TPM_RC       retVal = TPM_RC_SUCCESS;
198
199     pAssert(padded != NULL && message != NULL);
200
201     // A value of zero is not allowed because the KDF can't produce a result
202     // if the digest size is zero.
203     if(hLen <= 0)
204         return TPM_RC_VALUE;
205
206     // Basic size checks
207     //  make sure digest isn't too big for key size
208     if(padded->size < (2 * hLen) + 2)
209         ERROR_RETURN(TPM_RC_HASH);
210
211     // and that message will fit messageSize <= k - 2hLen - 2
212     if(message->size > (padded->size - (2 * hLen) - 2))
213         ERROR_RETURN(TPM_RC_VALUE);
214
215     // Hash L even if it is null
216     // Offset into padded leaving room for masked seed and byte of zero
217     pp = &padded->buffer[hLen + 1];
218     if(CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
219                       hLen, pp) != hLen)
220         ERROR_RETURN(TPM_RC_FAILURE);
221
222     // concatenate PS of k  mLen  2hLen  2
223     padLen = padded->size - message->size - (2 * hLen) - 2;
224     MemorySet(&pp[hLen], 0, padLen);
225     pp[hLen + padLen] = 0x01;
226     padLen += 1;
227     memcpy(&pp[hLen + padLen], message->buffer, message->size);
228
229     // The total size of db = hLen + pad + mSize;
230     dbSize = hLen + padLen + message->size;
231
```

```
232         // If testing, then use the provided seed. Otherwise, use values
233         // from the RNG
234         CryptRandomGenerate(hLen, mySeed);
235         DRBG_Generate(rand, mySeed, (UINT16)hLen);
236         // mask = MGF1 (seed, nSize  hLen  1)
237         CryptMGF1(dbSize, mask, hashAlg, hLen, seed);
238
239         // Create the masked db
240         pm = mask;
241         for(i = dbSize; i > 0; i--)
242             *pp++ ^= *pm++;
243         pp = &padded->buffer[hLen + 1];
244
245         // Run the masked data through MGF1
246         if(CryptMGF1(hLen, &padded->buffer[1], hashAlg, dbSize, pp) != (unsigned)hLen)
247             ERROR_RETURN(TPM_RC_VALUE);
248     // Now XOR the seed to create masked seed
249         pp = &padded->buffer[1];
250         pm = seed;
251         for(i = hLen; i > 0; i--)
252             *pp++ ^= *pm++;
253         // Set the first byte to zero
254         padded->buffer[0] = 0x00;
255     Exit:
256         return retVal;
257     }
```

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

```
258     static TPM_RC
259     OaepDecode(
260         TPM2B           *dataOut,       // OUT: the recovered data
261         TPM_ALG_ID       hashAlg,       // IN: algorithm to use for padding
262         const TPM2B     *label,         // IN: null-terminated string (may be NULL)
263         TPM2B           *padded         // IN: the padded data
264         )
265     {
266         UINT32          i;
267         BYTE            seedMask[MAX_DIGEST_SIZE];
268         UINT32          hLen = CryptHashGetDigestSize(hashAlg);
269
270         BYTE            mask[MAX_RSA_KEY_BYTES];
271         BYTE           *pp;
272         BYTE           *pm;
273         TPM_RC          retVal = TPM_RC_SUCCESS;
274
275         // Strange size (anything smaller can't be an OAEP padded block)
276         // Also check for no leading 0
277         if((padded->size < (unsigned)((2 * hLen) + 2)) || (padded->buffer[0] != 0))
278             ERROR_RETURN(TPM_RC_VALUE);
279     // Use the hash size to determine what to put through MGF1 in order
280     // to recover the seedMask
281         CryptMGF1(hLen, seedMask, hashAlg, padded->size - hLen - 1,
```



```
282                 &padded->buffer[hLen + 1]);
283
284         // Recover the seed into seedMask
285         pAssert(hLen <= sizeof(seedMask));
286         pp = &padded->buffer[1];
287         pm = seedMask;
288         for(i = hLen; i > 0; i--)
289             *pm++ ^= *pp++;
290
291         // Use the seed to generate the data mask
292         CryptMGF1(padded->size - hLen - 1, mask, hashAlg, hLen, seedMask);
293
294         // Use the mask generated from seed to recover the padded data
295         pp = &padded->buffer[hLen + 1];
296         pm = mask;
297         for(i = (padded->size - hLen - 1); i > 0; i--)
298             *pm++ ^= *pp++;
299
300         // Make sure that the recovered data has the hash of the label
301         // Put trial value in the seed mask
302         if((CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
303                            hLen, seedMask)) != hLen)
304             FAIL(FATAL_ERROR_INTERNAL);
305         if(memcmp(seedMask, mask, hLen) != 0)
306             ERROR_RETURN(TPM_RC_VALUE);
307
308         // find the start of the data
309         pm = &mask[hLen];
310         for(i = (UINT32)padded->size - (2 * hLen) - 1; i > 0; i--)
311         {
312             if(*pm++ != 0)
313                 break;
314         }
315         // If we ran out of data or didn't end with 0x01, then return an error
316         if(i == 0 || pm[-1] != 0x01)
317             ERROR_RETURN(TPM_RC_VALUE);
318
319         // pm should be pointing at the first part of the data
320         // and i is one greater than the number of bytes to move
321         i--;
322         if(i > dataOut->size)
323             // Special exit to preserve the size of the output buffer
324             return TPM_RC_VALUE;
325         memcpy(dataOut->buffer, pm, i);
326         dataOut->size = (UINT16)i;
327     Exit:
328         if(retVal != TPM_RC_SUCCESS)
329             dataOut->size = 0;
330         return retVal;
331     }
```

10.2.18.4.7 PKCS1v1_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

```
332 static TPM_RC
333 RSAES_PKCS1v1_5Encode(
334     TPM2B       *padded,        // OUT: the pad data
335     TPM2B       *message,       // IN: the message being padded
336     RAND_STATE  *rand
```

```
337         )
338     {
339         UINT32      ps = padded->size - message->size - 3;
340     //
341         if(message->size > padded->size - 11)
342             return TPM_RC_VALUE;
343         // move the message to the end of the buffer
344         memcpy(&padded->buffer[padded->size - message->size], message->buffer,
345                message->size);
346         // Set the first byte to 0x00 and the second to 0x02
347         padded->buffer[0] = 0;
348         padded->buffer[1] = 2;
349
350         // Fill with random bytes
351         DRBG_Generate(rand, &padded->buffer[2], (UINT16)ps);
352
353         // Set the delimiter for the random field to 0
354         padded->buffer[2 + ps] = 0;
355
356         // Now, the only messy part. Make sure that all the 'ps' bytes are non-zero
357         // In this implementation, use the value of the current index
358         for(ps++; ps > 1; ps--)
359         {
360             if(padded->buffer[ps] == 0)
361                 padded->buffer[ps] = 0x55;  // In the < 0.5% of the cases that the
362                                             // random value is 0, just pick a value to
363                                             // put into the spot.
364         }
365         return TPM_RC_SUCCESS;
366     }
```

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

```
367     static TPM_RC
368     RSAES_Decode(
369         TPM2B       *message,       // OUT: the recovered message
370         TPM2B       *coded          // IN: the encoded message
371         )
372     {
373         BOOL        fail = FALSE;
374         UINT16      pSize;
375
376         fail = (coded->size < 11);
377         fail = (coded->buffer[0] != 0x00) | fail;
378         fail = (coded->buffer[1] != 0x02) | fail;
379         for(pSize = 2; pSize < coded->size; pSize++)
380         {
381             if(coded->buffer[pSize] == 0)
382                 break;
383         }
384         pSize++;
385
386         // Make sure that pSize has not gone over the end and that there are at least 8
387         // bytes of pad data.
388         fail = (pSize >= coded->size) | fail;
389         fail = ((pSize - 2) < 8) | fail;
390         if((message->size < (UINT16)(coded->size - pSize)) || fail)
391             return TPM_RC_VALUE;
```



```
392     message->size = coded->size - pSize;
393     memcpy(message->buffer, &coded->buffer[pSize], coded->size - pSize);
394     return TPM_RC_SUCCESS;
395 }
```

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.

```
396 static TPM_RC
397 PssEncode(
398     TPM2B           *out,       // OUT: the encoded buffer
399     TPM_ALG_ID       hashAlg,   // IN: hash algorithm for the encoding
400     TPM2B           *digest,    // IN: the digest
401     RAND_STATE      *rand       // IN: random number source
402     )
403 {
404     UINT32               hLen = CryptHashGetDigestSize(hashAlg);
405     BYTE                 salt[MAX_RSA_KEY_BYTES - 1];
406     UINT16               saltSize;
407     BYTE                *ps = salt;
408     BYTE                *pOut;
409     UINT16               mLen;
410     HASH_STATE           hashState;
411
412     // These are fatal errors indicating bad TPM firmware
413     pAssert(out != NULL && hLen > 0 && digest != NULL);
414
415     // Get the size of the mask
416     mLen = (UINT16)(out->size - hLen - 1);
417
418     // Maximum possible salt size is mask length - 1
419     saltSize = mLen - 1;
420
421     // Use the maximum salt size allowed by FIPS 186-4
422     if(saltSize > hLen)
423         saltSize = (UINT16)hLen;
424
425 //using eOut for scratch space
426     // Set the first 8 bytes to zero
427     pOut = out->buffer;
428     memset(pOut, 0, 8);
429
430     // Get set the salt
431     DRBG_Generate(rand, salt, saltSize);
432
433     // Create the hash of the pad || input hash || salt
434     CryptHashStart(&hashState, hashAlg);
435     CryptDigestUpdate(&hashState, 8, pOut);
436     CryptDigestUpdate2B(&hashState, digest);
437     CryptDigestUpdate(&hashState, saltSize, salt);
438     CryptHashEnd(&hashState, hLen, &pOut[out->size - hLen - 1]);
439
440     // Create a mask
441     if(CryptMGF1(mLen, pOut, hashAlg, hLen, &pOut[mLen]) != mLen)
442         FAIL(FATAL_ERROR_INTERNAL);
443
444     // Since this implementation uses key sizes that are all even multiples of
445     // 8, just need to make sure that the most significant bit is CLEAR
446     *pOut &= 0x7f;
447
```

```
448         // Before we mess up the pOut value, set the last byte to 0xbc
449         pOut[out->size - 1] = 0xbc;
450
451         // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
452         pOut = &pOut[mLen - saltSize - 1];
453         *pOut++ ^= 0x01;
454
455         // XOR the salt data into the buffer
456         for(; saltSize > 0; saltSize--)
457             *pOut++ ^= *ps++;
458
459         // and we are done
460         return TPM_RC_SUCCESS;
461     }
```

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

```
462     static TPM_RC
463     PssDecode(
464         TPM_ALG_ID   hashAlg,        // IN: hash algorithm to use for the encoding
465         TPM2B       *dIn,            // In: the digest to compare
466         TPM2B       *eIn             // IN: the encoded data
467         )
468     {
469         UINT32          hLen = CryptHashGetDigestSize(hashAlg);
470         BYTE            mask[MAX_RSA_KEY_BYTES];
471         BYTE           *pm = mask;
472         BYTE           *pe;
473         BYTE            pad[8] = {0};
474         UINT32          i;
475         UINT32          mLen;
476         BYTE            fail;
477         TPM_RC          retVal = TPM_RC_SUCCESS;
478         HASH_STATE      hashState;
479
480         // These errors are indicative of failures due to programmer error
481         pAssert(dIn != NULL && eIn != NULL);
482         pe = eIn->buffer;
483
484         // check the hash scheme
485         if(hLen == 0)
486             ERROR_RETURN(TPM_RC_SCHEME);
487
488         // most significant bit must be zero
489         fail = pe[0] & 0x80;
490
491         // last byte must be 0xbc
492         fail |= pe[eIn->size - 1] ^ 0xbc;
493
494         // Use the hLen bytes at the end of the buffer to generate a mask
```

```
495         // Doesn't start at the end which is a flag byte
496         mLen = eIn->size - hLen - 1;
497         CryptMGF1(mLen, mask, hashAlg, hLen, &pe[mLen]);
498
499         // Clear the MSO of the mask to make it consistent with the encoding.
500         mask[0] &= 0x7F;
501
502         pAssert(mLen <= sizeof(mask));
503         // XOR the data into the mask to recover the salt. This sequence
504         // advances eIn so that it will end up pointing to the seed data
505         // which is the hash of the signature data
506         for(i = mLen; i > 0; i--)
507             *pm++ ^= *pe++;
508
509         // Find the first byte of 0x01 after a string of all 0x00
510         for(pm = mask, i = mLen; i > 0; i--)
511         {
512             if(*pm == 0x01)
513                 break;
514             else
515                 fail |= *pm++;
516         }
517         // i should not be zero
518         fail |= (i == 0);
519
520         // if we have failed, will continue using the entire mask as the salt value so
521         // that the timing attacks will not disclose anything (I don't think that this
522         // is a problem for TPM applications but, usually, we don't fail so this
523         // doesn't cost anything).
524         if(fail)
525         {
526             i = mLen;
527             pm = mask;
528         }
529         else
530         {
531             pm++;
532             i--;
533         }
534         // i contains the salt size and pm points to the salt. Going to use the input
535         // hash and the seed to recreate the hash in the lower portion of eIn.
536         CryptHashStart(&hashState, hashAlg);
537
538         // add the pad of 8 zeros
539         CryptDigestUpdate(&hashState, 8, pad);
540
541         // add the provided digest value
542         CryptDigestUpdate(&hashState, dIn->size, dIn->buffer);
543
544         // and the salt
545         CryptDigestUpdate(&hashState, i, pm);
546
547         // get the result
548         fail |= (CryptHashEnd(&hashState, hLen, mask) != hLen);
549
550         // Compare all bytes
551         for(pm = mask; hLen > 0; hLen--)
552             // don't use fail = because that could skip the increment and compare
553             // operations after the first failure and that gives away timing
554             // information.
555             fail |= *pm++ ^ *pe++;
556
557     retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
558 Exit:
559             return retVal;
560 }
```

10.2.18.4.11 RSASSA_Encode()

Encode a message using PKCS1v1().5 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*

```
561 static TPM_RC
562 RSASSA_Encode(
563     TPM2B                *pOut,      // IN:OUT on in, the size of the public key
564                                      //        on out, the encoded area
565     TPM_ALG_ID            hashAlg,   // IN: hash algorithm for PKCS1v1_5
566     TPM2B                *hIn        // IN: digest value to encode
567     )
568 {
569     const BYTE      *der;
570     BYTE            *eOut;
571     INT32            derSize = CryptHashGetDer(hashAlg, &der);
572     INT32            fillSize;
573     TPM_RC           retVal = TPM_RC_SUCCESS;
574
575     // Can't use this scheme if the algorithm doesn't have a DER string defined.
576     if(derSize == 0)
577         ERROR_RETURN(TPM_RC_SCHEME);
578
579     // If the digest size of 'hashAl' doesn't match the input digest size, then
580     // the DER will misidentify the digest so return an error
581     if(CryptHashGetDigestSize(hashAlg) != hIn->size)
582         ERROR_RETURN(TPM_RC_VALUE);
583     fillSize = pOut->size - derSize - hIn->size - 3;
584     eOut = pOut->buffer;
585
586     // Make sure that this combination will fit in the provided space
587     if(fillSize < 8)
588         ERROR_RETURN(TPM_RC_SIZE);
589
590     // Start filling
591     *eOut++ = 0; // initial byte of zero
592     *eOut++ = 1; // byte of 0x01
593     for(; fillSize > 0; fillSize--)
594         *eOut++ = 0xff; // bunch of 0xff
595     *eOut++ = 0; // another 0
596     for(; derSize > 0; derSize--)
597         *eOut++ = *der++;   // copy the DER
598     der = hIn->buffer;
599     for(fillSize = hIn->size; fillSize > 0; fillSize--)
600         *eOut++ = *der++;   // copy the hash
601 Exit:
602     return retVal;
603 }
```

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

```
604  static TPM_RC
605  RSASSA_Decode(
606      TPM_ALG_ID       hashAlg,        // IN: hash algorithm to use for the encoding
607      TPM2B           *hIn,            // In: the digest to compare
608      TPM2B           *eIn             // IN: the encoded data
609      )
610  {
611      BYTE             fail;
612      const BYTE      *der;
613      BYTE            *pe;
614      INT32            derSize = CryptHashGetDer(hashAlg, &der);
615      INT32            hashSize = CryptHashGetDigestSize(hashAlg);
616      INT32            fillSize;
617      TPM_RC           retVal;
618      BYTE            *digest;
619      UINT16           digestSize;
620
621      pAssert(hIn != NULL && eIn != NULL);
622      pe = eIn->buffer;
623
624      // Can't use this scheme if the algorithm doesn't have a DER string
625      // defined or if the provided hash isn't the right size
626      if(derSize == 0 || (unsigned)hashSize != hIn->size)
627          ERROR_RETURN(TPM_RC_SCHEME);
628
629      // Make sure that this combination will fit in the provided space
630      // Since no data movement takes place, can just walk though this
631      // and accept nearly random values. This can only be called from
632      // CryptValidateSignature() so eInSize is known to be in range.
633      fillSize = eIn->size - derSize - hashSize - 3;
634
635      // Start checking (fail will become non-zero if any of the bytes do not have
636      // the expected value.
637      fail = *pe++;                   // initial byte of zero
638      fail |= *pe++ ^ 1;              // byte of 0x01
639      for(; fillSize > 0; fillSize--)
640          fail |= *pe++ ^ 0xff;       // bunch of 0xff
641      fail |= *pe++;                  // another 0
642      for(; derSize > 0; derSize--)
643          fail |= *pe++ ^ *der++;    // match the DER
644      digestSize = hIn->size;
645      digest = hIn->buffer;
646      for(; digestSize > 0; digestSize--)
647          fail |= *pe++ ^ *digest++; // match the hash
648      retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
649  Exit:
650      return retVal;
651  }
```

10.2.18.5 Externally Accessible Functions

10.2.18.5.1 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.

```
652 TPMT_RSA_DECRYPT*
653 CryptRsaSelectScheme(
654     TPMI_DH_OBJECT       rsaHandle,      // IN: handle of an RSA key
655     TPMT_RSA_DECRYPT    *scheme          // IN: a sign or decrypt scheme
656     )
657 {
658     OBJECT              *rsaObject;
659     TPMT_ASYM_SCHEME    *keyScheme;
660     TPMT_RSA_DECRYPT    *retVal = NULL;
661
662     // Get sign object pointer
663     rsaObject = HandleToObject(rsaHandle);
664     keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
665
666     // if the default scheme of the object is TPM_ALG_NULL, then select the
667     // input scheme
668     if(keyScheme->scheme == TPM_ALG_NULL)
669     {
670         retVal = scheme;
671     }
672     // if the object scheme is not TPM_ALG_NULL and the input scheme is
673     // ALG_NULL, then select the default scheme of the object.
674     else if(scheme->scheme == TPM_ALG_NULL)
675     {
676         // if input scheme is NULL
677         retVal = (TPMT_RSA_DECRYPT *)keyScheme;
678     }
679     // get here if both the object scheme and the input scheme are
680     // not TPM_ALG_NULL. Need to insure that they are the same.
681     // IMPLEMENTATION NOTE: This could cause problems if future versions have
682     // schemes that have more values than just a hash algorithm. A new function
683     // (IsSchemeSame()) might be needed then.
684     else if(keyScheme->scheme == scheme->scheme
685             && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
686     {
687         retVal = scheme;
688     }
689     // two different, incompatible schemes specified will return NULL
690     return retVal;
691 }
```

10.2.18.5.2 CryptRsaLoadPrivateExponent()

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of *rsaKey* are not matched

```
692 TPM_RC
693 CryptRsaLoadPrivateExponent(
694     OBJECT          *rsaKey         // IN: the RSA key object
695     )
696 {
697     BN_RSA_INITIALIZED(bnN, &rsaKey->publicArea.unique.rsa);
698     BN_PRIME_INITIALIZED(bnP, &rsaKey->sensitive.sensitive.rsa);
699     BN_RSA(bnQ);
700     BN_PRIME(bnQr);
701     BN_WORD_INITIALIZED(bnE, (rsaKey->publicArea.parameters.rsaDetail.exponent == 0)
702                         ? RSA_DEFAULT_PUBLIC_EXPONENT
```



```
                        : rsaKey->publicArea.parameters.rsaDetail.exponent);
    TPM_RC          retVal = TPM_RC_SUCCESS;
    if(!rsaKey->attributes.privateExp)
    {
        TEST(ALG_NULL_VALUE);

        // Make sure that the bigNum used for the exponent is properly initialized
        RsaInitializeExponent(&rsaKey->privateExponent);

        // Find the second prime by division
        BnDiv(bnQ, bnQr, bnN, bnP);
        if(!BnEqualZero(bnQr))
            ERROR_RETURN(TPM_RC_BINDING);
        // Compute the private exponent and return it if found
        if(!ComputePrivateExponent(bnP, bnQ, bnE, bnN,
                                   &rsaKey->privateExponent))
            ERROR_RETURN(TPM_RC_BINDING);
    }
Exit:
    rsaKey->attributes.privateExp = (retVal == TPM_RC_SUCCESS);
    return retVal;
}
```

10.2.18.5.3 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

```
LIB_EXPORT TPM_RC
CryptRsaEncrypt(
    TPM2B_PUBLIC_KEY_RSA        *cOut,          // OUT: the encrypted data
    TPM2B                       *dIn,           // IN: the data to encrypt
    OBJECT                      *key,           // IN: the key used for encryption
    TPMT_RSA_DECRYPT            *scheme,        // IN: the type of padding and hash
                                                //     if needed
    const TPM2B                 *label,         // IN: in case it is needed
    RAND_STATE                  *rand           // IN: random number generator
                                                //     state (mostly for testing)
    )
{
    TPM_RC                       retVal = TPM_RC_SUCCESS;
    TPM2B_PUBLIC_KEY_RSA         dataIn;
//
    // if the input and output buffers are the same, copy the input to a scratch
    // buffer so that things don't get messed up.
    if(dIn == &cOut->b)
    {
        MemoryCopy2B(&dataIn.b, dIn, sizeof(dataIn.t.buffer));
```

```
745             dIn = &dataIn.b;
746         }
747         // All encryption schemes return the same size of data
748         cOut->t.size = key->publicArea.unique.rsa.t.size;
749         TEST(scheme->scheme);
750
751         switch(scheme->scheme)
752         {
753             case ALG_NULL_VALUE:  // 'raw' encryption
754             {
755                 INT32            i;
756                 INT32            dSize = dIn->size;
757                 // dIn can have more bytes than cOut as long as the extra bytes
758                 // are zero. Note: the more significant bytes of a number in a byte
759                 // buffer are the bytes at the start of the array.
760                 for(i = 0; (i < dSize) && (dIn->buffer[i] == 0); i++);
761                 dSize -= i;
762                 if(dSize > cOut->t.size)
763                     ERROR_RETURN(TPM_RC_VALUE);
764                 // Pad cOut with zeros if dIn is smaller
765                 memset(cOut->t.buffer, 0, cOut->t.size - dSize);
766                 // And copy the rest of the value
767                 memcpy(&cOut->t.buffer[cOut->t.size - dSize], &dIn->buffer[i], dSize);
768
769                 // If the size of dIn is the same as cOut dIn could be larger than
770                 // the modulus. If it is, then RSAEP() will catch it.
771             }
772             break;
773             case ALG_RSAES_VALUE:
774                 retVal = RSAES_PKCS1v1_5Encode(&cOut->b, dIn, rand);
775                 break;
776             case ALG_OAEP_VALUE:
777                 retVal = OaepEncode(&cOut->b, scheme->details.oaep.hashAlg, label, dIn,
778                                     rand);
779                 break;
780             default:
781                 ERROR_RETURN(TPM_RC_SCHEME);
782                 break;
783         }
784         // All the schemes that do padding will come here for the encryption step
785         // Check that the Encoding worked
786         if(retVal == TPM_RC_SUCCESS)
787             // Padding OK so do the encryption
788             retVal = RSAEP(&cOut->b, key);
789     Exit:
790         return retVal;
791     }
```

10.2.18.5.4 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

```
792 LIB_EXPORT TPM_RC
793 CryptRsaDecrypt(
794     TPM2B               *dOut,          // OUT: the decrypted data
```

```
795     TPM2B               *cIn,           // IN: the data to decrypt
796     OBJECT              *key,           // IN: the key to use for decryption
797     TPMT_RSA_DECRYPT    *scheme,        // IN: the padding scheme
798     const TPM2B         *label          // IN: in case it is needed for the scheme
799     )
800 {
801     TPM_RC                  retVal;
802
803     // Make sure that the necessary parameters are provided
804     pAssert(cIn != NULL && dOut != NULL && key != NULL);
805
806     // Size is checked to make sure that the encrypted value is the right size
807     if(cIn->size != key->publicArea.unique.rsa.t.size)
808         ERROR_RETURN(TPM_RC_SIZE);
809
810     TEST(scheme->scheme);
811
812     // For others that do padding, do the decryption in place and then
813     // go handle the decoding.
814     retVal = RSADP(cIn, key);
815     if(retVal == TPM_RC_SUCCESS)
816     {
817         // Remove padding
818         switch(scheme->scheme)
819         {
820             case ALG_NULL_VALUE:
821                 if(dOut->size < cIn->size)
822                     return TPM_RC_VALUE;
823                 MemoryCopy2B(dOut, cIn, dOut->size);
824                 break;
825             case ALG_RSAES_VALUE:
826                 retVal = RSAES_Decode(dOut, cIn);
827                 break;
828             case ALG_OAEP_VALUE:
829                 retVal = OaepDecode(dOut, scheme->details.oaep.hashAlg, label, cIn);
830                 break;
831             default:
832                 retVal = TPM_RC_SCHEME;
833                 break;
834         }
835     }
836 Exit:
837     return retVal;
838 }
```

10.2.18.5.5 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)

```
839 LIB_EXPORT TPM_RC
840 CryptRsaSign(
841     TPMT_SIGNATURE      *sigOut,
842     OBJECT              *key,           // IN: key to use
843     TPM2B_DIGEST        *hIn,           // IN: the digest to sign
844     RAND_STATE          *rand           // IN: the random number generator
845                                         //      to use (mostly for testing)
846     )
847 {
```

```
    TPM_RC                  retVal = TPM_RC_SUCCESS;
    UINT16                  modSize;

    // parameter checks
    pAssert(sigOut != NULL && key != NULL && hIn != NULL);

    modSize = key->publicArea.unique.rsa.t.size;

    // for all non-null signatures, the size is the size of the key modulus
    sigOut->signature.rsapss.sig.t.size = modSize;

    TEST(sigOut->sigAlg);

    switch(sigOut->sigAlg)
    {
        case ALG_NULL_VALUE:
            sigOut->signature.rsapss.sig.t.size = 0;
            return TPM_RC_SUCCESS;
        case ALG_RSAPSS_VALUE:
            retVal = PssEncode(&sigOut->signature.rsapss.sig.b,
                               sigOut->signature.rsapss.hash, &hIn->b, rand);
            break;
        case ALG_RSASSA_VALUE:
            retVal = RSASSA_Encode(&sigOut->signature.rsassa.sig.b,
                                   sigOut->signature.rsassa.hash, &hIn->b);
            break;
        default:
            retVal = TPM_RC_SCHEME;
    }
    if(retVal == TPM_RC_SUCCESS)
    {
        // Do the encryption using the private key
        retVal = RSADP(&sigOut->signature.rsapss.sig.b, key);
    }
    return retVal;
}
```

10.2.18.5.6 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

```
LIB_EXPORT TPM_RC
CryptRsaValidateSignature(
    TPMT_SIGNATURE  *sig,           // IN: signature
    OBJECT          *key,           // IN: public modulus
    TPM2B_DIGEST    *digest         // IN: The digest being validated
    )
{
    TPM_RC          retVal;
//
    // Fatal programming errors
    pAssert(key != NULL && sig != NULL && digest != NULL);
    switch(sig->sigAlg)
    {
        case ALG_RSAPSS_VALUE:
        case ALG_RSASSA_VALUE:
```

```
899                 break;
900             default:
901                 return TPM_RC_SCHEME;
902         }
903
904         // Errors that might be caused by calling parameters
905         if(sig->signature.rsassa.sig.t.size != key->publicArea.unique.rsa.t.size)
906             ERROR_RETURN(TPM_RC_SIGNATURE);
907
908         TEST(sig->sigAlg);
909
910         // Decrypt the block
911         retVal = RSAEP(&sig->signature.rsassa.sig.b, key);
912         if(retVal == TPM_RC_SUCCESS)
913         {
914             switch(sig->sigAlg)
915             {
916                 case ALG_RSAPSS_VALUE:
917                     retVal = PssDecode(sig->signature.any.hashAlg, &digest->b,
918                                        &sig->signature.rsassa.sig.b);
919                     break;
920                 case ALG_RSASSA_VALUE:
921                     retVal = RSASSA_Decode(sig->signature.any.hashAlg, &digest->b,
922                                            &sig->signature.rsassa.sig.b);
923                     break;
924                 default:
925                     return TPM_RC_SCHEME;
926             }
927         }
928     Exit:
929         return (retVal != TPM_RC_SUCCESS) ? TPM_RC_SIGNATURE : TPM_RC_SUCCESS;
930     }
931     #if SIMULATION && USE_RSA_KEY_CACHE
932     extern int s_rsaKeyCacheEnabled;
933     int GetCachedRsaKey(OBJECT *key, RAND_STATE *rand);
934     #define GET_CACHED_KEY(key, rand)                                   \
935                 (s_rsaKeyCacheEnabled && GetCachedRsaKey(key, rand))
936     #else
937     #define GET_CACHED_KEY(key, rand)
938     #endif
```

10.2.18.5.7 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

```
939     LIB_EXPORT TPM_RC
940     CryptRsaGenerateKey(
941         OBJECT              *rsaKey,            // IN/OUT: The object structure in which
942                                                 //          the key is created.
943         RAND_STATE          *rand               // IN: if not NULL, the deterministic
944                                                 //     RNG state
945         )
946     {
947         UINT32               i;
948         BN_PRIME(bnP); // These four declarations initialize the number to 0
949         BN_PRIME(bnQ);
950         BN_RSA(bnD);
```

```
951         BN_RSA(bnN);
952         BN_WORD(bnE);
953         UINT32                e;
954         int                   keySizeInBits;
955         TPMT_PUBLIC          *publicArea = &rsaKey->publicArea;
956         TPMT_SENSITIVE       *sensitive = &rsaKey->sensitive;
957         TPM_RC                retVal = TPM_RC_NO_RESULT;
958     //
959
960     // Need to make sure that the caller did not specify an exponent that is
961     // not supported
962         e = publicArea->parameters.rsaDetail.exponent;
963         if(e == 0)
964             e = RSA_DEFAULT_PUBLIC_EXPONENT;
965         if(e < 65537)
966             ERROR_RETURN(TPM_RC_RANGE);
967         if(e != RSA_DEFAULT_PUBLIC_EXPONENT && !IsPrimeInt(e))
968             ERROR_RETURN(TPM_RC_RANGE);
969         BnSetWord(bnE, e);
970         // Check that e is prime
971         // check for supported key size.
972         keySizeInBits = publicArea->parameters.rsaDetail.keyBits;
973         if(((keySizeInBits % 1024) != 0)
974            || (keySizeInBits > MAX_RSA_KEY_BITS)  // this might be redundant, but...
975            || (keySizeInBits == 0))
976             ERROR_RETURN(TPM_RC_VALUE);
977
978         // Set the prime size for instrumentation purposes
979         INSTRUMENT_SET(PrimeIndex, PRIME_INDEX(keySizeInBits / 2));
980
981     #if SIMULATION && USE_RSA_KEY_CACHE
982         if(GET_CACHED_KEY(rsaKey, rand))
983             return TPM_RC_SUCCESS;
984     #endif
985
986         // Make sure that key generation has been tested
987         TEST(ALG_NULL_VALUE);
988
989         // Need to initialize the privateExponent structure
990         RsaInitializeExponent(&rsaKey->privateExponent);
991
992         // The prime is computed in P. When a new prime is found, Q is checked to
993         // see if it is zero.  If so, P is copied to Q and a new P is found.
994         // When both P and Q are non-zero, the modulus and
995         // private exponent are computed and a trial encryption/decryption is
996         // performed.  If the encrypt/decrypt fails, assume that at least one of the
997         // primes is composite. Since we don't know which one, set Q to zero and start
998         // over and find a new pair of primes.
999
1000        for(i = 1; (retVal != TPM_RC_SUCCESS) && (i != 100); i++)
1001        {
1002            if(_plat__IsCanceled())
1003                ERROR_RETURN(TPM_RC_CANCELED);
1004
1005            BnGeneratePrimeForRSA(bnP, keySizeInBits / 2, e, rand);
1006            INSTRUMENT_INC(PrimeCounts[PrimeIndex]);
1007
1008            // If this is the second prime, make sure that it differs from the
1009            // first prime by at least 2^100
1010            if(BnEqualZero(bnQ))
1011            {
1012                // copy p to q and compute another prime in p
1013                BnCopy(bnQ, bnP);
1014                continue;
1015            }
1016            // Make sure that the difference is at least 100 bits. Need to do it this
```



```
1017            // way because the big numbers are only positive values
1018            if(BnUnsignedCmp(bnP, bnQ) < 0)
1019                BnSub(bnD, bnQ, bnP);
1020            else
1021                BnSub(bnD, bnP, bnQ);
1022            if(BnMsb(bnD) < 100)
1023                continue;
1024
1025            //Form the public modulus and set the unique value
1026            BnMult(bnN, bnP, bnQ);
1027            BnTo2B(bnN, &publicArea->unique.rsa.b,
1028                   (NUMBYTES)BITS_TO_BYTES(keySizeInBits));
1029
1030            // And the  prime to the sensitive area
1031            BnTo2B(bnP, &sensitive->sensitive.rsa.b,
1032                   (NUMBYTES)BITS_TO_BYTES(keySizeInBits) / 2);
1033
1034            // Make sure everything came out right. The MSb of the values must be
1035            // one
1036            if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1037               || ((sensitive->sensitive.rsa.t.buffer[0] & 0x80) == 0))
1038                FAIL(FATAL_ERROR_INTERNAL);
1039
1040            // Make sure that we can form the private exponent values
1041            if(ComputePrivateExponent(bnP, bnQ, bnE, bnN, &rsaKey->privateExponent)
1042               != TRUE)
1043            {
1044                // If ComputePrivateExponent could not find an inverse for
1045                // Q, then copy P and recompute P. This might
1046                // cause both to be recomputed if P is also zero
1047                if(BnEqualZero(bnQ))
1048                    BnCopy(bnQ, bnP);
1049                continue;
1050            }
1051            retVal = TPM_RC_SUCCESS;
1052            // Do a trial encryption decryption if this is a signing key
1053            if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
1054            {
1055                BN_RSA(temp1);
1056                BN_RSA(temp2);
1057                BnGenerateRandomInRange(temp1, bnN, rand);
1058
1059                // Encrypt with public exponent...
1060                BnModExp(temp2, temp1, bnE, bnN);
1061                // ...  then decrypt with private exponent
1062                RsaPrivateKeyOp(temp2, bnN, bnP, &rsaKey->privateExponent);
1063
1064                // If the starting and ending values are not the same,
1065                // start over )-;
1066                if(BnUnsignedCmp(temp2, temp1) != 0)
1067                {
1068                    BnSetWord(bnQ, 0);
1069                    retVal = TPM_RC_NO_RESULT;
1070                }
1071            }
1072        }
1073    Exit:
1074        if(retVal == TPM_RC_SUCCESS)
1075            rsaKey->attributes.privateExp = SET;
1076        return retVal;
1077    }
1078    #endif // ALG_RSA
```

10.2.19 CryptSmac.c

10.2.19.1 Introduction

This file contains the implementation of the message authentication codes based on a symmetric block cipher. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.2.19.2 Includes, Defines, and Typedefs

```
1   #define _CRYPT_HASH_C_
2   #include "Tpm.h"
3   #if SMAC_IMPLEMENTED
```

10.2.19.2.1 CryptSmacStart()

Function to start an SMAC.

```
4   UINT16
5   CryptSmacStart(
6       HASH_STATE              *state,
7       TPMU_PUBLIC_PARMS       *keyParameters,
8       TPM_ALG_ID               macAlg,          // IN: the type of MAC
9       TPM2B                   *key
10  )
11  {
12      UINT16                  retVal = 0;
13  //
14      // Make sure that the key size is correct. This should have been checked
15      // at key load, but...
16      if(BITS_TO_BYTES(keyParameters->symDetail.sym.keyBits.sym) == key->size)
17      {
18          switch(macAlg)
19          {
20  #if     ALG_CMAC
21              case ALG_CMAC_VALUE:
22                  retVal = CryptCmacStart(&state->state.smac, keyParameters,
23                                          macAlg, key);
24                  break;
25  #endif
26              default:
27                  break;
28          }
29      }
30      state->type = (retVal != 0) ? HASH_STATE_SMAC : HASH_STATE_EMPTY;
31      return retVal;
32  }
```

10.2.19.2.2 CryptMacStart()

Function to start either an HMAC or an SMAC. Cannot reuse the CryptHmacStart() function because of the difference in number of parameters.

```
33  UINT16
34  CryptMacStart(
35      HMAC_STATE              *state,
36      TPMU_PUBLIC_PARMS       *keyParameters,
37      TPM_ALG_ID               macAlg,          // IN: the type of MAC
38      TPM2B                   *key
```



```
39  )
40  {
41      MemorySet(state, 0, sizeof(HMAC_STATE));
42      if(CryptHashIsValidAlg(macAlg, FALSE))
43      {
44          return CryptHmacStart(state, macAlg, key->size, key->buffer);
45      }
46      else if(CryptSmacIsValidAlg(macAlg, FALSE))
47      {
48          return CryptSmacStart(&state->hashState, keyParameters, macAlg, key);
49      }
50      else
51          return 0;
52  }
```

10.2.19.2.3 CryptMacEnd()

Dispatch to the MAC end function using a size and buffer pointer.

```
53  UINT16
54  CryptMacEnd(
55      HMAC_STATE          *state,
56      UINT32               size,
57      BYTE                *buffer
58  )
59  {
60      UINT16              retVal = 0;
61      if(state->hashState.type == HASH_STATE_SMAC)
62          retVal = (state->hashState.state.smac.smacMethods.end)(
63                      &state->hashState.state.smac.state, size, buffer);
64      else if(state->hashState.type == HASH_STATE_HMAC)
65          retVal = CryptHmacEnd(state, size, buffer);
66      state->hashState.type = HASH_STATE_EMPTY;
67      return retVal;
68  }
```

10.2.19.2.4 CryptMacEnd2B()

Dispatch to the MAC end function using a 2B.

```
69  UINT16
70  CryptMacEnd2B (
71      HMAC_STATE          *state,
72      TPM2B               *data
73  )
74  {
75      return CryptMacEnd(state, data->size, data->buffer);
76  }
77  #endif // SMAC_IMPLEMENTED
 9
```

10.2.20 CryptSym.c

10.2.20.1 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.

10.2.20.2 Includes, Defines, and Typedefs

```
#include "Tpm.h"
#include "CryptSym.h"
```

10.2.20.3 Initialization and Data Access Functions

10.2.20.3.1 CryptSymInit()

This function is called to do _TPM_Init() processing

```
BOOL
CryptSymInit(
    void
    )
{
    return TRUE;
}
```

10.2.20.3.2 CryptSymStartup()

This function is called to do TPM2_Startup() processing

```
BOOL
CryptSymStartup(
    void
    )
{
    return TRUE;
}
```

10.2.20.3.3 CryptGetSymmetricBlockSize()

This function returns the block size of the algorithm.

Return Value	Meaning
<= 0	cipher not supported
> 0	the cipher block size in bytes

```
LIB_EXPORT INT16
CryptGetSymmetricBlockSize(
    TPM_ALG_ID      symmetricAlg,   // IN: the symmetric algorithm
    UINT16          keySizeInBits   // IN: the key size
    )
{
    switch(symmetricAlg)
    {
#if     ALG_AES
```



```
26          case ALG_AES_VALUE:
27              switch(keySizeInBits)
28              {
29                  case 128:
30                      return AES_128_BLOCK_SIZE_BYTES;
31                  case 192:
32                      return AES_192_BLOCK_SIZE_BYTES;
33                  case 256:
34                      return AES_256_BLOCK_SIZE_BYTES;
35                  default:
36                      break;
37              }
38              break;
39  #endif
40  #if     ALG_SM4
41          case ALG_SM4_VALUE:
42              switch(keySizeInBits)
43              {
44                  case 128:
45                      return SM4_128_BLOCK_SIZE_BYTES;
46                  default:
47                      break;
48              }
49  #endif
50  #if     ALG_CAMELLIA
51          case ALG_CAMELLIA_VALUE:
52              switch(keySizeInBits)
53              {
54                  case 128:
55                      return CAMELLIA_128_BLOCK_SIZE_BYTES;
56                  case 192:
57                      return CAMELLIA_192_BLOCK_SIZE_BYTES;
58                  case 256:
59                      return CAMELLIA_256_BLOCK_SIZE_BYTES;
60                  default:
61                      break;
62              }
63  #endif
64  #if     ALG_TDES
65          case ALG_TDES_VALUE:
66              switch(keySizeInBits)
67              {
68                  case 128:
69                      return TDES_128_BLOCK_SIZE_BYTES;
70                  case 192:
71                      return TDES_192_BLOCK_SIZE_BYTES;
72                  default:
73                      break;
74              }
75  #endif
76          default:
77              break;
78      }
79      return 0;
80  }
```

10.2.20.4 Symmetric Encryption

This function performs symmetric encryption based on the mode.

Error Returns	Meaning
TPM_RC_SIZE	*dSize* is not a multiple of the block size for an algorithm that requires it
TPM_RC_FAILURE	Fatal error

```
81  LIB_EXPORT TPM_RC
82  CryptSymmetricEncrypt(
83      BYTE                *dOut,          // OUT:
84      TPM_ALG_ID           algorithm,     // IN: the symmetric algorithm
85      UINT16               keySizeInBits, // IN: key size in bits
86      const BYTE          *key,           // IN: key buffer. The size of this buffer
87                                          //     in bytes is (keySizeInBits + 7) / 8
88      TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption.
89      TPM_ALG_ID           mode,          // IN: Mode to use
90      INT32                dSize,         // IN: data size (may need to be a
91                                          //     multiple of the blockSize)
92      const BYTE          *dIn            // IN: data buffer
93      )
94  {
95      BYTE                *pIv;
96      int                  i;
97      BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
98      BYTE                *pT;
99      tpmCryptKeySchedule_t        keySchedule;
100     INT16                blockSize;
101     TpmCryptSetSymKeyCall_t        encrypt;
102     BYTE                *iv;
103     BYTE                 defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
104 //
105     pAssert(dOut != NULL && key != NULL && dIn != NULL);
106     if(dSize == 0)
107         return TPM_RC_SUCCESS;
108
109     TEST(algorithm);
110     blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
111     if(blockSize == 0)
112         return TPM_RC_FAILURE;
113     // If the iv is provided, then it is expected to be block sized. In some cases,
114     // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
115     // with no knowledge of the actual block size. This function will set it.
116     if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
117     {
118         ivInOut->t.size = blockSize;
119         iv = ivInOut->t.buffer;
120     }
121     else
122         iv = defaultIv;
123     pIv = iv;
124
125     // Create encrypt key schedule and set the encryption function pointer.
126
127     SELECT(ENCRYPT);
128
129     switch(mode)
130     {
131 #if     ALG_CTR
132         case ALG_CTR_VALUE:
133             for(; dSize > 0; dSize -= blockSize)
134             {
135                 // Encrypt the current value of the IV(counter)
136                 ENCRYPT(&keySchedule, iv, tmp);
137
138                 //increment the counter (counter is big-endian so start at end)
```



```
139                 for(i = blockSize - 1; i >= 0; i--)
140                     if((iv[i] += 1) != 0)
141                         break;
142
143                 // XOR the encrypted counter value with input and put into output
144                 pT = tmp;
145                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
146                     *dOut++ = *dIn++ ^ *pT++;
147             }
148             break;
149 #endif
150 #if     ALG_OFB
151         case ALG_OFB_VALUE:
152             // This is written so that dIn and dOut may be the same
153             for(; dSize > 0; dSize -= blockSize)
154             {
155                 // Encrypt the current value of the "IV"
156                 ENCRYPT(&keySchedule, iv, iv);
157
158                 // XOR the encrypted IV into dIn to create the cipher text (dOut)
159                 pIv = iv;
160                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
161                     *dOut++ = (*pIv++ ^ *dIn++);
162             }
163             break;
164 #endif
165 #if     ALG_CBC
166         case ALG_CBC_VALUE:
167             // For CBC the data size must be an even multiple of the
168             // cipher block size
169             if((dSize % blockSize) != 0)
170                 return TPM_RC_SIZE;
171             // XOR the data block into the IV, encrypt the IV into the IV
172             // and then copy the IV to the output
173             for(; dSize > 0; dSize -= blockSize)
174             {
175                 pIv = iv;
176                 for(i = blockSize; i > 0; i--)
177                     *pIv++ ^= *dIn++;
178                 ENCRYPT(&keySchedule, iv, iv);
179                 pIv = iv;
180                 for(i = blockSize; i > 0; i--)
181                     *dOut++ = *pIv++;
182             }
183             break;
184 #endif
185         // CFB is not optional
186         case ALG_CFB_VALUE:
187             // Encrypt the IV into the IV, XOR in the data, and copy to output
188             for(; dSize > 0; dSize -= blockSize)
189             {
190                 // Encrypt the current value of the IV
191                 ENCRYPT(&keySchedule, iv, iv);
192                 pIv = iv;
193                 for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
194                     // XOR the data into the IV to create the cipher text
195                     // and put into the output
196                     *dOut++ = *pIv++ ^= *dIn++;
197             }
198             // If the inner loop (i loop) was smaller than blockSize, then dSize
199             // would have been smaller than blockSize and it is now negative. If
200             // it is negative, then it indicates how many bytes are needed to pad
201             // out the IV for the next round.
202             for(; dSize < 0; dSize++)
203                 *pIv++ = 0;
204             break;
```

```
205 #if     ALG_ECB
206         case ALG_ECB_VALUE:
207             // For ECB the data size must be an even multiple of the
208             // cipher block size
209             if((dSize % blockSize) != 0)
210                 return TPM_RC_SIZE;
211             // Encrypt the input block to the output block
212             for(; dSize > 0; dSize -= blockSize)
213             {
214                 ENCRYPT(&keySchedule, dIn, dOut);
215                 dIn = &dIn[blockSize];
216                 dOut = &dOut[blockSize];
217             }
218             break;
219 #endif
220         default:
221             return TPM_RC_FAILURE;
222     }
223     return TPM_RC_SUCCESS;
224 }
```

10.2.20.4.1 CryptSymmetricDecrypt()

This function performs symmetric decryption based on the mode.

Error Returns	Meaning
TPM_RC_FAILURE	A fatal error
TPM_RCS_SIZE	*dSize* is not a multiple of the block size for an algorithm that requires it

```
225 LIB_EXPORT TPM_RC
226 CryptSymmetricDecrypt(
227     BYTE                *dOut,          // OUT: decrypted data
228     TPM_ALG_ID           algorithm,     // IN: the symmetric algorithm
229     UINT16               keySizeInBits, // IN: key size in bits
230     const BYTE          *key,           // IN: key buffer. The size of this buffer
231                                         //     in bytes is (keySizeInBits + 7) / 8
232     TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption.
233     TPM_ALG_ID           mode,          // IN: Mode to use
234     INT32                dSize,         // IN: data size (may need to be a
235                                         //     multiple of the blockSize)
236     const BYTE          *dIn            // IN: data buffer
237     )
238 {
239     BYTE                *pIv;
240     int                  i;
241     BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
242     BYTE                *pT;
243     tpmCryptKeySchedule_t        keySchedule;
244     INT16                blockSize;
245     BYTE                *iv;
246     TpmCryptSetSymKeyCall_t        encrypt;
247     TpmCryptSetSymKeyCall_t        decrypt;
248     BYTE                 defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
249
250     // These are used but the compiler can't tell because they are initialized
251     // in case statements and it can't tell if they are always initialized
252     // when needed, so... Comment these out if the compiler can tell or doesn't
253     // care that these are initialized before use.
254     encrypt = NULL;
255     decrypt = NULL;
256
```



```
257     pAssert(dOut != NULL && key != NULL && dIn != NULL);
258     if(dSize == 0)
259         return TPM_RC_SUCCESS;
260
261     TEST(algorithm);
262     blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
263     if(blockSize == 0)
264         return TPM_RC_FAILURE;
265     // If the iv is provided, then it is expected to be block sized. In some cases,
266     // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
267     // with no knowledge of the actual block size. This function will set it.
268     if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
269     {
270         ivInOut->t.size = blockSize;
271         iv = ivInOut->t.buffer;
272     }
273     else
274         iv = defaultIv;
275
276     pIv = iv;
277     // Use the mode to select the key schedule to create. Encrypt always uses the
278     // encryption schedule. Depending on the mode, decryption might use either
279     // the decryption or encryption schedule.
280     switch(mode)
281     {
282 #if ALG_CBC || ALG_ECB
283         case ALG_CBC_VALUE: // decrypt = decrypt
284         case ALG_ECB_VALUE:
285             // For ECB and CBC, the data size must be an even multiple of the
286             // cipher block size
287             if((dSize % blockSize) != 0)
288                 return TPM_RC_SIZE;
289             SELECT(DECRYPT);
290             break;
291 #endif
292         default:
293             // For the remaining stream ciphers, use encryption to decrypt
294             SELECT(ENCRYPT);
295             break;
296     }
297     // Now do the mode-dependent decryption
298     switch(mode)
299     {
300 #if     ALG_CBC
301         case ALG_CBC_VALUE:
302             // Copy the input data to a temp buffer, decrypt the buffer into the
303             // output, XOR in the IV, and copy the temp buffer to the IV and repeat.
304             for(; dSize > 0; dSize -= blockSize)
305             {
306                 pT = tmp;
307                 for(i = blockSize; i > 0; i--)
308                     *pT++ = *dIn++;
309                 DECRYPT(&keySchedule, tmp, dOut);
310                 pIv = iv;
311                 pT = tmp;
312                 for(i = blockSize; i > 0; i--)
313                 {
314                     *dOut++ ^= *pIv;
315                     *pIv++ = *pT++;
316                 }
317             }
318             break;
319 #endif
320         case ALG_CFB_VALUE:
321             for(; dSize > 0; dSize -= blockSize)
322             {
```

```
323                     // Encrypt the IV into the temp buffer
324                     ENCRYPT(&keySchedule, iv, tmp);
325                     pT = tmp;
326                     pIv = iv;
327                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
328                         // Copy the current cipher text to IV, XOR
329                         // with the temp buffer and put into the output
330                         *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
331                 }
332                 // If the inner loop (i loop) was smaller than blockSize, then dSize
333                 // would have been smaller than blockSize and it is now negative
334                 // If it is negative, then it indicates how may fill bytes
335                 // are needed to pad out the IV for the next round.
336                 for(; dSize < 0; dSize++)
337                     *pIv++ = 0;
338
339                 break;
340     #if     ALG_CTR
341             case ALG_CTR_VALUE:
342                 for(; dSize > 0; dSize -= blockSize)
343                 {
344                     // Encrypt the current value of the IV(counter)
345                     ENCRYPT(&keySchedule, iv, tmp);
346
347                     //increment the counter (counter is big-endian so start at end)
348                     for(i = blockSize - 1; i >= 0; i--)
349                         if((iv[i] += 1) != 0)
350                             break;
351                     // XOR the encrypted counter value with input and put into output
352                     pT = tmp;
353                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
354                         *dOut++ = *dIn++ ^ *pT++;
355                 }
356                 break;
357     #endif
358     #if     ALG_ECB
359             case ALG_ECB_VALUE:
360                 for(; dSize > 0; dSize -= blockSize)
361                 {
362                     DECRYPT(&keySchedule, dIn, dOut);
363                     dIn = &dIn[blockSize];
364                     dOut = &dOut[blockSize];
365                 }
366                 break;
367     #endif
368     #if     ALG_OFB
369             case ALG_OFB_VALUE:
370                 // This is written so that dIn and dOut may be the same
371                 for(; dSize > 0; dSize -= blockSize)
372                 {
373                     // Encrypt the current value of the "IV"
374                     ENCRYPT(&keySchedule, iv, iv);
375
376                     // XOR the encrypted IV into dIn to create the cipher text (dOut)
377                     pIv = iv;
378                     for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
379                         *dOut++ = (*pIv++ ^ *dIn++);
380                 }
381                 break;
382     #endif
383             default:
384                 return TPM_RC_FAILURE;
385         }
386         return TPM_RC_SUCCESS;
387     }
```


10.2.20.4.2 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

```
388 TPM_RC
389 CryptSymKeyValidate(
390     TPMT_SYM_DEF_OBJECT *symDef,
391     TPM2B_SYM_KEY       *key
392     )
393 {
394     if(key->t.size != BITS_TO_BYTES(symDef->keyBits.sym))
395         return TPM_RCS_KEY_SIZE;
396 #if     ALG_TDES
397     if(symDef->algorithm == TPM_ALG_TDES && !CryptDesValidateKey(key))
398         return TPM_RCS_KEY;
399 #endif // ALG_TDES
400     return TPM_RC_SUCCESS;
401 }
```

10.2.21 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 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,
```



```
0x01, 0x40, 0x02, 0x28, 0x40, 0xa4, 0x04, 0x92, 0x30, 0x04, 0x11, 0x86,
0x08, 0x42, 0x00, 0x2c, 0x52, 0x04, 0x08, 0xc9, 0x84, 0x60, 0x48, 0x12,
0x09, 0x99, 0x24, 0x44, 0x00, 0x24, 0x00, 0x03, 0x14, 0x21, 0x00, 0x10,
0x01, 0x1a, 0x32, 0x05, 0x88, 0x20, 0x40, 0x40, 0x06, 0x09, 0xc3, 0x84,
0x40, 0x01, 0x30, 0x60, 0x18, 0x02, 0x68, 0x11, 0x90, 0x0c, 0x02, 0xa2,
0x04, 0x00, 0x86, 0x29, 0x89, 0x14, 0x24, 0x82, 0x02, 0x41, 0x08, 0x80,
0x04, 0x19, 0x80, 0x08, 0x10, 0x12, 0x68, 0x42, 0xa4, 0x04, 0x00, 0x02,
0x61, 0x10, 0x06, 0x0c, 0x10, 0x00, 0x01, 0x12, 0x10, 0x20, 0x03, 0x94,
0x21, 0x42, 0x12, 0x65, 0x18, 0x94, 0x0c, 0x0a, 0x04, 0x28, 0x01, 0x14,
0x29, 0x0a, 0xa4, 0x40, 0xd0, 0x00, 0x40, 0x01, 0x90, 0x04, 0x41, 0x20,
0x2d, 0x40, 0x82, 0x48, 0xc1, 0x20, 0x00, 0x10, 0x30, 0x01, 0x08, 0x24,
0x04, 0x59, 0x84, 0x24, 0x00, 0x02, 0x29, 0x82, 0x00, 0x61, 0x58, 0x02,
0x48, 0x81, 0x16, 0x48, 0x10, 0x00, 0x21, 0x11, 0x06, 0x00, 0xca, 0xa0,
0x40, 0x02, 0x00, 0x04, 0x91, 0xb0, 0x00, 0x42, 0x04, 0x0c, 0x81, 0x06,
0x09, 0x48, 0x14, 0x25, 0x92, 0x20, 0x25, 0x11, 0xa0, 0x00, 0x0a, 0x86,
0x0c, 0xc1, 0x02, 0x48, 0x00, 0x20, 0x45, 0x08, 0x32, 0x00, 0x98, 0x06,
0x04, 0x13, 0x22, 0x00, 0x82, 0x04, 0x48, 0x81, 0x14, 0x44, 0x82, 0x12,
0x24, 0x18, 0x10, 0x40, 0x43, 0x80, 0x28, 0xd0, 0x04, 0x20, 0x81, 0x24,
0x64, 0xd8, 0x00, 0x2c, 0x09, 0x12, 0x08, 0x41, 0xa2, 0x00, 0x00, 0x02,
0x41, 0xca, 0x20, 0x41, 0xc0, 0x10, 0x01, 0x18, 0xa4, 0x04, 0x18, 0xa4,
0x20, 0x12, 0x94, 0x20, 0x83, 0xa0, 0x40, 0x02, 0x32, 0x44, 0x80, 0x04,
0x00, 0x18, 0x00, 0x0c, 0x40, 0x86, 0x60, 0x8a, 0x00, 0x64, 0x88, 0x12,
0x05, 0x01, 0x82, 0x00, 0x4a, 0xa2, 0x01, 0xc1, 0x10, 0x61, 0x09, 0x04,
0x01, 0x88, 0x00, 0x60, 0x01, 0xb4, 0x40, 0x08, 0x06, 0x01, 0x03, 0x80,
0x08, 0x40, 0x94, 0x04, 0x8a, 0x20, 0x29, 0x80, 0x02, 0x0c, 0x52, 0x02,
0x01, 0x42, 0x84, 0x00, 0x80, 0x84, 0x64, 0x02, 0x32, 0x48, 0x00, 0x30,
0x44, 0x40, 0x22, 0x21, 0x00, 0x02, 0x08, 0xc3, 0xa0, 0x04, 0xd0, 0x20,
0x40, 0x18, 0x16, 0x40, 0x40, 0x00, 0x28, 0x52, 0x90, 0x08, 0x82, 0x14,
0x01, 0x18, 0x10, 0x08, 0x09, 0x82, 0x40, 0x0a, 0xa0, 0x20, 0x93, 0x80,
0x08, 0xc0, 0x00, 0x20, 0x52, 0x00, 0x05, 0x01, 0x10, 0x40, 0x11, 0x06,
0x0c, 0x82, 0x00, 0x00, 0x4b, 0x90, 0x44, 0x9a, 0x00, 0x28, 0x80, 0x90,
0x04, 0x4a, 0x06, 0x09, 0x43, 0x02, 0x28, 0x00, 0x34, 0x01, 0x18, 0x00,
0x65, 0x09, 0x80, 0x44, 0x03, 0x00, 0x24, 0x02, 0x82, 0x61, 0x48, 0x14,
0x41, 0x00, 0x12, 0x28, 0x00, 0x34, 0x08, 0x51, 0x04, 0x05, 0x12, 0x90,
0x28, 0x89, 0x84, 0x60, 0x12, 0x10, 0x49, 0x10, 0x26, 0x40, 0x49, 0x82,
0x00, 0x91, 0x10, 0x01, 0x0a, 0x24, 0x40, 0x88, 0x10, 0x4c, 0x10, 0x04,
0x00, 0x50, 0xa2, 0x2c, 0x40, 0x90, 0x48, 0x0a, 0xb0, 0x01, 0x50, 0x12,
0x08, 0x00, 0xa4, 0x04, 0x09, 0xa0, 0x28, 0x92, 0x02, 0x00, 0x43, 0x10,
0x21, 0x02, 0x20, 0x41, 0x81, 0x32, 0x00, 0x08, 0x04, 0x0c, 0x52, 0x00,
0x21, 0x49, 0x84, 0x20, 0x10, 0x02, 0x01, 0x81, 0x10, 0x48, 0x40, 0x22,
0x01, 0x01, 0x84, 0x69, 0xc1, 0x30, 0x01, 0xc8, 0x02, 0x44, 0x88, 0x00,
0x0c, 0x01, 0x02, 0x2d, 0xc0, 0x12, 0x61, 0x00, 0xa0, 0x00, 0xc0, 0x30,
0x40, 0x01, 0x12, 0x08, 0x0b, 0x20, 0x00, 0x80, 0x94, 0x40, 0x01, 0x84,
0x40, 0x00, 0x32, 0x00, 0x10, 0x84, 0x00, 0x0b, 0x24, 0x00, 0x01, 0x06,
0x29, 0x8a, 0x84, 0x41, 0x80, 0x10, 0x08, 0x08, 0x94, 0x4c, 0x03, 0x80,
0x01, 0x40, 0x96, 0x40, 0x41, 0x20, 0x20, 0x50, 0x22, 0x25, 0x89, 0xa2,
0x40, 0x40, 0xa4, 0x20, 0x02, 0x86, 0x28, 0x01, 0x20, 0x21, 0x4a, 0x10,
0x08, 0x00, 0x14, 0x08, 0x40, 0x04, 0x25, 0x42, 0x02, 0x21, 0x43, 0x10,
0x04, 0x92, 0x00, 0x21, 0x11, 0xa0, 0x4c, 0x18, 0x22, 0x09, 0x03, 0x84,
0x41, 0x89, 0x10, 0x04, 0x82, 0x22, 0x24, 0x01, 0x14, 0x08, 0x08, 0x84,
0x08, 0xc1, 0x00, 0x09, 0x42, 0xb0, 0x41, 0x8a, 0x02, 0x00, 0x80, 0x36,
0x04, 0x49, 0xa0, 0x24, 0x91, 0x00, 0x00, 0x02, 0x94, 0x41, 0x92, 0x02,
0x01, 0x08, 0x06, 0x08, 0x09, 0x00, 0x01, 0xd0, 0x16, 0x28, 0x89, 0x80,
0x60, 0x00, 0x00, 0x68, 0x01, 0x90, 0x0c, 0x50, 0x20, 0x01, 0x40, 0x80,
0x40, 0x42, 0x30, 0x41, 0x00, 0x20, 0x25, 0x81, 0x06, 0x40, 0x49, 0x00,
0x08, 0x01, 0x12, 0x49, 0x00, 0xa0, 0x20, 0x18, 0x30, 0x05, 0x01, 0xa6,
0x00, 0x10, 0x24, 0x28, 0x00, 0x02, 0x20, 0xc8, 0x20, 0x00, 0x88, 0x12,
0x0c, 0x90, 0x92, 0x00, 0x02, 0x26, 0x01, 0x42, 0x16, 0x49, 0x00, 0x04,
0x24, 0x42, 0x02, 0x01, 0x88, 0x80, 0x0c, 0x1a, 0x80, 0x08, 0x10, 0x00,
0x60, 0x02, 0x94, 0x44, 0x88, 0x00, 0x69, 0x11, 0x30, 0x08, 0x12, 0xa0,
0x24, 0x13, 0x84, 0x00, 0x82, 0x00, 0x65, 0xc0, 0x10, 0x28, 0x00, 0x30,
0x04, 0x03, 0x20, 0x01, 0x11, 0x06, 0x01, 0xc8, 0x80, 0x00, 0xc2, 0x20,
0x08, 0x10, 0x82, 0x0c, 0x13, 0x02, 0x0c, 0x52, 0x06, 0x40, 0x00, 0xb0,
0x61, 0x40, 0x10, 0x01, 0x98, 0x86, 0x04, 0x10, 0x84, 0x08, 0x92, 0x14,
0x60, 0x41, 0x80, 0x41, 0x1a, 0x10, 0x04, 0x81, 0x22, 0x40, 0x41, 0x20,
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 RSA_KEY_SIEVE && SIMULATION && RSA_INSTRUMENT
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
```


10.2.22 RsaKeyCache.c

10.2.22.1 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.

10.2.22.2 Includes, Types, Locals, and Defines

```
#include "Tpm.h"
#if 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
```



```
32  #ifdef RSA_KEY_SIZE_BITS_4096
33  #define RSA_4096    YES
34  #else
35  #define RSA_4096    NO
36  #endif
37  #define comma
38  TPMI_RSA_KEY_BITS       SupportedRsaKeySizes[] = {
39  #if RSA_1024
40      1024
41  #   undef comma
42  #   define comma ,
43  #endif
44  #if RSA_2048
45      comma 2048
46  #   undef comma
47  #   define comma ,
48  #endif
49  #if RSA_3072
50      comma 3072
51  #   undef comma
52  #   define comma ,
53  #endif
54  #if RSA_4096
55      comma 4096
56  #endif
57  };
58  #define RSA_KEY_CACHE_ENTRIES (RSA_1024 + RSA_2048 + RSA_3072 + RSA_4096)
```

The key cache holds one entry for each of the supported key sizes

```
59  RSA_KEY_CACHE        s_rsaKeyCache[RSA_KEY_CACHE_ENTRIES];
```

Indicates if the key cache is loaded. It can be loaded and enabled or disabled.

```
60  BOOL                 s_keyCacheLoaded = 0;
```

Indicates if the key cache is enabled

```
61  int                  s_rsaKeyCacheEnabled = FALSE;
```

10.2.22.2.1 RsaKeyCacheControl()

Used to enable and disable the RSA key cache.

```
62  LIB_EXPORT void
63  RsaKeyCacheControl(
64      int             state
65      )
66  {
67      s_rsaKeyCacheEnabled = state;
68  }
```

10.2.22.2.2 InitializeKeyCache()

This will initialize the key cache and attempt to write it to a file for later use.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
69  static BOOL
70  InitializeKeyCache(
71      OBJECT              *rsaKey,            // IN/OUT: The object structure in which
72                                              //          the key is created.
73      RAND_STATE          *rand               // IN: if not NULL, the deterministic
74                                              //     RNG state
75      )
76  {
77      int                  index;
78      TPM_KEY_BITS         keySave = rsaKey->publicArea.parameters.rsaDetail.keyBits;
79      BOOL                 OK = TRUE;
80  //
81      s_rsaKeyCacheEnabled = FALSE;
82      for(index = 0; OK && index < RSA_KEY_CACHE_ENTRIES; index++)
83      {
84          rsaKey->publicArea.parameters.rsaDetail.keyBits
85              = SupportedRsaKeySizes[index];
86          OK = (CryptRsaGenerateKey(rsaKey, rand) == TPM_RC_SUCCESS);
87          if(OK)
88          {
89              s_rsaKeyCache[index].publicModulus = rsaKey->publicArea.unique.rsa;
90              s_rsaKeyCache[index].privatePrime = rsaKey->sensitive.sensitive.rsa;
91              s_rsaKeyCache[index].privateExponent = rsaKey->privateExponent;
92          }
93      }
94      rsaKey->publicArea.parameters.rsaDetail.keyBits = keySave;
95      s_keyCacheLoaded = OK;
96  #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
97      if(OK)
98      {
99          FILE                *cacheFile;
100         const char          *fn = CACHE_FILE_NAME;
101
102 #if defined _MSC_VER
103         if(fopen_s(&cacheFile, fn, "w+b") != 0)
104 #else
105         cacheFile = fopen(fn, "w+b");
106         if(NULL == cacheFile)
107 #endif
108         {
109             printf("Can't open %s for write.\n", fn);
110         }
111         else
112         {
113             fseek(cacheFile, 0, SEEK_SET);
114             if(fwrite(s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
115                != sizeof(s_rsaKeyCache))
116             {
117                 printf("Error writing cache to %s.", fn);
118             }
119         }
120         if(cacheFile)
121             fclose(cacheFile);
122     }
123 #endif
124     return s_keyCacheLoaded;
125 }
```


10.2.22.2.3 KeyCacheLoaded()

Checks that key cache is loaded.

Return Value	Meaning
TRUE(1)	cache loaded
FALSE(0)	cache not loaded

```
126  static BOOL
127  KeyCacheLoaded(
128      OBJECT              *rsaKey,            // IN/OUT: The object structure in which
129                                              //          the key is created.
130      RAND_STATE          *rand               // IN: if not NULL, the deterministic
131                                              //     RNG state
132      )
133  {
134  #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
135      if(!s_keyCacheLoaded)
136      {
137          FILE            *cacheFile;
138          const char *     fn = CACHE_FILE_NAME;
139  #if defined _MSC_VER && 1
140          if(fopen_s(&cacheFile, fn, "r+b") == 0)
141  #else
142          cacheFile = fopen(fn, "r+b");
143          if(NULL != cacheFile)
144  #endif
145          {
146              fseek(cacheFile, 0L, SEEK_END);
147              if(ftell(cacheFile) == sizeof(s_rsaKeyCache))
148              {
149                  fseek(cacheFile, 0L, SEEK_SET);
150                  s_keyCacheLoaded = (
151                      fread(&s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
152                      == sizeof(s_rsaKeyCache));
153              }
154              fclose(cacheFile);
155          }
156      }
157  #endif
158      if(!s_keyCacheLoaded)
159          s_rsaKeyCacheEnabled = InitializeKeyCache(rsaKey, rand);
160      return s_keyCacheLoaded;
161  }
```

10.2.22.2.4 GetCachedRsaKey()

Return Value	Meaning
TRUE(1)	key loaded
FALSE(0)	key not loaded

```
162  BOOL
163  GetCachedRsaKey(
164      OBJECT              *key,
165      RAND_STATE          *rand               // IN: if not NULL, the deterministic
166                                              //     RNG state
167      )
168  {
169      int                     keyBits = key->publicArea.parameters.rsaDetail.keyBits;
```

```
170     int                     index;
171 //
172     if(KeyCacheLoaded(key, rand))
173     {
174         for(index = 0; index < RSA_KEY_CACHE_ENTRIES; index++)
175         {
176             if((s_rsaKeyCache[index].publicModulus.t.size * 8) == keyBits)
177             {
178                 key->publicArea.unique.rsa = s_rsaKeyCache[index].publicModulus;
179                 key->sensitive.sensitive.rsa = s_rsaKeyCache[index].privatePrime;
180                 key->privateExponent = s_rsaKeyCache[index].privateExponent;
181                 key->attributes.privateExp = SET;
182                 return TRUE;
183             }
184         }
185         return FALSE;
186     }
187     return s_keyCacheLoaded;
188 }
189 #endif  // defined SIMULATION && defined USE_RSA_KEY_CACHE
```

10.2.23 Ticket.c

10.2.23.1 Introduction

This clause contains the functions used for ticket computations.

10.2.23.2 Includes

```
#include "Tpm.h"
```

10.2.23.3 Functions

10.2.23.3.1 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(1)	safe to produce ticket
FALSE(0)	not safe to produce ticket

```
BOOL
TicketIsSafe(
    TPM2B           *buffer
    )
{
    TPM_GENERATED   valueToCompare = TPM_GENERATED_VALUE;
    BYTE            bufferToCompare[sizeof(valueToCompare)];
    BYTE            *marshalBuffer;
//
    // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
    // it is not safe to generate a ticket
    if(buffer->size < sizeof(valueToCompare))
        return FALSE;
    marshalBuffer = bufferToCompare;
    TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);
    if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
        return FALSE;
    else
        return TRUE;
}
```

10.2.23.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```
void
TicketComputeVerified(
    TPMI_RH_HIERARCHY    hierarchy,     // IN: hierarchy constant for ticket
    TPM2B_DIGEST        *digest,        // IN: digest
    TPM2B_NAME          *keyName,       // IN: name of key that signed the values
    TPMT_TK_VERIFIED    *ticket         // OUT: verified ticket
    )
{
    TPM2B_PROOF         *proof;
    HMAC_STATE           hmacState;
```

```
32  //
33      // Fill in ticket fields
34      ticket->tag = TPM_ST_VERIFIED;
35      ticket->hierarchy = hierarchy;
36      proof = HierarchyGetProof(hierarchy);
37
38      // Start HMAC using the proof value of the hierarchy as the HMAC key
39      ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
40                                               &proof->b);
41          //  TPM_ST_VERIFIED
42      CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
43          //  digest
44      CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
45          // key name
46      CryptDigestUpdate2B(&hmacState.hashState, &keyName->b);
47          // done
48      CryptHmacEnd2B(&hmacState, &ticket->digest.b);
49
50      return;
51  }
```

10.2.23.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```
52  void
53  TicketComputeAuth(
54      TPM_ST                  type,           // IN: the type of ticket.
55      TPMI_RH_HIERARCHY       hierarchy,      // IN: hierarchy constant for ticket
56      UINT64                  timeout,        // IN: timeout
57      BOOL                    expiresOnReset, // IN: flag to indicate if ticket expires on
58                                              //       TPM Reset
59      TPM2B_DIGEST           *cpHashA,        // IN: input cpHashA
60      TPM2B_NONCE            *policyRef,      // IN: input policyRef
61      TPM2B_NAME             *entityName,     // IN: name of entity
62      TPMT_TK_AUTH           *ticket          // OUT: Created ticket
63      )
64  {
65      TPM2B_PROOF            *proof;
66      HMAC_STATE              hmacState;
67  //
68      // Get proper proof
69      proof = HierarchyGetProof(hierarchy);
70
71      // Fill in ticket fields
72      ticket->tag = type;
73      ticket->hierarchy = hierarchy;
74
75      // Start HMAC with hierarchy proof as the HMAC key
76      ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
77                                               &proof->b);
78          //  TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED,
79      CryptDigestUpdateInt(&hmacState, sizeof(UINT16), ticket->tag);
80      // cpHash
81      CryptDigestUpdate2B(&hmacState.hashState, &cpHashA->b);
82          //  policyRef
83      CryptDigestUpdate2B(&hmacState.hashState, &policyRef->b);
84          //  keyName
85      CryptDigestUpdate2B(&hmacState.hashState, &entityName->b);
86          //  timeout
87      CryptDigestUpdateInt(&hmacState, sizeof(timeout), timeout);
88      if(timeout != 0)
89      {
90              //  epoch
```

```
91          CryptDigestUpdateInt(&hmacState.hashState, sizeof(CLOCK_NONCE),
92                               g_timeEpoch);
93              // reset count
94          if(expiresOnReset)
95              CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
96                                   gp.totalResetCount);
97      }
98          // done
99      CryptHmacEnd2B(&hmacState, &ticket->digest.b);
100
101     return;
102 }
```

10.2.23.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```
103 void
104 TicketComputeHashCheck(
105     TPMI_RH_HIERARCHY    hierarchy,     // IN: hierarchy constant for ticket
106     TPM_ALG_ID           hashAlg,       // IN: the hash algorithm for 'digest'
107     TPM2B_DIGEST        *digest,        // IN: input digest
108     TPMT_TK_HASHCHECK   *ticket         // OUT: Created ticket
109     )
110 {
111     TPM2B_PROOF         *proof;
112     HMAC_STATE           hmacState;
113 //
114     // Get proper proof
115     proof = HierarchyGetProof(hierarchy);
116
117     // Fill in ticket fields
118     ticket->tag = TPM_ST_HASHCHECK;
119     ticket->hierarchy = hierarchy;
120
121     // Start HMAC using hierarchy proof as HMAC key
122     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
123                                              &proof->b);
124         //  TPM_ST_HASHCHECK
125     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
126         //  hash algorithm
127     CryptDigestUpdateInt(&hmacState, sizeof(hashAlg), hashAlg);
128         //  digest
129     CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
130         // done
131     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
132
133     return;
134 }
```

10.2.23.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```
135 void
136 TicketComputeCreation(
137     TPMI_RH_HIERARCHY    hierarchy,     // IN: hierarchy for ticket
138     TPM2B_NAME          *name,          // IN: object name
139     TPM2B_DIGEST        *creation,      // IN: creation hash
140     TPMT_TK_CREATION    *ticket         // OUT: created ticket
141     )
142 {
143     TPM2B_PROOF         *proof;
```

```
144     HMAC_STATE           hmacState;
145
146     // Get proper proof
147     proof = HierarchyGetProof(hierarchy);
148
149     // Fill in ticket fields
150     ticket->tag = TPM_ST_CREATION;
151     ticket->hierarchy = hierarchy;
152
153     // Start HMAC using hierarchy proof as HMAC key
154     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
155                                              &proof->b);
156         //  TPM_ST_CREATION
157     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
158         //  name if provided
159     if(name != NULL)
160         CryptDigestUpdate2B(&hmacState.hashState, &name->b);
161         //  creation hash
162     CryptDigestUpdate2B(&hmacState.hashState, &creation->b);
163         // Done
164     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
165
166     return;
167 }
```

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.

The file listed below was generated by an automated tool using three documents as inputs. They are:

a) The TCG_Algorithm Registery,

b) Part 2 of this specification, and

c) A purpose-built document that contains vendor-specific information in tables.

A.2 Implementation.h

```
1   #ifndef _IMPLEMENTATION_H_
2   #define _IMPLEMENTATION_H_
3   #include    "TpmBuildSwitches.h"
4   #include    "BaseTypes.h"
5   #include    "TPMB.h"
6   #undef TRUE
7   #undef FALSE
8   #ifndef MAX
9   #define MAX(a, b) ((a) > (b) ? (a) : (b))
10  #endif
11  #ifndef MIN
12  #define MIN(a, b) ((a) < (b) ? (a) : (b))
13  #endif
```

From TPM 2.0 Part 2: Table 4 - Defines for Logic Values

```
14  #define TRUE                1
15  #define FALSE               0
16  #define YES                 1
17  #define NO                  0
18  #define SET                 1
19  #define CLEAR               0
```

From Vendor-Specific: Table 1 - Defines for Processor Values

```
20  #ifndef BIG_ENDIAN_TPM
21  #define BIG_ENDIAN_TPM              NO
22  #endif // BIG_ENDIAN_TPM
23  #define LITTLE_ENDIAN_TPM           !BIG_ENDIAN_TPM
24  #ifndef MOST_SIGNIFICANT_BIT_0
25  #define MOST_SIGNIFICANT_BIT_0      NO
26  #endif // MOST_SIGNIFICANT_BIT_0
27  #define LEAST_SIGNIFICANT_BIT_0     !MOST_SIGNIFICANT_BIT_0
28  #ifndef AUTO_ALIGN
29  #define AUTO_ALIGN                  NO
30  #endif // AUTO_ALIGN
```

From Vendor-Specific: Table 3 - Defines for Key Size Constants

```
31  #define RSA_KEY_SIZES_BITS              {1024,2048}
```

```
32  #define RSA_KEY_SIZE_BITS_1024          RSA_ALLOWED_KEY_SIZE_1024
33  #define RSA_KEY_SIZE_BITS_2048          RSA_ALLOWED_KEY_SIZE_2048
34  #define MAX_RSA_KEY_BITS                2048
35  #define MAX_RSA_KEY_BYTES               256
36  #define TDES_KEY_SIZES_BITS             {128,192}
37  #define TDES_KEY_SIZE_BITS_128          TDES_ALLOWED_KEY_SIZE_128
38  #define TDES_KEY_SIZE_BITS_192          TDES_ALLOWED_KEY_SIZE_192
39  #define MAX_TDES_KEY_BITS               192
40  #define MAX_TDES_KEY_BYTES              24
41  #define MAX_TDES_BLOCK_SIZE_BYTES       \
42              MAX(TDES_128_BLOCK_SIZE_BYTES, MAX(TDES_192_BLOCK_SIZE_BYTES, 0))
43  #define AES_KEY_SIZES_BITS              {128,256}
44  #define AES_KEY_SIZE_BITS_128           AES_ALLOWED_KEY_SIZE_128
45  #define AES_KEY_SIZE_BITS_256           AES_ALLOWED_KEY_SIZE_256
46  #define MAX_AES_KEY_BITS                256
47  #define MAX_AES_KEY_BYTES               32
48  #define MAX_AES_BLOCK_SIZE_BYTES        \
49              MAX(AES_128_BLOCK_SIZE_BYTES, MAX(AES_256_BLOCK_SIZE_BYTES, 0))
50  #define SM4_KEY_SIZES_BITS              {128}
51  #define SM4_KEY_SIZE_BITS_128           SM4_ALLOWED_KEY_SIZE_128
52  #define MAX_SM4_KEY_BITS                128
53  #define MAX_SM4_KEY_BYTES               16
54  #define MAX_SM4_BLOCK_SIZE_BYTES        MAX(SM4_128_BLOCK_SIZE_BYTES, 0)
55  #define CAMELLIA_KEY_SIZES_BITS         {128}
56  #define CAMELLIA_KEY_SIZE_BITS_128      CAMELLIA_ALLOWED_KEY_SIZE_128
57  #define MAX_CAMELLIA_KEY_BITS           128
58  #define MAX_CAMELLIA_KEY_BYTES          16
59  #define MAX_CAMELLIA_BLOCK_SIZE_BYTES   MAX(CAMELLIA_128_BLOCK_SIZE_BYTES, 0)
```

From Vendor-Specific: Table 4 - Defines for Implemented Curves

```
60  #define ECC_NIST_P192                   NO
61  #define ECC_NIST_P224                   NO
62  #define ECC_NIST_P256                   YES
63  #define ECC_NIST_P384                   YES
64  #define ECC_NIST_P521                   NO
65  #define ECC_BN_P256                     YES
66  #define ECC_BN_P638                     NO
67  #define ECC_SM2_P256                    NO
68  #define ECC_CURVES                      \
69              {TPM_ECC_BN_P256,   TPM_ECC_BN_P638,   TPM_ECC_NIST_P192,              \
70               TPM_ECC_NIST_P224, TPM_ECC_NIST_P256, TPM_ECC_NIST_P384,              \
71               TPM_ECC_NIST_P521, TPM_ECC_SM2_P256}
72  #define ECC_CURVE_COUNT                 \
73              (ECC_BN_P256   + ECC_BN_P638   + ECC_NIST_P192 + ECC_NIST_P224 +       \
74               ECC_NIST_P256 + ECC_NIST_P384 + ECC_NIST_P521 + ECC_SM2_P256)
75  #define MAX_ECC_KEY_BITS                \
76              MAX(ECC_BN_P256 * 256,   MAX(ECC_BN_P638 * 638,                        \
77              MAX(ECC_NIST_P192 * 192, MAX(ECC_NIST_P224 * 224,                      \
78              MAX(ECC_NIST_P256 * 256, MAX(ECC_NIST_P384 * 384,                      \
79              MAX(ECC_NIST_P521 * 521, MAX(ECC_SM2_P256 * 256,                       \
80              0))))))))
81  #define MAX_ECC_KEY_BYTES               BITS_TO_BYTES(MAX_ECC_KEY_BITS)
```

From Vendor-Specific: Table 6 - Defines for PLATFORM Values

```
82  #define PLATFORM_FAMILY         TPM_SPEC_FAMILY
83  #define PLATFORM_LEVEL          TPM_SPEC_LEVEL
84  #define PLATFORM_VERSION        TPM_SPEC_VERSION
85  #define PLATFORM_YEAR           TPM_SPEC_YEAR
86  #define PLATFORM_DAY_OF_YEAR    TPM_SPEC_DAY_OF_YEAR
```

From Vendor-Specific: Table 7 - Defines for Implementation Values


```
87   #define FIELD_UPGRADE_IMPLEMENTED       NO
88   #define RADIX_BITS                      32
89   #define HASH_ALIGNMENT                  4
90   #define SYMMETRIC_ALIGNMENT             4
91   #define HASH_LIB                        OSSL
92   #define SYM_LIB                         OSSL
93   #define MATH_LIB                        OSSL
94   #define BSIZE                           UINT16
95   #define IMPLEMENTATION_PCR              24
96   #define PLATFORM_PCR                    24
97   #define DRTM_PCR                        17
98   #define HCRTM_PCR                       0
99   #define NUM_LOCALITIES                  5
100  #define MAX_HANDLE_NUM                  3
101  #define MAX_ACTIVE_SESSIONS             64
102  #define CONTEXT_SLOT                    UINT16
103  #define CONTEXT_COUNTER                 UINT64
104  #define MAX_LOADED_SESSIONS             3
105  #define MAX_SESSION_NUM                 3
106  #define MAX_LOADED_OBJECTS              3
107  #define MIN_EVICT_OBJECTS               2
108  #define NUM_POLICY_PCR_GROUP            1
109  #define NUM_AUTHVALUE_PCR_GROUP         1
110  #define MAX_CONTEXT_SIZE                2474
111  #define MAX_DIGEST_BUFFER               1024
112  #define MAX_NV_INDEX_SIZE               2048
113  #define MAX_NV_BUFFER_SIZE              1024
114  #define MAX_CAP_BUFFER                  1024
115  #define NV_MEMORY_SIZE                  16384
116  #define MIN_COUNTER_INDICES             8
117  #define NUM_STATIC_PCR                  16
118  #define MAX_ALG_LIST_SIZE               64
119  #define PRIMARY_SEED_SIZE               32
120  #define CONTEXT_ENCRYPT_ALGORITHM       AES
121  #define NV_CLOCK_UPDATE_INTERVAL        12
122  #define NUM_POLICY_PCR                  1
123  #define MAX_COMMAND_SIZE                4096
124  #define MAX_RESPONSE_SIZE               4096
125  #define ORDERLY_BITS                    8
126  #define MAX_SYM_DATA                    128
127  #define MAX_RNG_ENTROPY_SIZE            64
128  #define RAM_INDEX_SPACE                 512
129  #define RSA_DEFAULT_PUBLIC_EXPONENT     0x00010001
130  #define ENABLE_PCR_NO_INCREMENT         YES
131  #define CRT_FORMAT_RSA                  YES
132  #define VENDOR_COMMAND_COUNT            0
133  #define MAX_VENDOR_BUFFER_SIZE          1024
134  #define TPM_MAX_DERIVATION_BITS         8192
```

From Vendor-Specific: Table 2 - Defines for Implemented Algorithms

```
135  #define ALG_AES                         ALG_YES
136  #define ALG_CAMELLIA                    ALG_NO      /* Not specified by vendor */
137  #define ALG_CBC                         ALG_YES
138  #define ALG_CFB                         ALG_YES
139  #define ALG_CMAC                        ALG_YES
140  #define ALG_CTR                         ALG_YES
141  #define ALG_ECB                         ALG_YES
142  #define ALG_ECC                         ALG_YES
143  #define ALG_ECDAA                       (ALG_YES && ALG_ECC)
144  #define ALG_ECDH                        (ALG_YES && ALG_ECC)
145  #define ALG_ECDSA                       (ALG_YES && ALG_ECC)
146  #define ALG_ECMQV                       (ALG_NO && ALG_ECC)
147  #define ALG_ECSCHNORR                   (ALG_YES && ALG_ECC)
148  #define ALG_HMAC                        ALG_YES
```

```
149  #define ALG_KDF1_SP800_108              ALG_YES
150  #define ALG_KDF1_SP800_56A              (ALG_YES && ALG_ECC)
151  #define ALG_KDF2                        ALG_NO
152  #define ALG_KEYEDHASH                   ALG_YES
153  #define ALG_MGF1                        ALG_YES
154  #define ALG_OAEP                        (ALG_YES && ALG_RSA)
155  #define ALG_OFB                         ALG_YES
156  #define ALG_RSA                         ALG_YES
157  #define ALG_RSAES                       (ALG_YES && ALG_RSA)
158  #define ALG_RSAPSS                      (ALG_YES && ALG_RSA)
159  #define ALG_RSASSA                      (ALG_YES && ALG_RSA)
160  #define ALG_SHA                         ALG_NO      /* Not specified by vendor */
161  #define ALG_SHA1                        ALG_YES
162  #define ALG_SHA256                      ALG_YES
163  #define ALG_SHA384                      ALG_YES
164  #define ALG_SHA512                      ALG_NO
165  #define ALG_SM2                         (ALG_NO && ALG_ECC)
166  #define ALG_SM3_256                     ALG_NO
167  #define ALG_SM4                         ALG_NO
168  #define ALG_SYMCIPHER                   ALG_YES
169  #define ALG_TDES                        ALG_NO
170  #define ALG_XOR                         ALG_YES
```

From TCG Algorithm Registry: Table 2 - Definition of TPM_ALG_ID Constants

```
171  typedef UINT16                          TPM_ALG_ID;
172  #define     ALG_ERROR_VALUE             0x0000
173  #define TPM_ALG_ERROR                   (TPM_ALG_ID)(ALG_ERROR_VALUE)
174  #define     ALG_RSA_VALUE               0x0001
175  #if         ALG_RSA
176  #define TPM_ALG_RSA                     (TPM_ALG_ID)(ALG_RSA_VALUE)
177  #endif   // ALG_RSA
178  #define     ALG_TDES_VALUE              0x0003
179  #if         ALG_TDES
180  #define TPM_ALG_TDES                    (TPM_ALG_ID)(ALG_TDES_VALUE)
181  #endif   // ALG_TDES
182  #define     ALG_SHA_VALUE               0x0004
183  #if         ALG_SHA
184  #define TPM_ALG_SHA                     (TPM_ALG_ID)(ALG_SHA_VALUE)
185  #endif   // ALG_SHA
186  #define     ALG_SHA1_VALUE              0x0004
187  #if         ALG_SHA1
188  #define TPM_ALG_SHA1                    (TPM_ALG_ID)(ALG_SHA1_VALUE)
189  #endif   // ALG_SHA1
190  #define     ALG_HMAC_VALUE              0x0005
191  #if         ALG_HMAC
192  #define TPM_ALG_HMAC                    (TPM_ALG_ID)(ALG_HMAC_VALUE)
193  #endif   // ALG_HMAC
194  #define     ALG_AES_VALUE               0x0006
195  #if         ALG_AES
196  #define TPM_ALG_AES                     (TPM_ALG_ID)(ALG_AES_VALUE)
197  #endif   // ALG_AES
198  #define     ALG_MGF1_VALUE              0x0007
199  #if         ALG_MGF1
200  #define TPM_ALG_MGF1                    (TPM_ALG_ID)(ALG_MGF1_VALUE)
201  #endif   // ALG_MGF1
202  #define     ALG_KEYEDHASH_VALUE         0x0008
203  #if         ALG_KEYEDHASH
204  #define TPM_ALG_KEYEDHASH               (TPM_ALG_ID)(ALG_KEYEDHASH_VALUE)
205  #endif   // ALG_KEYEDHASH
206  #define     ALG_XOR_VALUE               0x000A
207  #if         ALG_XOR
208  #define TPM_ALG_XOR                     (TPM_ALG_ID)(ALG_XOR_VALUE)
209  #endif   // ALG_XOR
210  #define     ALG_SHA256_VALUE            0x000B
```



```
211  #if          ALG_SHA256
212  #define TPM_ALG_SHA256                  (TPM_ALG_ID)(ALG_SHA256_VALUE)
213  #endif    // ALG_SHA256
214  #define      ALG_SHA384_VALUE            0x000C
215  #if          ALG_SHA384
216  #define TPM_ALG_SHA384                  (TPM_ALG_ID)(ALG_SHA384_VALUE)
217  #endif    // ALG_SHA384
218  #define      ALG_SHA512_VALUE            0x000D
219  #if          ALG_SHA512
220  #define TPM_ALG_SHA512                  (TPM_ALG_ID)(ALG_SHA512_VALUE)
221  #endif    // ALG_SHA512
222  #define      ALG_NULL_VALUE              0x0010
223  #define TPM_ALG_NULL                    (TPM_ALG_ID)(ALG_NULL_VALUE)
224  #define      ALG_SM3_256_VALUE           0x0012
225  #if          ALG_SM3_256
226  #define TPM_ALG_SM3_256                 (TPM_ALG_ID)(ALG_SM3_256_VALUE)
227  #endif    // ALG_SM3_256
228  #define      ALG_SM4_VALUE               0x0013
229  #if          ALG_SM4
230  #define TPM_ALG_SM4                     (TPM_ALG_ID)(ALG_SM4_VALUE)
231  #endif    // ALG_SM4
232  #define      ALG_RSASSA_VALUE            0x0014
233  #if          ALG_RSASSA
234  #define TPM_ALG_RSASSA                  (TPM_ALG_ID)(ALG_RSASSA_VALUE)
235  #endif    // ALG_RSASSA
236  #define      ALG_RSAES_VALUE             0x0015
237  #if          ALG_RSAES
238  #define TPM_ALG_RSAES                   (TPM_ALG_ID)(ALG_RSAES_VALUE)
239  #endif    // ALG_RSAES
240  #define      ALG_RSAPSS_VALUE            0x0016
241  #if          ALG_RSAPSS
242  #define TPM_ALG_RSAPSS                  (TPM_ALG_ID)(ALG_RSAPSS_VALUE)
243  #endif    // ALG_RSAPSS
244  #define      ALG_OAEP_VALUE              0x0017
245  #if          ALG_OAEP
246  #define TPM_ALG_OAEP                    (TPM_ALG_ID)(ALG_OAEP_VALUE)
247  #endif    // ALG_OAEP
248  #define      ALG_ECDSA_VALUE             0x0018
249  #if          ALG_ECDSA
250  #define TPM_ALG_ECDSA                   (TPM_ALG_ID)(ALG_ECDSA_VALUE)
251  #endif    // ALG_ECDSA
252  #define      ALG_ECDH_VALUE              0x0019
253  #if          ALG_ECDH
254  #define TPM_ALG_ECDH                    (TPM_ALG_ID)(ALG_ECDH_VALUE)
255  #endif    // ALG_ECDH
256  #define      ALG_ECDAA_VALUE             0x001A
257  #if          ALG_ECDAA
258  #define TPM_ALG_ECDAA                   (TPM_ALG_ID)(ALG_ECDAA_VALUE)
259  #endif    // ALG_ECDAA
260  #define      ALG_SM2_VALUE               0x001B
261  #if          ALG_SM2
262  #define TPM_ALG_SM2                     (TPM_ALG_ID)(ALG_SM2_VALUE)
263  #endif    // ALG_SM2
264  #define      ALG_ECSCHNORR_VALUE         0x001C
265  #if          ALG_ECSCHNORR
266  #define TPM_ALG_ECSCHNORR               (TPM_ALG_ID)(ALG_ECSCHNORR_VALUE)
267  #endif    // ALG_ECSCHNORR
268  #define      ALG_ECMQV_VALUE             0x001D
269  #if          ALG_ECMQV
270  #define TPM_ALG_ECMQV                   (TPM_ALG_ID)(ALG_ECMQV_VALUE)
271  #endif    // ALG_ECMQV
272  #define      ALG_KDF1_SP800_56A_VALUE    0x0020
273  #if          ALG_KDF1_SP800_56A
274  #define TPM_ALG_KDF1_SP800_56A          (TPM_ALG_ID)(ALG_KDF1_SP800_56A_VALUE)
275  #endif    // ALG_KDF1_SP800_56A
276  #define      ALG_KDF2_VALUE              0x0021
```

```
#if         ALG_KDF2
#define TPM_ALG_KDF2                    (TPM_ALG_ID)(ALG_KDF2_VALUE)
#endif   // ALG_KDF2
#define     ALG_KDF1_SP800_108_VALUE    0x0022
#if         ALG_KDF1_SP800_108
#define TPM_ALG_KDF1_SP800_108          (TPM_ALG_ID)(ALG_KDF1_SP800_108_VALUE)
#endif   // ALG_KDF1_SP800_108
#define     ALG_ECC_VALUE               0x0023
#if         ALG_ECC
#define TPM_ALG_ECC                     (TPM_ALG_ID)(ALG_ECC_VALUE)
#endif   // ALG_ECC
#define     ALG_SYMCIPHER_VALUE         0x0025
#if         ALG_SYMCIPHER
#define TPM_ALG_SYMCIPHER               (TPM_ALG_ID)(ALG_SYMCIPHER_VALUE)
#endif   // ALG_SYMCIPHER
#define     ALG_CAMELLIA_VALUE          0x0026
#if         ALG_CAMELLIA
#define TPM_ALG_CAMELLIA                (TPM_ALG_ID)(ALG_CAMELLIA_VALUE)
#endif   // ALG_CAMELLIA
#define     ALG_CMAC_VALUE              0x003F
#if         ALG_CMAC
#define TPM_ALG_CMAC                    (TPM_ALG_ID)(ALG_CMAC_VALUE)
#endif   // ALG_CMAC
#define     ALG_CTR_VALUE               0x0040
#if         ALG_CTR
#define TPM_ALG_CTR                     (TPM_ALG_ID)(ALG_CTR_VALUE)
#endif   // ALG_CTR
#define     ALG_OFB_VALUE               0x0041
#if         ALG_OFB
#define TPM_ALG_OFB                     (TPM_ALG_ID)(ALG_OFB_VALUE)
#endif   // ALG_OFB
#define     ALG_CBC_VALUE               0x0042
#if         ALG_CBC
#define TPM_ALG_CBC                     (TPM_ALG_ID)(ALG_CBC_VALUE)
#endif   // ALG_CBC
#define     ALG_CFB_VALUE               0x0043
#if         ALG_CFB
#define TPM_ALG_CFB                     (TPM_ALG_ID)(ALG_CFB_VALUE)
#endif   // ALG_CFB
#define     ALG_ECB_VALUE               0x0044
#if         ALG_ECB
#define TPM_ALG_ECB                     (TPM_ALG_ID)(ALG_ECB_VALUE)
#endif   // ALG_ECB
```

Values derived from Table 1:2

```
#define     ALG_FIRST_VALUE             0x0001
#define TPM_ALG_FIRST                   (TPM_ALG_ID)(ALG_FIRST_VALUE)
#define     ALG_LAST_VALUE              0x0044
#define TPM_ALG_LAST                    (TPM_ALG_ID)(ALG_LAST_VALUE)
```

From TCG Algorithm Registry: Table 3 - Definition of TPM_ECC_CURVE Constants

```
typedef UINT16              TPM_ECC_CURVE;
#define TPM_ECC_NONE        (TPM_ECC_CURVE)(0x0000)
#define TPM_ECC_NIST_P192   (TPM_ECC_CURVE)(0x0001)
#define TPM_ECC_NIST_P224   (TPM_ECC_CURVE)(0x0002)
#define TPM_ECC_NIST_P256   (TPM_ECC_CURVE)(0x0003)
#define TPM_ECC_NIST_P384   (TPM_ECC_CURVE)(0x0004)
#define TPM_ECC_NIST_P521   (TPM_ECC_CURVE)(0x0005)
#define TPM_ECC_BN_P256     (TPM_ECC_CURVE)(0x0010)
#define TPM_ECC_BN_P638     (TPM_ECC_CURVE)(0x0011)
#define TPM_ECC_SM2_P256    (TPM_ECC_CURVE)(0x0020)
```


From TCG Algorithm Registry: Table 12 - Defines for SHA1 Hash Values

```
#define SHA1_DIGEST_SIZE    20
#define SHA1_BLOCK_SIZE     64
#define SHA1_DER_SIZE       15
#define SHA1_DER            \
            0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x0E,                    \
            0x03, 0x02, 0x1A, 0x05, 0x00, 0x04, 0x14
```

From TCG Algorithm Registry: Table 13 - Defines for SHA256 Hash Values

```
#define SHA256_DIGEST_SIZE  32
#define SHA256_BLOCK_SIZE   64
#define SHA256_DER_SIZE     19
#define SHA256_DER          \
            0x30, 0x31, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86,                    \
            0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01, 0x05,                    \
            0x00, 0x04, 0x20
```

From TCG Algorithm Registry: Table 14 - Defines for SHA384 Hash Values

```
#define SHA384_DIGEST_SIZE  48
#define SHA384_BLOCK_SIZE   128
#define SHA384_DER_SIZE     19
#define SHA384_DER          \
            0x30, 0x41, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86,                    \
            0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02, 0x05,                    \
            0x00, 0x04, 0x30
```

From TCG Algorithm Registry: Table 15 - Defines for SHA512 Hash Values

```
#define SHA512_DIGEST_SIZE  64
#define SHA512_BLOCK_SIZE   128
#define SHA512_DER_SIZE     19
#define SHA512_DER          \
            0x30, 0x51, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86,                    \
            0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03, 0x05,                    \
            0x00, 0x04, 0x40
```

From TCG Algorithm Registry: Table 16 - Defines for SM3_256 Hash Values

```
#define SM3_256_DIGEST_SIZE     32
#define SM3_256_BLOCK_SIZE      64
#define SM3_256_DER_SIZE        18
#define SM3_256_DER             \
            0x30, 0x30, 0x0C, 0x06, 0x08, 0x2A, 0x81,                          \
            0x1C, 0x81, 0x45, 0x01, 0x83, 0x11, 0x05, 0x00,                    \
            0x04, 0x20
```

From TCG Algorithm Registry: Table 17 - Defines for AES Symmetric Cipher Algorithm Constants

```
#define AES_ALLOWED_KEY_SIZE_128    YES
#define AES_ALLOWED_KEY_SIZE_192    YES
#define AES_ALLOWED_KEY_SIZE_256    YES
#define AES_128_BLOCK_SIZE_BYTES    16
#define AES_192_BLOCK_SIZE_BYTES    16
#define AES_256_BLOCK_SIZE_BYTES    16
```

From TCG Algorithm Registry: Table 18 - Defines for SM4 Symmetric Cipher Algorithm Constants

```
#define SM4_ALLOWED_KEY_SIZE_128    YES
#define SM4_128_BLOCK_SIZE_BYTES    16
```

From TCG Algorithm Registry: Table 19 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```
#define CAMELLIA_ALLOWED_KEY_SIZE_128   YES
#define CAMELLIA_ALLOWED_KEY_SIZE_192   YES
#define CAMELLIA_ALLOWED_KEY_SIZE_256   YES
#define CAMELLIA_128_BLOCK_SIZE_BYTES   16
#define CAMELLIA_192_BLOCK_SIZE_BYTES   16
#define CAMELLIA_256_BLOCK_SIZE_BYTES   16
```

From TCG Algorithm Registry: Table 17 - Defines for TDES Symmetric Cipher Algorithm Constants

```
#define TDES_ALLOWED_KEY_SIZE_128   YES
#define TDES_ALLOWED_KEY_SIZE_192   YES
#define TDES_128_BLOCK_SIZE_BYTES   8
#define TDES_192_BLOCK_SIZE_BYTES   8
```

From Vendor-Specific: Table 5 - Defines for Implemented Commands

```
#define CC_AC_GetCapability                 CC_YES
#define CC_AC_Send                          CC_YES
#define CC_ActivateCredential               CC_YES
#define CC_Certify                          CC_YES
#define CC_CertifyCreation                  CC_YES
#define CC_ChangeEPS                        CC_YES
#define CC_ChangePPS                        CC_YES
#define CC_Clear                            CC_YES
#define CC_ClearControl                     CC_YES
#define CC_ClockRateAdjust                  CC_YES
#define CC_ClockSet                         CC_YES
#define CC_Commit                           (CC_YES && ALG_ECC)
#define CC_ContextLoad                      CC_YES
#define CC_ContextSave                      CC_YES
#define CC_Create                           CC_YES
#define CC_CreateLoaded                     CC_YES
#define CC_CreatePrimary                    CC_YES
#define CC_DictionaryAttackLockReset        CC_YES
#define CC_DictionaryAttackParameters       CC_YES
#define CC_Duplicate                        CC_YES
#define CC_ECC_Parameters                   (CC_YES && ALG_ECC)
#define CC_ECDH_KeyGen                      (CC_YES && ALG_ECC)
#define CC_ECDH_ZGen                        (CC_YES && ALG_ECC)
#define CC_EC_Ephemeral                     (CC_YES && ALG_ECC)
#define CC_EncryptDecrypt                   CC_YES
#define CC_EncryptDecrypt2                  CC_YES
#define CC_EventSequenceComplete            CC_YES
#define CC_EvictControl                     CC_YES
#define CC_FieldUpgradeData                 CC_NO
#define CC_FieldUpgradeStart                CC_NO
#define CC_FirmwareRead                     CC_NO
#define CC_FlushContext                     CC_YES
#define CC_GetCapability                    CC_YES
#define CC_GetCommandAuditDigest            CC_YES
#define CC_GetRandom                        CC_YES
#define CC_GetSessionAuditDigest            CC_YES
#define CC_GetTestResult                    CC_YES
#define CC_GetTime                          CC_YES
#define CC_HMAC                             (CC_YES && !ALG_CMAC)
#define CC_HMAC_Start                       (CC_YES && !ALG_CMAC)
#define CC_Hash                             CC_YES
#define CC_HashSequenceStart                CC_YES
#define CC_HierarchyChangeAuth              CC_YES
#define CC_HierarchyControl                 CC_YES
#define CC_Import                           CC_YES
#define CC_IncrementalSelfTest              CC_YES
```



```
432  #define CC_Load                          CC_YES
433  #define CC_LoadExternal                  CC_YES
434  #define CC_MAC                           (CC_YES && ALG_CMAC)
435  #define CC_MAC_Start                     (CC_YES && ALG_CMAC)
436  #define CC_MakeCredential                CC_YES
437  #define CC_NV_Certify                    CC_YES
438  #define CC_NV_ChangeAuth                 CC_YES
439  #define CC_NV_DefineSpace                CC_YES
440  #define CC_NV_Extend                     CC_YES
441  #define CC_NV_GlobalWriteLock            CC_YES
442  #define CC_NV_Increment                  CC_YES
443  #define CC_NV_Read                       CC_YES
444  #define CC_NV_ReadLock                   CC_YES
445  #define CC_NV_ReadPublic                 CC_YES
446  #define CC_NV_SetBits                    CC_YES
447  #define CC_NV_UndefineSpace              CC_YES
448  #define CC_NV_UndefineSpaceSpecial       CC_YES
449  #define CC_NV_Write                      CC_YES
450  #define CC_NV_WriteLock                  CC_YES
451  #define CC_ObjectChangeAuth              CC_YES
452  #define CC_PCR_Allocate                  CC_YES
453  #define CC_PCR_Event                     CC_YES
454  #define CC_PCR_Extend                    CC_YES
455  #define CC_PCR_Read                      CC_YES
456  #define CC_PCR_Reset                     CC_YES
457  #define CC_PCR_SetAuthPolicy             CC_YES
458  #define CC_PCR_SetAuthValue              CC_YES
459  #define CC_PP_Commands                   CC_YES
460  #define CC_PolicyAuthValue               CC_YES
461  #define CC_PolicyAuthorize               CC_YES
462  #define CC_PolicyAuthorizeNV             CC_YES
463  #define CC_PolicyCommandCode             CC_YES
464  #define CC_PolicyCounterTimer            CC_YES
465  #define CC_PolicyCpHash                  CC_YES
466  #define CC_PolicyDuplicationSelect       CC_YES
467  #define CC_PolicyGetDigest               CC_YES
468  #define CC_PolicyLocality                CC_YES
469  #define CC_PolicyNV                      CC_YES
470  #define CC_PolicyNameHash                CC_YES
471  #define CC_PolicyNvWritten               CC_YES
472  #define CC_PolicyOR                      CC_YES
473  #define CC_PolicyPCR                     CC_YES
474  #define CC_PolicyPassword                CC_YES
475  #define CC_PolicyPhysicalPresence        CC_YES
476  #define CC_PolicyRestart                 CC_YES
477  #define CC_PolicySecret                  CC_YES
478  #define CC_PolicySigned                  CC_YES
479  #define CC_PolicyTemplate                CC_YES
480  #define CC_PolicyTicket                  CC_YES
481  #define CC_Policy_AC_SendSelect          CC_YES
482  #define CC_Quote                         CC_YES
483  #define CC_RSA_Decrypt                   (CC_YES && ALG_RSA)
484  #define CC_RSA_Encrypt                   (CC_YES && ALG_RSA)
485  #define CC_ReadClock                     CC_YES
486  #define CC_ReadPublic                    CC_YES
487  #define CC_Rewrap                        CC_YES
488  #define CC_SelfTest                      CC_YES
489  #define CC_SequenceComplete              CC_YES
490  #define CC_SequenceUpdate                CC_YES
491  #define CC_SetAlgorithmSet               CC_YES
492  #define CC_SetCommandCodeAuditStatus     CC_YES
493  #define CC_SetPrimaryPolicy              CC_YES
494  #define CC_Shutdown                      CC_YES
495  #define CC_Sign                          CC_YES
496  #define CC_StartAuthSession              CC_YES
497  #define CC_Startup                       CC_YES
```

```
498  #define CC_StirRandom                        CC_YES
499  #define CC_TestParms                         CC_YES
500  #define CC_Unseal                            CC_YES
501  #define CC_Vendor_TCG_Test                   CC_YES
502  #define CC_VerifySignature                   CC_YES
503  #define CC_ZGen_2Phase                       (CC_YES && ALG_ECC)
```

From TPM 2.0 Part 2: Table 12 - Definition of TPM_CC Constants

```
504  typedef UINT32                               TPM_CC;
505  #if         CC_NV_UndefineSpaceSpecial
506  #define TPM_CC_NV_UndefineSpaceSpecial       (TPM_CC)(0x0000011F)
507  #endif
508  #if         CC_EvictControl
509  #define TPM_CC_EvictControl                  (TPM_CC)(0x00000120)
510  #endif
511  #if         CC_HierarchyControl
512  #define TPM_CC_HierarchyControl              (TPM_CC)(0x00000121)
513  #endif
514  #if         CC_NV_UndefineSpace
515  #define TPM_CC_NV_UndefineSpace              (TPM_CC)(0x00000122)
516  #endif
517  #if         CC_ChangeEPS
518  #define TPM_CC_ChangeEPS                     (TPM_CC)(0x00000124)
519  #endif
520  #if         CC_ChangePPS
521  #define TPM_CC_ChangePPS                     (TPM_CC)(0x00000125)
522  #endif
523  #if         CC_Clear
524  #define TPM_CC_Clear                         (TPM_CC)(0x00000126)
525  #endif
526  #if         CC_ClearControl
527  #define TPM_CC_ClearControl                  (TPM_CC)(0x00000127)
528  #endif
529  #if         CC_ClockSet
530  #define TPM_CC_ClockSet                      (TPM_CC)(0x00000128)
531  #endif
532  #if         CC_HierarchyChangeAuth
533  #define TPM_CC_HierarchyChangeAuth           (TPM_CC)(0x00000129)
534  #endif
535  #if         CC_NV_DefineSpace
536  #define TPM_CC_NV_DefineSpace                (TPM_CC)(0x0000012A)
537  #endif
538  #if         CC_PCR_Allocate
539  #define TPM_CC_PCR_Allocate                  (TPM_CC)(0x0000012B)
540  #endif
541  #if         CC_PCR_SetAuthPolicy
542  #define TPM_CC_PCR_SetAuthPolicy             (TPM_CC)(0x0000012C)
543  #endif
544  #if         CC_PP_Commands
545  #define TPM_CC_PP_Commands                   (TPM_CC)(0x0000012D)
546  #endif
547  #if         CC_SetPrimaryPolicy
548  #define TPM_CC_SetPrimaryPolicy              (TPM_CC)(0x0000012E)
549  #endif
550  #if         CC_FieldUpgradeStart
551  #define TPM_CC_FieldUpgradeStart             (TPM_CC)(0x0000012F)
552  #endif
553  #if         CC_ClockRateAdjust
554  #define TPM_CC_ClockRateAdjust               (TPM_CC)(0x00000130)
555  #endif
556  #if         CC_CreatePrimary
557  #define TPM_CC_CreatePrimary                 (TPM_CC)(0x00000131)
558  #endif
559  #if         CC_NV_GlobalWriteLock
```



```
560  #define TPM_CC_NV_GlobalWriteLock           (TPM_CC)(0x00000132)
561  #endif
562  #if         CC_GetCommandAuditDigest
563  #define TPM_CC_GetCommandAuditDigest        (TPM_CC)(0x00000133)
564  #endif
565  #if         CC_NV_Increment
566  #define TPM_CC_NV_Increment                 (TPM_CC)(0x00000134)
567  #endif
568  #if         CC_NV_SetBits
569  #define TPM_CC_NV_SetBits                   (TPM_CC)(0x00000135)
570  #endif
571  #if         CC_NV_Extend
572  #define TPM_CC_NV_Extend                    (TPM_CC)(0x00000136)
573  #endif
574  #if         CC_NV_Write
575  #define TPM_CC_NV_Write                     (TPM_CC)(0x00000137)
576  #endif
577  #if         CC_NV_WriteLock
578  #define TPM_CC_NV_WriteLock                 (TPM_CC)(0x00000138)
579  #endif
580  #if         CC_DictionaryAttackLockReset
581  #define TPM_CC_DictionaryAttackLockReset    (TPM_CC)(0x00000139)
582  #endif
583  #if         CC_DictionaryAttackParameters
584  #define TPM_CC_DictionaryAttackParameters   (TPM_CC)(0x0000013A)
585  #endif
586  #if         CC_NV_ChangeAuth
587  #define TPM_CC_NV_ChangeAuth                (TPM_CC)(0x0000013B)
588  #endif
589  #if         CC_PCR_Event
590  #define TPM_CC_PCR_Event                    (TPM_CC)(0x0000013C)
591  #endif
592  #if         CC_PCR_Reset
593  #define TPM_CC_PCR_Reset                    (TPM_CC)(0x0000013D)
594  #endif
595  #if         CC_SequenceComplete
596  #define TPM_CC_SequenceComplete             (TPM_CC)(0x0000013E)
597  #endif
598  #if         CC_SetAlgorithmSet
599  #define TPM_CC_SetAlgorithmSet              (TPM_CC)(0x0000013F)
600  #endif
601  #if         CC_SetCommandCodeAuditStatus
602  #define TPM_CC_SetCommandCodeAuditStatus    (TPM_CC)(0x00000140)
603  #endif
604  #if         CC_FieldUpgradeData
605  #define TPM_CC_FieldUpgradeData             (TPM_CC)(0x00000141)
606  #endif
607  #if         CC_IncrementalSelfTest
608  #define TPM_CC_IncrementalSelfTest          (TPM_CC)(0x00000142)
609  #endif
610  #if         CC_SelfTest
611  #define TPM_CC_SelfTest                     (TPM_CC)(0x00000143)
612  #endif
613  #if         CC_Startup
614  #define TPM_CC_Startup                      (TPM_CC)(0x00000144)
615  #endif
616  #if         CC_Shutdown
617  #define TPM_CC_Shutdown                     (TPM_CC)(0x00000145)
618  #endif
619  #if         CC_StirRandom
620  #define TPM_CC_StirRandom                   (TPM_CC)(0x00000146)
621  #endif
622  #if         CC_ActivateCredential
623  #define TPM_CC_ActivateCredential           (TPM_CC)(0x00000147)
624  #endif
625  #if         CC_Certify
```

```
626  #define TPM_CC_Certify                      (TPM_CC)(0x00000148)
627  #endif
628  #if         CC_PolicyNV
629  #define TPM_CC_PolicyNV                     (TPM_CC)(0x00000149)
630  #endif
631  #if         CC_CertifyCreation
632  #define TPM_CC_CertifyCreation              (TPM_CC)(0x0000014A)
633  #endif
634  #if         CC_Duplicate
635  #define TPM_CC_Duplicate                    (TPM_CC)(0x0000014B)
636  #endif
637  #if         CC_GetTime
638  #define TPM_CC_GetTime                      (TPM_CC)(0x0000014C)
639  #endif
640  #if         CC_GetSessionAuditDigest
641  #define TPM_CC_GetSessionAuditDigest        (TPM_CC)(0x0000014D)
642  #endif
643  #if         CC_NV_Read
644  #define TPM_CC_NV_Read                      (TPM_CC)(0x0000014E)
645  #endif
646  #if         CC_NV_ReadLock
647  #define TPM_CC_NV_ReadLock                  (TPM_CC)(0x0000014F)
648  #endif
649  #if         CC_ObjectChangeAuth
650  #define TPM_CC_ObjectChangeAuth             (TPM_CC)(0x00000150)
651  #endif
652  #if         CC_PolicySecret
653  #define TPM_CC_PolicySecret                 (TPM_CC)(0x00000151)
654  #endif
655  #if         CC_Rewrap
656  #define TPM_CC_Rewrap                       (TPM_CC)(0x00000152)
657  #endif
658  #if         CC_Create
659  #define TPM_CC_Create                       (TPM_CC)(0x00000153)
660  #endif
661  #if         CC_ECDH_ZGen
662  #define TPM_CC_ECDH_ZGen                    (TPM_CC)(0x00000154)
663  #endif
664  #if         CC_HMAC
665  #define TPM_CC_HMAC                         (TPM_CC)(0x00000155)
666  #endif
667  #if         CC_MAC
668  #define TPM_CC_MAC                          (TPM_CC)(0x00000155)
669  #endif
670  #if         CC_Import
671  #define TPM_CC_Import                       (TPM_CC)(0x00000156)
672  #endif
673  #if         CC_Load
674  #define TPM_CC_Load                         (TPM_CC)(0x00000157)
675  #endif
676  #if         CC_Quote
677  #define TPM_CC_Quote                        (TPM_CC)(0x00000158)
678  #endif
679  #if         CC_RSA_Decrypt
680  #define TPM_CC_RSA_Decrypt                  (TPM_CC)(0x00000159)
681  #endif
682  #if         CC_HMAC_Start
683  #define TPM_CC_HMAC_Start                   (TPM_CC)(0x0000015B)
684  #endif
685  #if         CC_MAC_Start
686  #define TPM_CC_MAC_Start                    (TPM_CC)(0x0000015B)
687  #endif
688  #if         CC_SequenceUpdate
689  #define TPM_CC_SequenceUpdate               (TPM_CC)(0x0000015C)
690  #endif
691  #if         CC_Sign
```



```
692  #define TPM_CC_Sign                          (TPM_CC)(0x0000015D)
693  #endif
694  #if         CC_Unseal
695  #define TPM_CC_Unseal                        (TPM_CC)(0x0000015E)
696  #endif
697  #if         CC_PolicySigned
698  #define TPM_CC_PolicySigned                  (TPM_CC)(0x00000160)
699  #endif
700  #if         CC_ContextLoad
701  #define TPM_CC_ContextLoad                   (TPM_CC)(0x00000161)
702  #endif
703  #if         CC_ContextSave
704  #define TPM_CC_ContextSave                   (TPM_CC)(0x00000162)
705  #endif
706  #if         CC_ECDH_KeyGen
707  #define TPM_CC_ECDH_KeyGen                   (TPM_CC)(0x00000163)
708  #endif
709  #if         CC_EncryptDecrypt
710  #define TPM_CC_EncryptDecrypt                (TPM_CC)(0x00000164)
711  #endif
712  #if         CC_FlushContext
713  #define TPM_CC_FlushContext                  (TPM_CC)(0x00000165)
714  #endif
715  #if         CC_LoadExternal
716  #define TPM_CC_LoadExternal                  (TPM_CC)(0x00000167)
717  #endif
718  #if         CC_MakeCredential
719  #define TPM_CC_MakeCredential                (TPM_CC)(0x00000168)
720  #endif
721  #if         CC_NV_ReadPublic
722  #define TPM_CC_NV_ReadPublic                 (TPM_CC)(0x00000169)
723  #endif
724  #if         CC_PolicyAuthorize
725  #define TPM_CC_PolicyAuthorize               (TPM_CC)(0x0000016A)
726  #endif
727  #if         CC_PolicyAuthValue
728  #define TPM_CC_PolicyAuthValue               (TPM_CC)(0x0000016B)
729  #endif
730  #if         CC_PolicyCommandCode
731  #define TPM_CC_PolicyCommandCode             (TPM_CC)(0x0000016C)
732  #endif
733  #if         CC_PolicyCounterTimer
734  #define TPM_CC_PolicyCounterTimer            (TPM_CC)(0x0000016D)
735  #endif
736  #if         CC_PolicyCpHash
737  #define TPM_CC_PolicyCpHash                  (TPM_CC)(0x0000016E)
738  #endif
739  #if         CC_PolicyLocality
740  #define TPM_CC_PolicyLocality                (TPM_CC)(0x0000016F)
741  #endif
742  #if         CC_PolicyNameHash
743  #define TPM_CC_PolicyNameHash                (TPM_CC)(0x00000170)
744  #endif
745  #if         CC_PolicyOR
746  #define TPM_CC_PolicyOR                      (TPM_CC)(0x00000171)
747  #endif
748  #if         CC_PolicyTicket
749  #define TPM_CC_PolicyTicket                  (TPM_CC)(0x00000172)
750  #endif
751  #if         CC_ReadPublic
752  #define TPM_CC_ReadPublic                    (TPM_CC)(0x00000173)
753  #endif
754  #if         CC_RSA_Encrypt
755  #define TPM_CC_RSA_Encrypt                   (TPM_CC)(0x00000174)
756  #endif
757  #if         CC_StartAuthSession
```

```
758  #define TPM_CC_StartAuthSession              (TPM_CC)(0x00000176)
759  #endif
760  #if         CC_VerifySignature
761  #define TPM_CC_VerifySignature               (TPM_CC)(0x00000177)
762  #endif
763  #if         CC_ECC_Parameters
764  #define TPM_CC_ECC_Parameters                (TPM_CC)(0x00000178)
765  #endif
766  #if         CC_FirmwareRead
767  #define TPM_CC_FirmwareRead                  (TPM_CC)(0x00000179)
768  #endif
769  #if         CC_GetCapability
770  #define TPM_CC_GetCapability                 (TPM_CC)(0x0000017A)
771  #endif
772  #if         CC_GetRandom
773  #define TPM_CC_GetRandom                     (TPM_CC)(0x0000017B)
774  #endif
775  #if         CC_GetTestResult
776  #define TPM_CC_GetTestResult                 (TPM_CC)(0x0000017C)
777  #endif
778  #if         CC_Hash
779  #define TPM_CC_Hash                          (TPM_CC)(0x0000017D)
780  #endif
781  #if         CC_PCR_Read
782  #define TPM_CC_PCR_Read                      (TPM_CC)(0x0000017E)
783  #endif
784  #if         CC_PolicyPCR
785  #define TPM_CC_PolicyPCR                     (TPM_CC)(0x0000017F)
786  #endif
787  #if         CC_PolicyRestart
788  #define TPM_CC_PolicyRestart                 (TPM_CC)(0x00000180)
789  #endif
790  #if         CC_ReadClock
791  #define TPM_CC_ReadClock                     (TPM_CC)(0x00000181)
792  #endif
793  #if         CC_PCR_Extend
794  #define TPM_CC_PCR_Extend                    (TPM_CC)(0x00000182)
795  #endif
796  #if         CC_PCR_SetAuthValue
797  #define TPM_CC_PCR_SetAuthValue              (TPM_CC)(0x00000183)
798  #endif
799  #if         CC_NV_Certify
800  #define TPM_CC_NV_Certify                    (TPM_CC)(0x00000184)
801  #endif
802  #if         CC_EventSequenceComplete
803  #define TPM_CC_EventSequenceComplete         (TPM_CC)(0x00000185)
804  #endif
805  #if         CC_HashSequenceStart
806  #define TPM_CC_HashSequenceStart             (TPM_CC)(0x00000186)
807  #endif
808  #if         CC_PolicyPhysicalPresence
809  #define TPM_CC_PolicyPhysicalPresence        (TPM_CC)(0x00000187)
810  #endif
811  #if         CC_PolicyDuplicationSelect
812  #define TPM_CC_PolicyDuplicationSelect       (TPM_CC)(0x00000188)
813  #endif
814  #if         CC_PolicyGetDigest
815  #define TPM_CC_PolicyGetDigest               (TPM_CC)(0x00000189)
816  #endif
817  #if         CC_TestParms
818  #define TPM_CC_TestParms                     (TPM_CC)(0x0000018A)
819  #endif
820  #if         CC_Commit
821  #define TPM_CC_Commit                        (TPM_CC)(0x0000018B)
822  #endif
823  #if         CC_PolicyPassword
```



```
824  #define TPM_CC_PolicyPassword                (TPM_CC)(0x0000018C)
825  #endif
826  #if          CC_ZGen_2Phase
827  #define TPM_CC_ZGen_2Phase                   (TPM_CC)(0x0000018D)
828  #endif
829  #if          CC_EC_Ephemeral
830  #define TPM_CC_EC_Ephemeral                  (TPM_CC)(0x0000018E)
831  #endif
832  #if          CC_PolicyNvWritten
833  #define TPM_CC_PolicyNvWritten               (TPM_CC)(0x0000018F)
834  #endif
835  #if          CC_PolicyTemplate
836  #define TPM_CC_PolicyTemplate                (TPM_CC)(0x00000190)
837  #endif
838  #if          CC_CreateLoaded
839  #define TPM_CC_CreateLoaded                  (TPM_CC)(0x00000191)
840  #endif
841  #if          CC_PolicyAuthorizeNV
842  #define TPM_CC_PolicyAuthorizeNV             (TPM_CC)(0x00000192)
843  #endif
844  #if          CC_EncryptDecrypt2
845  #define TPM_CC_EncryptDecrypt2               (TPM_CC)(0x00000193)
846  #endif
847  #if          CC_AC_GetCapability
848  #define TPM_CC_AC_GetCapability              (TPM_CC)(0x00000194)
849  #endif
850  #if          CC_AC_Send
851  #define TPM_CC_AC_Send                       (TPM_CC)(0x00000195)
852  #endif
853  #if          CC_Policy_AC_SendSelect
854  #define TPM_CC_Policy_AC_SendSelect          (TPM_CC)(0x00000196)
855  #endif
856  #define CC_VEND                              0x20000000
857  #if          CC_Vendor_TCG_Test
858  #define TPM_CC_Vendor_TCG_Test               (TPM_CC)(0x20000000)
859  #endif
```

Additional values for benefit of code

```
860  #define TPM_CC_FIRST                         0x0000011F
861  #define TPM_CC_LAST                          0x00000196
862  #if COMPRESSED_LISTS
863  #define ADD_FILL            0
864  #else
865  #define ADD_FILL            1
866  #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)

```
867  #define LIBRARY_COMMAND_ARRAY_SIZE       (0      \
868      + (ADD_FILL || CC_NV_UndefineSpaceSpecial)          /* 0x0000011F */       \
869      + (ADD_FILL || CC_EvictControl)                     /* 0x00000120 */       \
870      + (ADD_FILL || CC_HierarchyControl)                 /* 0x00000121 */       \
871      + (ADD_FILL || CC_NV_UndefineSpace)                 /* 0x00000122 */       \
872      +  ADD_FILL                                         /* 0x00000123 */       \
873      + (ADD_FILL || CC_ChangeEPS)                        /* 0x00000124 */       \
874      + (ADD_FILL || CC_ChangePPS)                        /* 0x00000125 */       \
875      + (ADD_FILL || CC_Clear)                            /* 0x00000126 */       \
876      + (ADD_FILL || CC_ClearControl)                     /* 0x00000127 */       \
877      + (ADD_FILL || CC_ClockSet)                         /* 0x00000128 */       \
878      + (ADD_FILL || CC_HierarchyChangeAuth)              /* 0x00000129 */       \
879      + (ADD_FILL || CC_NV_DefineSpace)                   /* 0x0000012A */       \
880      + (ADD_FILL || CC_PCR_Allocate)                     /* 0x0000012B */       \
881      + (ADD_FILL || CC_PCR_SetAuthPolicy)                /* 0x0000012C */       \
```

```
882     + (ADD_FILL || CC_PP_Commands)                          /* 0x0000012D */      \
883     + (ADD_FILL || CC_SetPrimaryPolicy)                     /* 0x0000012E */      \
884     + (ADD_FILL || CC_FieldUpgradeStart)                    /* 0x0000012F */      \
885     + (ADD_FILL || CC_ClockRateAdjust)                      /* 0x00000130 */      \
886     + (ADD_FILL || CC_CreatePrimary)                        /* 0x00000131 */      \
887     + (ADD_FILL || CC_NV_GlobalWriteLock)                   /* 0x00000132 */      \
888     + (ADD_FILL || CC_GetCommandAuditDigest)                /* 0x00000133 */      \
889     + (ADD_FILL || CC_NV_Increment)                         /* 0x00000134 */      \
890     + (ADD_FILL || CC_NV_SetBits)                           /* 0x00000135 */      \
891     + (ADD_FILL || CC_NV_Extend)                            /* 0x00000136 */      \
892     + (ADD_FILL || CC_NV_Write)                             /* 0x00000137 */      \
893     + (ADD_FILL || CC_NV_WriteLock)                         /* 0x00000138 */      \
894     + (ADD_FILL || CC_DictionaryAttackLockReset)            /* 0x00000139 */      \
895     + (ADD_FILL || CC_DictionaryAttackParameters)           /* 0x0000013A */      \
896     + (ADD_FILL || CC_NV_ChangeAuth)                        /* 0x0000013B */      \
897     + (ADD_FILL || CC_PCR_Event)                            /* 0x0000013C */      \
898     + (ADD_FILL || CC_PCR_Reset)                            /* 0x0000013D */      \
899     + (ADD_FILL || CC_SequenceComplete)                     /* 0x0000013E */      \
900     + (ADD_FILL || CC_SetAlgorithmSet)                      /* 0x0000013F */      \
901     + (ADD_FILL || CC_SetCommandCodeAuditStatus)            /* 0x00000140 */      \
902     + (ADD_FILL || CC_FieldUpgradeData)                     /* 0x00000141 */      \
903     + (ADD_FILL || CC_IncrementalSelfTest)                  /* 0x00000142 */      \
904     + (ADD_FILL || CC_SelfTest)                             /* 0x00000143 */      \
905     + (ADD_FILL || CC_Startup)                              /* 0x00000144 */      \
906     + (ADD_FILL || CC_Shutdown)                             /* 0x00000145 */      \
907     + (ADD_FILL || CC_StirRandom)                           /* 0x00000146 */      \
908     + (ADD_FILL || CC_ActivateCredential)                   /* 0x00000147 */      \
909     + (ADD_FILL || CC_Certify)                              /* 0x00000148 */      \
910     + (ADD_FILL || CC_PolicyNV)                             /* 0x00000149 */      \
911     + (ADD_FILL || CC_CertifyCreation)                      /* 0x0000014A */      \
912     + (ADD_FILL || CC_Duplicate)                            /* 0x0000014B */      \
913     + (ADD_FILL || CC_GetTime)                              /* 0x0000014C */      \
914     + (ADD_FILL || CC_GetSessionAuditDigest)                /* 0x0000014D */      \
915     + (ADD_FILL || CC_NV_Read)                              /* 0x0000014E */      \
916     + (ADD_FILL || CC_NV_ReadLock)                          /* 0x0000014F */      \
917     + (ADD_FILL || CC_ObjectChangeAuth)                     /* 0x00000150 */      \
918     + (ADD_FILL || CC_PolicySecret)                         /* 0x00000151 */      \
919     + (ADD_FILL || CC_Rewrap)                               /* 0x00000152 */      \
920     + (ADD_FILL || CC_Create)                               /* 0x00000153 */      \
921     + (ADD_FILL || CC_ECDH_ZGen)                            /* 0x00000154 */      \
922     + (ADD_FILL || CC_HMAC || CC_MAC)                       /* 0x00000155 */      \
923     + (ADD_FILL || CC_Import)                               /* 0x00000156 */      \
924     + (ADD_FILL || CC_Load)                                 /* 0x00000157 */      \
925     + (ADD_FILL || CC_Quote)                                /* 0x00000158 */      \
926     + (ADD_FILL || CC_RSA_Decrypt)                          /* 0x00000159 */      \
927     +  ADD_FILL                                             /* 0x0000015A */      \
928     + (ADD_FILL || CC_HMAC_Start || CC_MAC_Start)           /* 0x0000015B */      \
929     + (ADD_FILL || CC_SequenceUpdate)                       /* 0x0000015C */      \
930     + (ADD_FILL || CC_Sign)                                 /* 0x0000015D */      \
931     + (ADD_FILL || CC_Unseal)                               /* 0x0000015E */      \
932     +  ADD_FILL                                             /* 0x0000015F */      \
933     + (ADD_FILL || CC_PolicySigned)                         /* 0x00000160 */      \
934     + (ADD_FILL || CC_ContextLoad)                          /* 0x00000161 */      \
935     + (ADD_FILL || CC_ContextSave)                          /* 0x00000162 */      \
936     + (ADD_FILL || CC_ECDH_KeyGen)                          /* 0x00000163 */      \
937     + (ADD_FILL || CC_EncryptDecrypt)                       /* 0x00000164 */      \
938     + (ADD_FILL || CC_FlushContext)                         /* 0x00000165 */      \
939     +  ADD_FILL                                             /* 0x00000166 */      \
940     + (ADD_FILL || CC_LoadExternal)                         /* 0x00000167 */      \
941     + (ADD_FILL || CC_MakeCredential)                       /* 0x00000168 */      \
942     + (ADD_FILL || CC_NV_ReadPublic)                        /* 0x00000169 */      \
943     + (ADD_FILL || CC_PolicyAuthorize)                      /* 0x0000016A */      \
944     + (ADD_FILL || CC_PolicyAuthValue)                      /* 0x0000016B */      \
945     + (ADD_FILL || CC_PolicyCommandCode)                    /* 0x0000016C */      \
946     + (ADD_FILL || CC_PolicyCounterTimer)                   /* 0x0000016D */      \
947     + (ADD_FILL || CC_PolicyCpHash)                         /* 0x0000016E */      \
```



```
948         + (ADD_FILL || CC_PolicyLocality)                    /* 0x0000016F */       \
949         + (ADD_FILL || CC_PolicyNameHash)                    /* 0x00000170 */       \
950         + (ADD_FILL || CC_PolicyOR)                          /* 0x00000171 */       \
951         + (ADD_FILL || CC_PolicyTicket)                      /* 0x00000172 */       \
952         + (ADD_FILL || CC_ReadPublic)                        /* 0x00000173 */       \
953         + (ADD_FILL || CC_RSA_Encrypt)                       /* 0x00000174 */       \
954         +  ADD_FILL                                          /* 0x00000175 */       \
955         + (ADD_FILL || CC_StartAuthSession)                  /* 0x00000176 */       \
956         + (ADD_FILL || CC_VerifySignature)                   /* 0x00000177 */       \
957         + (ADD_FILL || CC_ECC_Parameters)                    /* 0x00000178 */       \
958         + (ADD_FILL || CC_FirmwareRead)                      /* 0x00000179 */       \
959         + (ADD_FILL || CC_GetCapability)                     /* 0x0000017A */       \
960         + (ADD_FILL || CC_GetRandom)                         /* 0x0000017B */       \
961         + (ADD_FILL || CC_GetTestResult)                     /* 0x0000017C */       \
962         + (ADD_FILL || CC_Hash)                              /* 0x0000017D */       \
963         + (ADD_FILL || CC_PCR_Read)                          /* 0x0000017E */       \
964         + (ADD_FILL || CC_PolicyPCR)                         /* 0x0000017F */       \
965         + (ADD_FILL || CC_PolicyRestart)                     /* 0x00000180 */       \
966         + (ADD_FILL || CC_ReadClock)                         /* 0x00000181 */       \
967         + (ADD_FILL || CC_PCR_Extend)                        /* 0x00000182 */       \
968         + (ADD_FILL || CC_PCR_SetAuthValue)                  /* 0x00000183 */       \
969         + (ADD_FILL || CC_NV_Certify)                        /* 0x00000184 */       \
970         + (ADD_FILL || CC_EventSequenceComplete)             /* 0x00000185 */       \
971         + (ADD_FILL || CC_HashSequenceStart)                 /* 0x00000186 */       \
972         + (ADD_FILL || CC_PolicyPhysicalPresence)            /* 0x00000187 */       \
973         + (ADD_FILL || CC_PolicyDuplicationSelect)           /* 0x00000188 */       \
974         + (ADD_FILL || CC_PolicyGetDigest)                   /* 0x00000189 */       \
975         + (ADD_FILL || CC_TestParms)                         /* 0x0000018A */       \
976         + (ADD_FILL || CC_Commit)                            /* 0x0000018B */       \
977         + (ADD_FILL || CC_PolicyPassword)                    /* 0x0000018C */       \
978         + (ADD_FILL || CC_ZGen_2Phase)                       /* 0x0000018D */       \
979         + (ADD_FILL || CC_EC_Ephemeral)                      /* 0x0000018E */       \
980         + (ADD_FILL || CC_PolicyNvWritten)                   /* 0x0000018F */       \
981         + (ADD_FILL || CC_PolicyTemplate)                    /* 0x00000190 */       \
982         + (ADD_FILL || CC_CreateLoaded)                      /* 0x00000191 */       \
983         + (ADD_FILL || CC_PolicyAuthorizeNV)                 /* 0x00000192 */       \
984         + (ADD_FILL || CC_EncryptDecrypt2)                   /* 0x00000193 */       \
985         + (ADD_FILL || CC_AC_GetCapability)                  /* 0x00000194 */       \
986         + (ADD_FILL || CC_AC_Send)                           /* 0x00000195 */       \
987         + (ADD_FILL || CC_Policy_AC_SendSelect)              /* 0x00000196 */       \
988         )
989     #define VENDOR_COMMAND_ARRAY_SIZE   (0 + CC_Vendor_TCG_Test)
990     #define COMMAND_COUNT       (LIBRARY_COMMAND_ARRAY_SIZE + VENDOR_COMMAND_ARRAY_SIZE)
991     #define HASH_COUNT          \
992                 (ALG_SHA1 + ALG_SHA256 + ALG_SHA384 + ALG_SHA512 + ALG_SM3_256)
993     #define MAX_HASH_BLOCK_SIZE \
994                 (MAX(ALG_SHA1    * SHA1_BLOCK_SIZE,                                  \
995                  MAX(ALG_SHA256  * SHA256_BLOCK_SIZE,                                \
996                  MAX(ALG_SHA384  * SHA384_BLOCK_SIZE,                                \
997                  MAX(ALG_SHA512  * SHA512_BLOCK_SIZE,                                \
998                  MAX(ALG_SM3_256 * SM3_256_BLOCK_SIZE,                               \
999                  0))))))
1000    #define MAX_DIGEST_SIZE     \
1001                (MAX(ALG_SHA1    * SHA1_DIGEST_SIZE,                                \
1002                 MAX(ALG_SHA256  * SHA256_DIGEST_SIZE,                              \
1003                 MAX(ALG_SHA384  * SHA384_DIGEST_SIZE,                              \
1004                 MAX(ALG_SHA512  * SHA512_DIGEST_SIZE,                              \
1005                 MAX(ALG_SM3_256 * SM3_256_DIGEST_SIZE,                             \
1006                 0))))))
1007    #if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
1008    #error "Hash data not valid"
1009    #endif
```

Define the 2B structure that would hold any hash block

```
TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
```

Following typedef is for some old code

```
typedef TPM2B_MAX_HASH_BLOCK    TPM2B_HASH_BLOCK;
```

Additional symmetric constants

```
#ifndef ALG_AES
#define ALG_AES                         NO
#endif
#ifndef MAX_AES_KEY_BITS
#define MAX_AES_KEY_BITS                0
#define MAX_AES_BLOCK_SIZE_BYTES        0
#endif
#ifndef ALG_CAMELLIA
#define ALG_CAMELLIA                    NO
#endif
#ifndef MAX_CAMELLIA_KEY_BITS
#define MAX_CAMELLIA_KEY_BITS           0
#define MAX_CAMELLIA_BLOCK_SIZE_BYTES   0
#endif
#ifndef ALG_SM4
#define ALG_SM4                         NO
#endif
#ifndef MAX_SM4_KEY_BITS
#define MAX_SM4_KEY_BITS                0
#define MAX_SM4_BLOCK_SIZE_BYTES        0
#endif
#ifndef ALG_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(ALG_AES      * MAX_AES_KEY_BITS,                          \
             MAX(ALG_CAMELLIA * MAX_CAMELLIA_KEY_BITS,                     \
             MAX(ALG_SM4      * MAX_SM4_KEY_BITS,                          \
             MAX(ALG_TDES     * MAX_TDES_KEY_BITS,                         \
             0)))))
#define MAX_SYM_KEY_BYTES               ((MAX_SYM_KEY_BITS + 7) / 8)
#define MAX_SYM_BLOCK_SIZE              \
            (MAX(ALG_AES      * MAX_AES_BLOCK_SIZE_BYTES,                  \
             MAX(ALG_CAMELLIA * MAX_CAMELLIA_BLOCK_SIZE_BYTES,             \
             MAX(ALG_SM4      * MAX_SM4_BLOCK_SIZE_BYTES,                  \
             MAX(ALG_TDES     * MAX_TDES_BLOCK_SIZE_BYTES,                 \
             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
#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>
```

B.2.2.1.2. Links to the OpenSSL HASH code

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
#if     ALG_SM3_256
#   error "The version of OpenSSL used by this code does not support SM3"
#endif
```

The defines below are only needed when compiling CryptHash.c or CryptSmac.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)                                                   \
```



```
19                  ((hashState)->def->method.start)(&(hashState)->state);
```

Add data to the hash

```
20  #define HASH_DATA_METHOD_DEF                                                \
21                  void (HASH_DATA_METHOD)(PANY_HASH_STATE state,              \
22                                          PCBYTE buffer,                      \
23                                          size_t size)
24  #define HASH_DATA(hashState, dInSize, dIn)                                  \
25                  ((hashState)->def->method.data)(&(hashState)->state, dIn, dInSize)
```

Finalize the hash and get the digest

```
26  #define HASH_END_METHOD_DEF                                                 \
27                  void (HASH_END_METHOD)(BYTE *buffer, PANY_HASH_STATE state)
28  #define HASH_END(hashState, buffer)                                         \
29                  ((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.

```
30  #define HASH_STATE_COPY_METHOD_DEF                                          \
31                  void (HASH_STATE_COPY_METHOD)(PANY_HASH_STATE to,           \
32                                                PCANY_HASH_STATE from,        \
33                                                size_t size)
34  #define HASH_STATE_COPY(hashStateOut, hashStateIn)                          \
35                  ((hashStateIn)->def->method.copy)(&(hashStateOut)->state,   \
36                                                &(hashStateIn)->state,        \
37                                                (hashStateIn)->def->contextSize)
```

Copy (with reformatting when necessary) an internal hash structure to an external blob

```
38  #define  HASH_STATE_EXPORT_METHOD_DEF                                       \
39                  void (HASH_STATE_EXPORT_METHOD)(BYTE *to,                   \
40                                                  PCANY_HASH_STATE from,      \
41                                                  size_t size)
42  #define  HASH_STATE_EXPORT(to, hashStateFrom)                               \
43                  ((hashStateFrom)->def->method.copyOut)                      \
44                          (&(((BYTE *)(to))[offsetof(HASH_STATE, state)]),    \
45                           &(hashStateFrom)->state,                           \
46                           (hashStateFrom)->def->contextSize)
```

Copy from an external blob to an internal formate (with reformatting when necessary

```
47  #define  HASH_STATE_IMPORT_METHOD_DEF                                       \
48                  void (HASH_STATE_IMPORT_METHOD)(PANY_HASH_STATE to,         \
49                                                  const BYTE *from,           \
50                                                   size_t size)
51  #define  HASH_STATE_IMPORT(hashStateTo, from)                               \
52                  ((hashStateTo)->def->method.copyIn)                         \
53                          (&(hashStateTo)->state,                             \
54                           &(((const BYTE *)(from))[offsetof(HASH_STATE, state)]),\
55                           (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

```
56  #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

```
1   #ifndef _TPM_TO_OSSL_MATH_H
2   #define _TPM_TO_OSSL_MATH_H
3   #if MATH_LIB == OSSL
4   #include <openssl/evp.h>
5   #include <openssl/ec.h>
6   #if OPENSSL_VERSION_NUMBER >= 0x10100000L
7   #include <openssl/bn_lcl.h>
8   #endif
9   #include <openssl/bn.h>
```

B.2.2.2.2. Macros and Defines

Make sure that the library is using the correct size for a crypt word

```
10  #if    defined THIRTY_TWO_BIT && (RADIX_BITS != 32)  \
11      || ((defined SIXTY_FOUR_BIT_LONG || defined SIXTY_FOUR_BIT) \
12          && (RADIX_BITS != 64))
13  #  error "Ossl library is using different radix"
14  #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.

```
15  #define BIG_VAR(name, bits)                                    \
16      BN_VAR(name##Bn, (bits));                                  \
17      BIGNUM          _##name;                                   \
18      BIGNUM          *name = BigInitialized(&_##name,           \
19                              BnInit(name##Bn,                   \
20                              BYTES_TO_CRYPT_WORDS(sizeof(_##name##Bn.d))))
```

Allocate a BIGNUM and initialize with the values in a *bigNum* initializer

```
21  #define BIG_INITIALIZED(name, initializer)                     \
22      BIGNUM          _##name;                                   \
23      BIGNUM          *name = BigInitialized(&_##name, initializer)
24  typedef struct
25  {
26      const ECC_CURVE_DATA    *C;     // the TPM curve values
27      EC_GROUP                *G;     // group parameters
28      BN_CTX                  *CTX;   // the context for the math (this might not be
29                                      // the context in which the curve was created>;
30  } OSSL_CURVE_DATA;
31  typedef OSSL_CURVE_DATA      *bigCurve;
32  #define AccessCurveData(E)       ((E)->C)
33  #define CURVE_INITIALIZED(name, initializer)                   \
34      OSSL_CURVE_DATA     _##name;                               \
35      bigCurve            name =  BnCurveInitialize(&_##name, initializer)
36  #include "TpmToOsslSupport_fp.h"
37  #define CURVE_FREE(E)                   \
38      if(E != NULL)                       \
39      {                                   \
40          if(E->G != NULL)                \
41              EC_GROUP_free(E->G);        \
```

```
42          OsslContextLeave(E->CTX);       \
43      }
44  #define OSSL_ENTER()     BN_CTX      *CTX = OsslContextEnter()
45  #define OSSL_LEAVE()     OsslContextLeave(CTX)
```

This definition would change if there were something to report

```
46  #define MathLibSimulationEnd()
47  #endif // MATH_LIB == OSSL
48  #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.

```
1   #ifndef _TPM_TO_OSSL_SYM_H_
2   #define _TPM_TO_OSSL_SYM_H_
3   #if SYM_LIB == OSSL
4   #include <openssl/aes.h>
5   #include <openssl/des.h>
6   #include <openssl/bn.h>
7   #include <openssl/ossl_typ.h>
```

B.2.2.3.2. Links to the OpenSSL AES code

```
8   #if     ALG_SM4
9   #error "SM4 is not available"
10  #endif
11  #if     ALG_CAMELLIA
12  #error "Camellia is not available"
13  #endif
```

Define the order of parameters to the library functions that do block encryption and decryption.

```
14  typedef void(*TpmCryptSetSymKeyCall_t)(
15      const BYTE  *in,
16      BYTE        *out,
17      void *keySchedule
18      );
```

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.

```
19  #define SWIZZLE(keySchedule, in, out)                                   \
20      (const BYTE *)(in), (BYTE *)(out), (void *)(keySchedule)
```

Macros to set up the encryption/decryption key schedules

AES:

```
21  #define TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)          \
22      AES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
23  #define TpmCryptSetDecryptKeyAES(key, keySizeInBits, schedule)          \
24      AES_set_decrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
```

TDES:

```
25  #define TpmCryptSetEncryptKeyTDES(key, keySizeInBits, schedule)         \
26      TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *)(schedule))
27  #define TpmCryptSetDecryptKeyTDES(key, keySizeInBits, schedule)         \
28      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  #if     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

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

```
3   void
4   TDES_set_encrypt_key(
5       const BYTE                  *key,
6       UINT16                       keySizeInBits,
7       tpmKeyScheduleTDES          *keySchedule
8       )
9   {
10      DES_set_key_unchecked((const_DES_cblock *)key, &keySchedule[0]);
11      DES_set_key_unchecked((const_DES_cblock *)&key[8], &keySchedule[1]);
12      // If is two-key, copy the schedule for K1 into K3, otherwise, compute the
13      // the schedule for K3
14      if(keySizeInBits == 128)
15          keySchedule[2] = keySchedule[0];
16      else
17          DES_set_key_unchecked((const_DES_cblock *)&key[16],
18                                &keySchedule[2]);
19  }
```

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.

```
20  void TDES_encrypt(
21      const BYTE              *in,
22      BYTE                    *out,
23      tpmKeyScheduleTDES      *ks
24      )
25  {
26      DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
27                       &ks[0], &ks[1], &ks[2],
28                       DES_ENCRYPT);
29  }
```


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

```
1   #include "Tpm.h"
2   #if MATH_LIB == OSSL
3   #include "TpmToOsslMath_fp.h"
```

B.2.3.2.3. Functions

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.

```
4   void
5   OsslToTpmBn(
6       bigNum          bn,
7       BIGNUM          *osslBn
8       )
9   {
10      if(bn != NULL)
11      {
12          if((crypt_uword_t *)osslBn->d != bn->d)
13          {
14              int         i;
15              pAssert((unsigned)osslBn->top <= BnGetAllocated(bn));
16              for(i = 0; i < osslBn->top; i++)
17                  bn->d[i] = osslBn->d[i];
18          }
19          BnSetTop(bn, osslBn->top);
20      }
21  }
```

B.2.3.2.3.2. BigInitialized()

This function initializes an OSSL BIGNUM from a TPM bignum.

```
22  BIGNUM *
23  BigInitialized(
24      BIGNUM              *toInit,
25      bigConst             initializer
26      )
27  {
```

```
28      if(toInit == NULL || initializer == NULL)
29          return NULL;
30      toInit->d = (BN_ULONG *)&initializer->d[0];
31      toInit->dmax = initializer->allocated;
32      toInit->top = initializer->size;
33      toInit->neg = 0;
34      toInit->flags = 0;
35      return toInit;
36  }
37  #ifndef OSSL_DEBUG
38  #   define BIGNUM_PRINT(label, bn, eol)
39  #   define DEBUG_PRINT(x)
40  #else
41  #   define DEBUG_PRINT(x)   printf("%s", x)
42  #   define BIGNUM_PRINT(label, bn, eol) BIGNUM_print((label), (bn), (eol))
43  static
44  void BIGNUM_print(
45      const char      *label,
46      const BIGNUM    *a,
47      BOOL             eol
48      )
49  {
50      BN_ULONG        *d;
51      int              i;
52      int              notZero = FALSE;
53
54      if(label != NULL)
55          printf("%s", label);
56      if(a == NULL)
57      {
58          printf("NULL");
59          goto done;
60      }
61      if (a->neg)
62          printf("-");
63      for(i = a->top, d = &a->d[i - 1]; i > 0; i--)
64      {
65          int         j;
66          BN_ULONG    l = *d--;
67          for(j = BN_BITS2 - 8; j >= 0; j -= 8)
68          {
69              BYTE    b = (BYTE)((l >> j) & 0xFF);
70              notZero = notZero || (b != 0);
71              if(notZero)
72                  printf("%02x", b);
73          }
74          if(!notZero)
75              printf("0");
76      }
77  done:
78      if(eol)
79          printf("\n");
80      return;
81  }
82  #endif
83  #ifdef LIBRARY_COMPATIBILITY_CHECK
84  void
85  MathLibraryCompatibilityCheck(
86      void
87      )
88  {
89      OSSL_ENTER();
90      BIGNUM          *osslTemp = BN_CTX_get(CTX);
91      BN_VAR(tpmTemp, 64 * 8); // allocate some space for a test value
92      crypt_uword_t           i;
93      TPM2B_TYPE(TEST, 16);
```



```
94      TPM2B_TEST              test = {{16, {0x0F, 0x0E, 0x0D, 0x0C,
95                                            0x0B, 0x0A, 0x09, 0x08,
96                                            0x07, 0x06, 0x05, 0x04,
97                                            0x03, 0x02, 0x01, 0x00}}};
98      // Convert the test TPM2B to a bigNum
99      BnFrom2B(tpmTemp, &test.b);
100     // Convert the test TPM2B to an OpenSSL BIGNUM
101     BN_bin2bn(test.t.buffer, test.t.size, osslTemp);
102     // Make sure the values are consistent
103     cAssert(osslTemp->top == (int)tpmTemp->size);
104     for(i = 0; i < tpmTemp->size; i++)
105         cAssert(osslTemp->d[0] == tpmTemp->d[0]);
106     OSSL_LEAVE();
107 }
108 #endif
```

B.2.3.2.3.3. BnModMult()

Does multiply and divide returning the remainder of the divide.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
109 LIB_EXPORT BOOL
110 BnModMult(
111     bigNum              result,
112     bigConst            op1,
113     bigConst            op2,
114     bigConst            modulus
115     )
116 {
117     OSSL_ENTER();
118     BIG_INITIALIZED(bnResult, result);
119     BIG_INITIALIZED(bnOp1, op1);
120     BIG_INITIALIZED(bnOp2, op2);
121     BIG_INITIALIZED(bnMod, modulus);
122     BIG_VAR(bnTemp, (LARGEST_NUMBER_BITS * 4));
123     BOOL                OK;
124     pAssert(BnGetAllocated(result) >= BnGetSize(modulus));
125     OK = BN_mul(bnTemp, bnOp1, bnOp2, CTX);
126     OK = OK && BN_div(NULL, bnResult, bnTemp, bnMod, CTX);
127     if(OK)
128     {
129         result->size = bnResult->top;
130         OsslToTpmBn(result, bnResult);
131     }
132     OSSL_LEAVE();
133     return OK;
134 }
```

B.2.3.2.3.4. BnMult()

Multiplies two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
135 LIB_EXPORT BOOL
136 BnMult(
137     bigNum              result,
138     bigConst            multiplicand,
139     bigConst            multiplier
140     )
141 {
142     OSSL_ENTER();
143     BN_VAR(temp, (LARGEST_NUMBER_BITS * 2));
144     BIG_INITIALIZED(bnTemp, temp);
145     BIG_INITIALIZED(bnA, multiplicand);
146     BIG_INITIALIZED(bnB, multiplier);
147     BOOL                OK;
148     pAssert(result->allocated >=
149             (BITS_TO_CRYPT_WORDS(BnSizeInBits(multiplicand)
150                                  + BnSizeInBits(multiplier))));
151     OK = BN_mul(bnTemp, bnA, bnB, CTX);
152     if(OK)
153     {
154         OsslToTpmBn(temp, bnTemp);
155         BnCopy(result, temp);
156     }
157     OSSL_LEAVE();
158     return OK;
159 }
```

B.2.3.2.3.5. BnDiv()

This function divides two *bigNum* values. The function returns FALSE if there is an error in the operation.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
160 LIB_EXPORT BOOL
161 BnDiv(
162     bigNum              quotient,
163     bigNum              remainder,
164     bigConst            dividend,
165     bigConst            divisor
166     )
167 {
168     OSSL_ENTER();
169     BIG_INITIALIZED(bnQ, quotient);
170     BIG_INITIALIZED(bnR, remainder);
171     BIG_INITIALIZED(bnDend, dividend);
172     BIG_INITIALIZED(bnSor, divisor);
173     BOOL        OK;
174     pAssert(!BnEqualZero(divisor));
175     if(BnGetSize(dividend) < BnGetSize(divisor))
176     {
177         if(quotient)
178             BnSetWord(quotient, 0);
179         if(remainder)
180             BnCopy(remainder, dividend);
181         OK = TRUE;
```



```
182         }
183         else
184         {
185             pAssert((quotient == NULL)
186                     || (quotient->allocated >= (unsigned)(dividend->size
187                                                 - divisor->size)));
188             pAssert((remainder == NULL)
189                     || (remainder->allocated >= divisor->size));
190             OK = BN_div(bnQ, bnR, bnDend, bnSor, CTX);
191             if(OK)
192             {
193                 OsslToTpmBn(quotient, bnQ);
194                 OsslToTpmBn(remainder, bnR);
195             }
196         }
197         DEBUG_PRINT("In BnDiv:\n");
198         BIGNUM_PRINT("   bnDividend: ", bnDend, TRUE);
199         BIGNUM_PRINT("    bnDivisor: ", bnSor, TRUE);
200         BIGNUM_PRINT("   bnQuotient: ", bnQ, TRUE);
201         BIGNUM_PRINT("  bnRemainder: ", bnR, TRUE);
202         OSSL_LEAVE();
203         return OK;
204     }
205     #if     ALG_RSA
```

B.2.3.2.3.6. BnGcd()

Get the greatest common divisor of two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
206     LIB_EXPORT BOOL
207     BnGcd(
208         bigNum      gcd,            // OUT: the common divisor
209         bigConst    number1,        // IN:
210         bigConst    number2         // IN:
211         )
212     {
213         OSSL_ENTER();
214         BIG_INITIALIZED(bnGcd, gcd);
215         BIG_INITIALIZED(bn1, number1);
216         BIG_INITIALIZED(bn2, number2);
217         BOOL            OK;
218         pAssert(gcd != NULL);
219         OK = BN_gcd(bnGcd, bn1, bn2, CTX);
220         if(OK)
221         {
222             OsslToTpmBn(gcd, bnGcd);
223             gcd->size = bnGcd->top;
224         }
225         OSSL_LEAVE();
226         return OK;
227     }
```

B.2.3.2.3.7. 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

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
228 LIB_EXPORT BOOL
229 BnModExp(
230     bigNum                  result,         // OUT: the result
231     bigConst                number,         // IN: number to exponentiate
232     bigConst                exponent,       // IN:
233     bigConst                modulus         // IN:
234     )
235 {
236     OSSL_ENTER();
237     BIG_INITIALIZED(bnResult, result);
238     BIG_INITIALIZED(bnN, number);
239     BIG_INITIALIZED(bnE, exponent);
240     BIG_INITIALIZED(bnM, modulus);
241     BOOL            OK;
242 //
243     OK = BN_mod_exp(bnResult, bnN, bnE, bnM, CTX);
244     if(OK)
245     {
246         OsslToTpmBn(result, bnResult);
247     }
248     OSSL_LEAVE();
249     return OK;
250 }
```

B.2.3.2.3.8. BnModInverse()

Modular multiplicative inverse

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
251 LIB_EXPORT BOOL
252 BnModInverse(
253     bigNum                  result,
254     bigConst                number,
255     bigConst                modulus
256     )
257 {
258     OSSL_ENTER();
259     BIG_INITIALIZED(bnResult, result);
260     BIG_INITIALIZED(bnN, number);
261     BIG_INITIALIZED(bnM, modulus);
262     BOOL                OK;
263
264     OK = (BN_mod_inverse(bnResult, bnN, bnM, CTX) != NULL);
265     if(OK)
266     {
267         OsslToTpmBn(result, bnResult);
268     }
269     OSSL_LEAVE();
270     return OK;
271 }
272 #endif // ALG_RSA
273 #if     ALG_ECC
```


B.2.3.2.3.9. PointFromOssl()

Function to copy the point result from an OSSL function to a *bigNum*

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```
274 static BOOL
275 PointFromOssl(
276     bigPoint         pOut,       // OUT: resulting point
277     EC_POINT        *pIn,        // IN: the point to return
278     bigCurve         E           // IN: the curve
279     )
280 {
281     BIGNUM         *x = NULL;
282     BIGNUM         *y = NULL;
283     BOOL            OK;
284     BN_CTX_start(E->CTX);
285 //
286     x = BN_CTX_get(E->CTX);
287     y = BN_CTX_get(E->CTX);
288
289     if(y == NULL)
290         FAIL(FATAL_ERROR_ALLOCATION);
291     // If this returns false, then the point is at infinity
292     OK = EC_POINT_get_affine_coordinates_GFp(E->G, pIn, x, y, E->CTX);
293     if(OK)
294     {
295         OsslToTpmBn(pOut->x, x);
296         OsslToTpmBn(pOut->y, y);
297         BnSetWord(pOut->z, 1);
298     }
299     else
300         BnSetWord(pOut->z, 0);
301     BN_CTX_end(E->CTX);
302     return OK;
303 }
```

B.2.3.2.3.10. EcPointInitialized()

Allocate and initialize a point.

```
304 static EC_POINT *
305 EcPointInitialized(
306     pointConst          initializer,
307     bigCurve            E
308     )
309 {
310     BIG_INITIALIZED(bnX, (initializer != NULL) ? initializer->x : NULL);
311     BIG_INITIALIZED(bnY, (initializer != NULL) ? initializer->y : NULL);
312
313     EC_POINT            *P = (initializer != NULL && E != NULL)
314                                 ? EC_POINT_new(E->G) : NULL;
315     pAssert(E != NULL);
316     if(P != NULL)
317         EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, E->CTX);
318     return P;
319 }
```


B.2.3.2.3.11. 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*

```
320 bigCurve
321 BnCurveInitialize(
322     bigCurve            E,          // IN: curve structure to initialize
323     TPM_ECC_CURVE       curveId     // IN: curve identifier
324     )
325 {
326     EC_GROUP                *group = NULL;
327     EC_POINT                *P = NULL;
328     const ECC_CURVE_DATA    *C = GetCurveData(curveId);
329     BN_CTX                  *CTX = NULL;
330
331     BIG_INITIALIZED(bnP, C != NULL ? C->prime : NULL);
332     BIG_INITIALIZED(bnA, C != NULL ? C->a : NULL);
333     BIG_INITIALIZED(bnB, C != NULL ? C->b : NULL);
334     BIG_INITIALIZED(bnX, C != NULL ? C->base.x : NULL);
335     BIG_INITIALIZED(bnY, C != NULL ? C->base.y : NULL);
336     BIG_INITIALIZED(bnN, C != NULL ? C->order : NULL);
337     BIG_INITIALIZED(bnH, C != NULL ? C->h : NULL);
338     int                     OK = (C != NULL);
339 //
340     OK = OK && ((CTX = OsslContextEnter()) != NULL);
341
342     // initialize EC group, associate a generator point and initialize the point
343     // from the parameter data
344     // Create a group structure
345     OK = OK && (group = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, CTX)) != NULL;
346
347     // Allocate a point in the group that will be used in setting the
348     // generator. This is not needed after the generator is set.
349     OK = OK && ((P = EC_POINT_new(group)) != NULL);
350     // Need to use this in case Montgomery method is being used
351     OK = OK
352         && EC_POINT_set_affine_coordinates_GFp(group, P, bnX, bnY, CTX);
353     // Now set the generator
354     OK = OK && EC_GROUP_set_generator(group, P, bnN, bnH);
355
356     if(P != NULL)
357         EC_POINT_free(P);
358
359     if(!OK && group != NULL)
360     {
361         EC_GROUP_free(group);
362         group = NULL;
363     }
364     if(!OK && CTX != NULL)
365     {
366         OsslContextLeave(CTX);
367         CTX = NULL;
368     }
369
370     E->G = group;
371     E->CTX = CTX;
372     E->C = C;
```



```
373
374         return OK ? E : NULL;
375     }
```

B.2.3.2.3.12. BnEccModMult()

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

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```
376     LIB_EXPORT BOOL
377     BnEccModMult(
378         bigPoint            R,          // OUT: computed point
379         pointConst          S,          // IN: point to multiply by 'd' (optional)
380         bigConst            d,          // IN: scalar for [d]S
381         bigCurve            E
382         )
383     {
384         EC_POINT           *pR = EC_POINT_new(E->G);
385         EC_POINT           *pS = EcPointInitialized(S, E);
386         BIG_INITIALIZED(bnD, d);
387
388         if(S == NULL)
389             EC_POINT_mul(E->G, pR, bnD, NULL, NULL, E->CTX);
390         else
391             EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
392         PointFromOssl(R, pR, E);
393         EC_POINT_free(pR);
394         EC_POINT_free(pS);
395         return !BnEqualZero(R->z);
396     }
```

B.2.3.2.3.13. BnEccModMult2()

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

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```
397     LIB_EXPORT BOOL
398     BnEccModMult2(
399         bigPoint            R,          // OUT: computed point
400         pointConst          S,          // IN: optional point
401         bigConst            d,          // IN: scalar for [d]S or [d]G
402         pointConst          Q,          // IN: second point
403         bigConst            u,          // IN: second scalar
404         bigCurve            E           // IN: curve
405         )
406     {
407         EC_POINT           *pR = EC_POINT_new(E->G);
408         EC_POINT           *pS = EcPointInitialized(S, E);
409         BIG_INITIALIZED(bnD, d);
410         EC_POINT           *pQ = EcPointInitialized(Q, E);
411         BIG_INITIALIZED(bnU, u);
412
413         if(S == NULL || S == (pointConst)&(AccessCurveData(E)->base))
```

```
414             EC_POINT_mul(E->G, pR, bnD, pQ, bnU, E->CTX);
415         else
416         {
417             const EC_POINT        *points[2];
418             const BIGNUM          *scalars[2];
419             points[0] = pS;
420             points[1] = pQ;
421             scalars[0] = bnD;
422             scalars[1] = bnU;
423             EC_POINTs_mul(E->G, pR, NULL, 2, points, scalars, E->CTX);
424         }
425         PointFromOssl(R, pR, E);
426         EC_POINT_free(pR);
427         EC_POINT_free(pS);
428         EC_POINT_free(pQ);
429         return !BnEqualZero(R->z);
430     }
```

B.2.3.2.4. BnEccAdd()

This function does addition of two points.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```
431     LIB_EXPORT BOOL
432     BnEccAdd(
433         bigPoint            R,          // OUT: computed point
434         pointConst          S,          // IN: point to multiply by 'd'
435         pointConst          Q,          // IN: second point
436         bigCurve            E           // IN: curve
437         )
438     {
439         EC_POINT           *pR = EC_POINT_new(E->G);
440         EC_POINT           *pS = EcPointInitialized(S, E);
441         EC_POINT           *pQ = EcPointInitialized(Q, E);
442     //
443         EC_POINT_add(E->G, pR, pS, pQ, E->CTX);
444
445         PointFromOssl(R, pR, E);
446         EC_POINT_free(pR);
447         EC_POINT_free(pS);
448         EC_POINT_free(pQ);
449         return !BnEqualZero(R->z);
450     }
451     #endif // ALG_ECC
452     #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. Description

This module simulates the cancel pins on the TPM.

C.2.2. Includes, Typedefs, Structures, and Defines

```
#include "PlatformData.h"
#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

```
LIB_EXPORT int
_plat__IsCanceled(
    void
    )
{
    // return cancel flag
    return s_isCanceled;
}
```

C.2.3.2. _plat__SetCancel()

Set cancel flag.

```
LIB_EXPORT void
_plat__SetCancel(
    void
    )
{
    s_isCanceled = TRUE;
    return;
}
```

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. Description

This file contains the routines that are used by the simulator to mimic a hardware clock on a TPM.

In this implementation, all the time values are measured in millisecond. However, the precision of the clock functions may be implementation dependent.

C.3.2. Includes and Data Definitions

```
1   #include "PlatformData.h"
2   #include "Platform_fp.h"
3   #include "TpmFail_fp.h"
4   #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. When the clock is reset, the indication that the clock was stopped is also set.

```
5   LIB_EXPORT void
6   _plat__TimerReset(
7       void
8       )
9   {
10      s_lastSystemTime = 0;
11      s_tpmTime = 0;
12      s_adjustRate = CLOCK_NOMINAL;
13      s_timerReset = TRUE;
14      s_timerStopped = TRUE;
15      return;
16  }
```

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.

```
17  LIB_EXPORT void
18  _plat__TimerRestart(
19      void
20      )
21  {
22      s_timerStopped = TRUE;
23      return;
24  }
```


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.

```
25  #include <time.h>
26  clock_t     debugTime;
```

C.3.4.2. _plat__RealTime()

This is another, probably futile, attempt to define a portable function that will return a 64-bit clock value that has *mSec* resolution.

```
27  uint64_t
28  _plat__RealTime(
29      void
30  )
31  {
32      clock64_t           time;
33  #ifdef _MSC_VER
34      struct _timeb       sysTime;
35  //
36      _ftime_s(&sysTime);
37      time = (clock64_t)(sysTime.time) * 1000 + sysTime.millitm;
38      // set the time back by one hour if daylight savings
39      if(sysTime.dstflag)
40          time -= 1000 * 60 * 60;  // mSec/sec * sec/min * min/hour = ms/hour
41  #else
42      // hopefully, this will work with most UNIX systems
43      struct timespec     systime;
44  //
45      clock_gettime(CLOCK_MONOTONIC, &systime);
46      time = (clock64_t)systime.tv_sec * 1000 + (systime.tv_nsec / 1000000);
47  #endif
48      return time;
49  }
```

C.3.4.3. _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().

```
50  LIB_EXPORT uint64_t
51  _plat__TimerRead(
52      void
53      )
54  {
55  #ifdef HARDWARE_CLOCK
56  #error      "need a defintion for reading the hardware clock"
57      return HARDWARE_CLOCK
```



```
58  #else
59      clock64_t         timeDiff;
60      clock64_t         adjustedTimeDiff;
61      clock64_t         timeNow;
62      clock64_t         readjustedTimeDiff;
63
64      // This produces a timeNow that is basically locked to the system clock.
65      timeNow = _plat__RealTime();
66
67      // if this hasn't been initialized, initialize it
68      if(s_lastSystemTime == 0)
69      {
70          s_lastSystemTime = timeNow;
71          debugTime = clock();
72          s_lastReportedTime = 0;
73          s_realTimePrevious = 0;
74      }
75      // The system time can bounce around and that's OK as long as we don't allow
76      // time to go backwards. When the time does appear to go backwards, set
77      // lastSystemTime to be the new value and then update the reported time.
78      if(timeNow < s_lastReportedTime)
79          s_lastSystemTime = timeNow;
80      s_lastReportedTime = s_lastReportedTime + timeNow - s_lastSystemTime;
81      s_lastSystemTime = timeNow;
82      timeNow = s_lastReportedTime;
83
84      // The code above produces a timeNow that is similar to the value returned
85      // by Clock(). The difference is that timeNow does not max out, and it is
86      // at a ms. rate rather than at a CLOCKS_PER_SEC rate. The code below
87      // uses that value and does the rate adjustment on the time value.
88      // If there is no difference in time, then skip all the computations
89      if(s_realTimePrevious >= timeNow)
90          return s_tpmTime;
91      // Compute the amount of time since the last update of the system clock
92      timeDiff = timeNow - s_realTimePrevious;
93
94      // Do the time rate adjustment and conversion from CLOCKS_PER_SEC to mSec
95      adjustedTimeDiff = (timeDiff * CLOCK_NOMINAL) / ((uint64_t)s_adjustRate);
96
97      // update the TPM time with the adjusted timeDiff
98      s_tpmTime += (clock64_t)adjustedTimeDiff;
99
100     // Might have some rounding error that would loose CLOCKS. See what is not
101     // being used. As mentioned above, this could result in putting back more than
102     // is taken out. Here, we are trying to recreate timeDiff.
103     readjustedTimeDiff = (adjustedTimeDiff * (uint64_t)s_adjustRate )
104                                 / CLOCK_NOMINAL;
105
106     // adjusted is now converted back to being the amount we should advance the
107     // previous sampled time. It should always be less than or equal to timeDiff.
108     // That is, we could not have use more time than we started with.
109     s_realTimePrevious = s_realTimePrevious + readjustedTimeDiff;
110
111 #ifdef  DEBUGGING_TIME
112     // Put this in so that TPM time will pass much faster than real time when
113     // doing debug.
114     // A value of 1000 for DEBUG_TIME_MULTIPLER will make each ms into a second
115     // A good value might be 100
116     return (s_tpmTime * DEBUG_TIME_MULTIPLIER);
117 #endif
118     return s_tpmTime;
119 #endif
120 }
```

C.3.4.4. _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.

```
121 LIB_EXPORT BOOL
122 _plat__TimerWasReset(
123    void
124     )
125 {
126     BOOL         retVal = s_timerReset;
127     s_timerReset = FALSE;
128     return retVal;
129 }
```

C.3.4.5. _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.

```
130 LIB_EXPORT BOOL
131 _plat__TimerWasStopped(
132     void
133     )
134 {
135     BOOL         retVal = s_timerStopped;
136     s_timerStopped = FALSE;
137     return retVal;
138 }
```

C.3.4.6. _plat__ClockAdjustRate()

Adjust the clock rate

```
139 LIB_EXPORT void
140 _plat__ClockAdjustRate(
141     int              adjust         // IN: the adjust number.  It could be positive
142                                     //     or negative
143     )
144 {
145     // We expect the caller should only use a fixed set of constant values to
146     // adjust the rate
147     switch(adjust)
148     {
149         case CLOCK_ADJUST_COARSE:
150             s_adjustRate += CLOCK_ADJUST_COARSE;
151             break;
152         case -CLOCK_ADJUST_COARSE:
153             s_adjustRate -= CLOCK_ADJUST_COARSE;
154             break;
155         case CLOCK_ADJUST_MEDIUM:
156             s_adjustRate += CLOCK_ADJUST_MEDIUM;
157             break;
158         case -CLOCK_ADJUST_MEDIUM:
159             s_adjustRate -= CLOCK_ADJUST_MEDIUM;
160             break;
```



```
161         case CLOCK_ADJUST_FINE:
162             s_adjustRate += CLOCK_ADJUST_FINE;
163             break;
164         case -CLOCK_ADJUST_FINE:
165             s_adjustRate -= CLOCK_ADJUST_FINE;
166             break;
167         default:
168             // ignore any other values;
169             break;
170     }
171
172     if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
173         s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
174     if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
175         s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;
176
177     return;
178 }
```

C.4 Entropy.c

C.4.1. Includes and Local Values

```
#define _CRT_RAND_S
#include <stdlib.h>
#include <memory.h>
#include "PlatformData.h"
#include "Platform_fp.h"
#include <time.h>
#ifdef _MSC_VER
#include <process.h>
#else
#include <unistd.h>
#endif
```

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;
```

C.4.2. Functions

C.4.2.1. rand32()

Local function to get a 32-bit random number

```
static uint32_t
rand32(
    void
)
{
    uint32_t    rndNum = rand();
#if RAND_MAX < UINT16_MAX
    // If the maximum value of the random number is a 15-bit number, then shift it up
    // 15 bits, get 15 more bits, shift that up 2 and then XOR in another value to get
    // a full 32 bits.
    rndNum = (rndNum << 15) ^ rand();
    rndNum = (rndNum << 2) ^ rand();
#elif RAND_MAX == UINT16_MAX
    // If the maximum size is 16-bits, shift it and add another 16 bits
    rndNum = (rndNum << 16) ^ rand();
#elif RAND_MAX < UINT32_MAX
    // If 31 bits, then shift 1 and include another random value to get the extra bit
    rndNum = (rndNum << 1) ^ rand();
#endif
    return rndNum;
}
```


C.4.2.2. _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.

Return Value	Meaning
< 0	hardware failure of the entropy generator, this is sticky
>= 0	the returned amount of entropy (bytes)

```
34  LIB_EXPORT int32_t
35  _plat__GetEntropy(
36      unsigned char        *entropy,           // output buffer
37      uint32_t              amount             // amount requested
38  )
39  {
40      uint32_t             rndNum;
41      int32_t              ret;
42  //
43      if(amount == 0)
44      {
45          // Seed the platform entropy source if the entropy source is software. There
46          // is no reason to put a guard macro (#if or #ifdef) around this code because
47          // this code would not be here if someone was changing it for a system with
48          // actual hardware.
49          //
50          // NOTE 1: The following command does not provide proper cryptographic
51          // entropy. Its primary purpose to make sure that different instances of the
52          // simulator, possibly started by a script on the same machine, are seeded
53          // differently. Vendors of the actual TPMs need to ensure availability of
54          // proper entropy using their platform-specific means.
55          //
56          // NOTE 2: In debug builds by default the reference implementation will seed
57          // its RNG deterministically (without using any platform provided randomness).
58          // See the USE_DEBUG_RNG macro and DRBG_GetEntropy() function.
59  #ifdef _MSC_VER
60          srand((unsigned)_plat__RealTime() ^ _getpid());
61  #else
62          srand((unsigned)_plat__RealTime() ^ getpid());
63  #endif
64          lastEntropy = rand32();
65          ret = 0;
66      }
67      else
68      {
69          rndNum = rand32();
70          if(rndNum == lastEntropy)
71          {
72              ret = -1;
73          }
74          else
75          {
76              lastEntropy = rndNum;
77              // Each process will have its random number generator initialized
78              // according to the process id and the initialization time. This is not a
79              // lot of entropy so, to add a bit more, XOR the current time value into
80              // the returned entropy value.
81              // NOTE: the reason for including the time here rather than have it in
82              // in the value assigned to lastEntropy is that rand() could be broken and
83              // using the time would in the lastEntropy value would hide this.
84              rndNum ^= (uint32_t)_plat__RealTime();
85
86              // Only provide entropy 32 bits at a time to test the ability
87              // of the caller to deal with partial results.
```

```
88                  ret = MIN(amount, sizeof(rndNum));
89                  memcpy(entropy, &rndNum, ret);
90              }
91          }
92          return ret;
93      }
```

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. Description

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 and Local

```
1   #include <memory.h>
2   #include <string.h>
3   #include <assert.h>
4   #include "PlatformData.h"
5   #include "Platform_fp.h"
6   #if FILE_BACKED_NV
7   #   include         <stdio.h>
8   FILE                *s_NvFile = NULL;
9   #endif
```

C.6.3. Functions

C.6.3.1. NvFileOpen()

This function opens the file used to hold the NV image.

```
10  #if FILE_BACKED_NV
```

Return Value	Meaning
>= 0	success
-1	error

```
11  static int
12  NvFileOpen(
13      const char      *mode
14  )
15  {
16      // Try to open an exist NVChip file for read/write
17  #   if defined _MSC_VER && 1
18      if(fopen_s(&s_NvFile, "NVChip", mode) != 0)
19          s_NvFile = NULL;
20  #   else
21      s_NvFile = fopen("NVChip", mode);
22  #   endif
23      return (s_NvFile == NULL) ? -1 : 0;
24  }
```

C.6.3.2. NvFileCommit()

Write all of the contents of the NV image to a file.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```
25  static int
26  NvFileCommit(
27      void
28  )
29  {
30      int         OK;
31      // If NV file is not available, return failure
32      if(s_NvFile == NULL)
33          return 1;
34      // Write RAM data to NV
35      fseek(s_NvFile, 0, SEEK_SET);
36      OK = (NV_MEMORY_SIZE == fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NvFile));
37      OK = OK && (0 == fflush(s_NvFile));
38      assert(OK);
39      return OK;
40  }
```

C.6.3.3. NvFileSize()

This function gets the size of the NV file and puts the file pointer were desired using the seek method values. SEEK_SET => beginning; SEEK_CUr() => current position and SEEK_END => to the end of the file.

```
41  static long
42  NvFileSize(
43      int         leaveAt
44  )
45  {
46      long    fileSize;
47      long    filePos = ftell(s_NvFile);
48  //
49      assert(NULL != s_NvFile);
50
51      fseek(s_NvFile, 0, SEEK_END);
52      fileSize = ftell(s_NvFile);
53      switch(leaveAt)
54      {
55          case SEEK_SET:
56              filePos = 0;
57          case SEEK_CUR:
58              fseek(s_NvFile, filePos, SEEK_SET);
59              break;
60          case SEEK_END:
61              break;
62          default:
63              assert(FALSE);
64              break;
65      }
66      return fileSize;
67  }
68  #endif
```

C.6.3.4. _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

```
69  LIB_EXPORT void
70  _plat__NvErrors(
71      int                 recoverable,
72      int                 unrecoverable
73      )
74  {
75      s_NV_unrecoverable = unrecoverable;
76      s_NV_recoverable = recoverable;
77  }
```

C.6.3.5. _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

```
78  LIB_EXPORT int
79  _plat__NVEnable(
80      void            *platParameter  // IN: platform specific parameters
81      )
82  {
83      NOT_REFERENCED(platParameter);          // to keep compiler quiet
84  //
85      // Start assuming everything is OK
86      s_NV_unrecoverable = FALSE;
87      s_NV_recoverable = FALSE;
88  #if FILE_BACKED_NV
89      if(s_NvFile != NULL)
90          return 0;
91      // Initialize all the bytes in the ram copy of the NV
92      _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
93
94      // If the file exists
95      if(NvFileOpen("r+b") >= 0)
96      {
97          long    fileSize = NvFileSize(SEEK_SET);    // get the file size and leave the
98                                                      // file pointer at the start
99  //
100         // If the size is right, read the data
101         if(NV_MEMORY_SIZE == fileSize)
102             fread(s_NV, 1, NV_MEMORY_SIZE, s_NvFile);
103         else
104             NvFileCommit();     // for any other size, initialize it
105     }
106     // If NVChip file does not exist, try to create it for read/write.
107     else if(NvFileOpen("w+b") >= 0)
108         NvFileCommit();             // Initialize the file
109     assert(NULL != s_NvFile);       // Just in case we are broken for some reason.
110 #endif
111     // NV contents have been initialized and the error checks have been performed. For
```



```
112         // simulation purposes, use the signaling interface to indicate if an error is
113         // to be simulated and the type of the error.
114         if(s_NV_unrecoverable)
115             return -1;
116         return s_NV_recoverable;
117     }
```

C.6.3.6. _plat__NVDisable()

Disable NV memory

```
118     LIB_EXPORT void
119     _plat__NVDisable(
120         void
121         )
122     {
123     #if  FILE_BACKED_NV
124         if(NULL != s_NvFile)
125             fclose(s_NvFile);     // Close NV file
126         s_NvFile = NULL;         // Set file handle to NULL
127     #endif
128         return;
129     }
```

C.6.3.7. _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

```
130     LIB_EXPORT int
131     _plat__IsNvAvailable(
132         void
133         )
134     {
135         int         retVal = 0;
136         // NV is not available if the TPM is in failure mode
137         if(!s_NvIsAvailable)
138             retVal = 1;
139     #if FILE_BACKED_NV
140         else
141             retVal = (s_NvFile == NULL);
142     #endif
143         return retVal;
144     }
```

C.6.3.8. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```
145     LIB_EXPORT void
146     _plat__NvMemoryRead(
147         unsigned int     startOffset,   // IN: read start
148         unsigned int     size,          // IN: size of bytes to read
149         void            *data           // OUT: data buffer
150         )
```

```
151 {
152     assert(startOffset + size <= NV_MEMORY_SIZE);
153     memcpy(data, &s_NV[startOffset], size);     // Copy data from RAM
154     return;
155 }
```

C.6.3.9. _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

```
156 LIB_EXPORT int
157 _plat__NvIsDifferent(
158     unsigned int     startOffset,   // IN: read start
159     unsigned int     size,          // IN: size of bytes to read
160     void            *data           // IN: data buffer
161     )
162 {
163     return (memcmp(&s_NV[startOffset], data, size) != 0);
164 }
```

C.6.3.10. _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.

```
165 LIB_EXPORT void
166 _plat__NvMemoryWrite(
167     unsigned int     startOffset,   // IN: write start
168     unsigned int     size,          // IN: size of bytes to write
169     void            *data           // OUT: data buffer
170     )
171 {
172     assert(startOffset + size <= NV_MEMORY_SIZE);
173     memcpy(&s_NV[startOffset], data, size);     // Copy the data to the NV image
174 }
```

C.6.3.11. _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.

```
175 LIB_EXPORT void
176 _plat__NvMemoryClear(
177     unsigned int     start,         // IN: clear start
178     unsigned int     size           // IN: number of bytes to clear
179     )
180 {
181     assert(start + size <= NV_MEMORY_SIZE);
182     // In this implementation, assume that the erase value for NV is all 1s
183     memset(&s_NV[start], 0xff, size);
184 }
```


C.6.3.12. _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

```
185 LIB_EXPORT void
186 _plat__NvMemoryMove(
187     unsigned int     sourceOffset,  // IN: source offset
188     unsigned int     destOffset,    // IN: destination offset
189     unsigned int     size           // IN: size of data being moved
190     )
191 {
192     assert(sourceOffset + size <= NV_MEMORY_SIZE);
193     assert(destOffset + size <= NV_MEMORY_SIZE);
194     memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);      // Move data in RAM
195     return;
196 }
```

C.6.3.13. _plat__NvCommit()

This function writes the local copy of NV to NV for permanent store. It will write NV_MEMORY_SIZE bytes to NV. If a file is use, the entire file is written.

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
197 LIB_EXPORT int
198 _plat__NvCommit(
199     void
200     )
201 {
202 #if FILE_BACKED_NV
203     return (NvFileCommit() ? 0 : 1);
204 #else
205     return 0;
206 #endif
207 }
```

C.6.3.14. _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

```
208 LIB_EXPORT void
209 _plat__SetNvAvail(
210     void
211     )
212 {
213     s_NvIsAvailable = TRUE;
214     return;
215 }
```

C.6.3.15. _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

```
216 LIB_EXPORT void
217 _plat__ClearNvAvail(
218     void
219     )
220 {
221     s_NvIsAvailable = FALSE;
222     return;
223 }
```

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
```

```
27  _plat__Signal_Reset(
28      void
29      )
30  {
31      // Initialize locality
32      s_locality = 0;
33
34      // Command cancel
35      s_isCanceled = FALSE;
36
37      _TPM_Init();
38
39      // if we are doing reset but did not have a power failure, then we should
40      // not need to reload NV ...
41
42      return 0;
43  }
```

C.7.2.4. _plat__Signal_PowerOff()

Signal platform power off

```
44  LIB_EXPORT void
45  _plat__Signal_PowerOff(
46      void
47      )
48  {
49      // Prepare NV memory for power off
50      _plat__NVDisable();
51
52      return;
53  }
```

C.8 Platform_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. When the clock is reset, the indication that the clock was stopped is also set.

```
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.

```
19  LIB_EXPORT void
20  _plat__TimerRestart(
21      void
22      );
```

C.8.2.3. _plat__RealTime()

This is another, probably futile, attempt to define a portable function that will return a 64-bit clock value that has *mSec* resolution.

```
23  uint64_t
24  _plat__RealTime(
25      void
26  );
```

C.8.2.4. _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().

```
27  LIB_EXPORT uint64_t
28  _plat__TimerRead(
29      void
30      );
```

C.8.2.5. _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.

```
31  LIB_EXPORT BOOL
32  _plat__TimerWasReset(
33     void
34      );
```

C.8.2.6. _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.

```
35  LIB_EXPORT BOOL
36  _plat__TimerWasStopped(
37      void
38      );
```


C.8.2.7. _plat__ClockAdjustRate()

Adjust the clock rate

```
LIB_EXPORT void
_plat__ClockAdjustRate(
    int             adjust          // IN: the adjust number.  It could be positive
                                    //     or negative
    );
```

C.8.3. From Entropy.c

C.8.3.1. _plat__GetEntropy()

This function is used to get available hardware entropy. In a hardware implementation of this function, there would be no call to the system to get entropy.

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
);
```

C.8.4. From LocalityPlat.c

C.8.4.1. _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
    );
```

C.8.4.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
LIB_EXPORT void
_plat__LocalitySet(
    unsigned char    locality
    );
```

C.8.5. From NVMem.c

C.8.5.1. _plat__NvErrors()

This function is used by the simulator to set the error flags in the NV subsystem to simulate an error in the NV loading process

```
LIB_EXPORT void
_plat__NvErrors(
    int                 recoverable,
    int                 unrecoverable
    );
```

C.8.5.2. _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
    );
```

C.8.5.3. _plat__NVDisable()

Disable NV memory

```
LIB_EXPORT void
_plat__NVDisable(
    void
    );
```

C.8.5.4. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```
LIB_EXPORT int
_plat__IsNvAvailable(
    void
    );
```

C.8.5.5. _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.8.5.6. _plat__NvIsDifferent()

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE(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.8.5.7. _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.8.5.8. _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.8.5.9. _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.8.5.10. _plat__NvCommit()

This function writes the local copy of NV to NV for permanent store. It will write NV_MEMORY_SIZE bytes to NV. If a file is use, the entire file is written.

Return Value	Meaning
0	NV write success
non-0	NV write fail

```
103 LIB_EXPORT int
104 _plat__NvCommit(
105     void
106     );
```

C.8.5.11. _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.8.5.12. _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.8.6. From PowerPlat.c

C.8.6.1. _plat__Signal_PowerOn()

Signal platform power on

```
115 LIB_EXPORT int
116 _plat__Signal_PowerOn(
117     void
118     );
```

C.8.6.2. _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
    );
```

C.8.6.3. _plat_Signal_Reset()

This a TPM reset without a power loss.

```
LIB_EXPORT int
_plat__Signal_Reset(
    void
    );
```

C.8.6.4. _plat__Signal_PowerOff()

Signal platform power off

```
LIB_EXPORT void
_plat__Signal_PowerOff(
    void
    );
```

C.8.7. From PPPlat.c

C.8.7.1. _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
    );
```

C.8.7.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```
LIB_EXPORT void
_plat__Signal_PhysicalPresenceOn(
    void
    );
```


C.8.7.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
139 LIB_EXPORT void
140 _plat__Signal_PhysicalPresenceOff(
141     void
142     );
```

C.8.8. From RunCommand.c

C.8.8.1. _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.8.8.2. _plat__Fail()

This is the platform depended failure exit for the TPM.

```
150 LIB_EXPORT NORETURN void
151 _plat__Fail(
152     void
153     );
```

C.8.9. From Unique.c

C.8.10. _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   #ifdef _MSC_VER
6   #include <sys/types.h>
7   #include <sys/timeb.h>
8   #else
9   #include <sys/time.h>
10  #include <time.h>
11  #endif
12  #ifndef HARDWARE_CLOCK
13  typedef uint64_t     clock64_t;
```

This is the value returned the last time that the system clock was read. This is only relevant for a simulator or virtual TPM.

```
14  extern clock64_t       s_realTimePrevious;
```

These values are used to try to synthesize a long lived version of clock().

```
15  extern clock64_t        s_lastSystemTime;
16  extern clock64_t        s_lastReportedTime;
```

This is the rate adjusted value that is the equivalent of what would be read from a hardware register that produced rate adjusted time.

```
17  extern clock64_t        s_tpmTime;
18  #endif // HARDWARE_CLOCK
```

This value indicates that the timer was reset

```
19  extern BOOL              s_timerReset;
```

This value indicates that the timer was stopped. It causes a clock discontinuity.

```
20  extern BOOL              s_timerStopped;
```

CLOCK_NOMINAL is the number of hardware ticks per *mS*. A value of 300000 means that the nominal clock rate used to drive the hardware clock is 30 MHz(). The adjustment rates are used to determine the conversion of the hardware ticks to internal hardware clock value. In practice, we would expect that there woudl be a hardware register will accumulated *mS*. It would be incremented by the output of a pre-scaler. The pre-scaler would divide the ticks from the clock by some value that would compensate for the difference between clock time and real time. The code in Clock does the emulation of this function.

```
21  #define     CLOCK_NOMINAL           30000
```

A 1% change in rate is 300 counts

```
22  #define     CLOCK_ADJUST_COARSE     300
```

A 0.1% change in rate is 30 counts

```
23  #define     CLOCK_ADJUST_MEDIUM     30
```

A minimum change in rate is 1 count

```
24  #define     CLOCK_ADJUST_FINE       1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

```
25  #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

```
26  extern uint64_t         s_initClock;
```

This variable records the timer adjustment factor.

```
27  extern unsigned int         s_adjustRate;
```

For LocalityPlat.c Locality of current command

```
28  extern unsigned char s_locality;
```

For 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.

```
29  #if (!defined VTPM) || ((VTPM != NO) && (VTPM != YES))
30  #   undef VTPM
31  #   define      VTPM            YES                 // Default: Either YES or NO
32  #endif
```

For a simulation, use a file to back up the NV

```
33  #if (!defined FILE_BACKED_NV) || ((FILE_BACKED_NV != NO) && (FILE_BACKED_NV != YES))
34  #   undef   FILE_BACKED_NV
35  #   define  FILE_BACKED_NV          (VTPM && YES)     // Default: Either YES or NO
36  #endif
37  #if SIMULATION
38  #   undef       FILE_BACKED_NV
39  #   define      FILE_BACKED_NV          YES
40  #endif // SIMULATION
41  extern unsigned char     s_NV[NV_MEMORY_SIZE];
42  extern BOOL              s_NvIsAvailable;
43  extern BOOL              s_NV_unrecoverable;
44  extern BOOL              s_NV_recoverable;
```

For PPPlat.c Physical presence. It is initialized to FALSE

```
45  extern BOOL     s_physicalPresence;
```

From Power

```
46  extern BOOL        s_powerLost;
```

For Entropy.c

```
47  extern uint32_t         lastEntropy;
```



```
48  #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

```
1   #include    "Implementation.h"
2   #include    "PlatformData.h"
```

From Cancel.c

```
3   BOOL                s_isCanceled;
```

From Clock.c

```
4   unsigned int        s_adjustRate;
5   BOOL                s_timerReset;
6   BOOL                s_timerStopped;
7   #ifndef HARDWARE_CLOCK
8   clock64_t           s_realTimePrevious;
9   clock64_t           s_tpmTime;
10  clock64_t           s_lastSystemTime;
11  clock64_t           s_lastReportedTime;
12  #endif
```

From LocalityPlat.c

```
13  unsigned char       s_locality;
```

From Power.c

```
14  BOOL                s_powerLost;
```

From Entropy.c This values is used to determine if the entropy generator is broken. If two consecutive values are the same, then the entropy generator is considered to be broken.

```
15  uint32_t            lastEntropy;
```

For NVMem.c

```
16  unsigned char       s_NV[NV_MEMORY_SIZE];
17  BOOL                s_NvIsAvailable;
18  BOOL                s_NV_unrecoverable;
19  BOOL                s_NV_recoverable;
```

From PPPlat.c

```
20  BOOL  s_physicalPresence;
```


C.11 PPPlat.c

C.11.1. Description

This module simulates the physical presence interface pins on the TPM.

C.11.2. Includes

```
1   #include "PlatformData.h"
2   #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

```
3   LIB_EXPORT int
4   _plat__PhysicalPresenceAsserted(
5       void
6       )
7   {
8       // Do not know how to check physical presence without real hardware.
9       // so always return TRUE;
10      return s_physicalPresence;
11  }
```

C.11.3.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```
12  LIB_EXPORT void
13  _plat__Signal_PhysicalPresenceOn(
14      void
15      )
16  {
17      s_physicalPresence = TRUE;
18      return;
19  }
```

C.11.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
20  LIB_EXPORT void
21  _plat__Signal_PhysicalPresenceOff(
22      void
23      )
24  {
25      s_physicalPresence = FALSE;
26      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

```
1   #include "PlatformData.h"
2   #include "Platform_fp.h"
3   #include <setjmp.h>
4   #include "ExecCommand_fp.h"
5   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.

```
6   LIB_EXPORT void
7   _plat__RunCommand(
8       uint32_t         requestSize,    // IN: command buffer size
9       unsigned char   *request,        // IN: command buffer
10      uint32_t        *responseSize,   // IN/OUT: response buffer size
11      unsigned char   **response       // IN/OUT: response buffer
12      )
13  {
14      setjmp(s_jumpBuffer);
15      ExecuteCommand(requestSize, request, responseSize, response);
16  }
```

C.12.3.2. _plat__Fail()

This is the platform depended failure exit for the TPM.

```
17  LIB_EXPORT NORETURN void
18  _plat__Fail(
19      void
20      )
21  {
22      longjmp(&s_jumpBuffer[0], 1);
23  }
```

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

```
1   #include "PlatformData.h"
2   #include "Platform_fp.h"
3   const char notReallyUnique[] =
4   "This is not really a unique value. A real unique value should"
5   " 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.

```
6   LIB_EXPORT uint32_t
7   _plat__GetUnique(
8       uint32_t            which,          // authorities (0) or details
9       uint32_t            bSize,          // size of the buffer
10      unsigned char      *b               // output buffer
11      )
12  {
13      const char         *from = notReallyUnique;
14      uint32_t            retVal = 0;
15
16      if(which == 0) // the authorities value
17      {
18          for(retVal = 0;
19          *from != 0 && retVal < bSize;
20              retVal++)
21          {
22              *b++ = *from++;
23          }
24      }
25      else
26      {
27  #define uSize  sizeof(notReallyUnique)
28          b = &b[((bSize < uSize) ? bSize : uSize) - 1];
29          for(retVal = 0;
30          *from != 0 && retVal < bSize;
31              retVal++)
32          {
33              *b-- = *from++;
34          }
35      }
36      return retVal;
37  }
```


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()

This is the 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      );
```

D.2.2.10. _rpc__Send_Command()

This is the interface to the TPM code.

```
93  void
94  _rpc__Send_Command(
95      unsigned char     locality,
```



```
96      _IN_BUFFER        request,
97      _OUT_BUFFER      *response
98      );
```

D.2.2.11. _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.2.2.12. _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.2.2.13. _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.2.2.14. _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.2.2.15. _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.2.2.16. _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. It registers the interface and starts listening for clients

```
123 int
124 main(
125     int              argc,
126     char            *argv[]
127     );
```

D.2.3.2. nothing()

This function does nothing. It only exists so that there are two named functions in this section to avoid a 'dangling paragraph'.

```
128 void
129 nothing(
130     void
131 );
132 #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 completed 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
```

D.3.3. 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
```

D.3.4. Enumerations and Structures

```
enum TpmEndPointInfo
{
    tpmPlatformAvailable = 0x01,
    tpmUsesTbs = 0x02,
    tpmInRawMode = 0x04,
    tpmSupportsPP = 0x08
};

// Existing RPC interface type definitions retained so that the implementation
```

```
37  // can be re-used
38  typedef struct in_buffer
39  {
40      unsigned long BufferSize;
41      unsigned char *Buffer;
42  } _IN_BUFFER;
43
44  typedef unsigned char *_OUTPUT_BUFFER;
45
46  typedef struct out_buffer
47  {
48      uint32_t         BufferSize;
49      _OUTPUT_BUFFER   Buffer;
50  } _OUT_BUFFER;
51
52  #ifndef WIN32
53  typedef unsigned long        DWORD;
54  typedef void                *LPVOID;
55  #undef WINAPI
56  #endif
57
58  #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

```
1   #include "TpmBuildSwitches.h"
2   #include <stdio.h>
3   #ifdef _MSC_VER
4   #include <windows.h>
5   #include <winsock.h>
6   #else
7   typedef int SOCKET;
8   #endif
9   #include <string.h>
10  #include <stdlib.h>
11  #include <stdint.h>
12  //#include "BaseTypes.h"
13  #include "TpmTcpProtocol.h"
14  #include "Manufacture_fp.h"
15  #include "Simulator_fp.h"
```

To access key cache control in TPM

```
16  void RsaKeyCacheControl(int state);
17  #ifndef __IGNORE_STATE__
18  static uint32_t ServerVersion = 1;
19  #define MAX_BUFFER 1048576
20  char InputBuffer[MAX_BUFFER];       //The input data buffer for the simulator.
21  char OutputBuffer[MAX_BUFFER];      //The output data buffer for the simulator.
22  struct
23  {
24      uint32_t    largestCommandSize;
25      uint32_t    largestCommand;
26      uint32_t    largestResponseSize;
27      uint32_t    largestResponse;
28  } CommandResponseSizes = {0};
29  #endif // __IGNORE_STATE___
```

D.4.3. Functions

D.4.3.1. CreateSocket()

This function creates a socket listening on *PortNumber*.

```
30  static int
31  CreateSocket(
32      int                 PortNumber,
33      SOCKET             *listenSocket
34      )
35  {
36      WSADATA             wsaData;
37      struct              sockaddr_in MyAddress;
38      int res;
39  //
40      // Initialize Winsock
41      res = WSAStartup(MAKEWORD(2, 2), &wsaData);
42      if(res != 0)
```

```
43      {
44          printf("WSAStartup failed with error: %d\n", res);
45          return -1;
46      }
47      // create listening socket
48      *listenSocket = socket(PF_INET, SOCK_STREAM, 0);
49      if(INVALID_SOCKET == *listenSocket)
50      {
51          printf("Cannot create server listen socket.  Error is 0x%x\n",
52                 WSAGetLastError());
53          return -1;
54      }
55      // bind the listening socket to the specified port
56      ZeroMemory(&MyAddress, sizeof(MyAddress));
57      MyAddress.sin_port = htons((short)PortNumber);
58      MyAddress.sin_family = AF_INET;
59
60      res = bind(*listenSocket, (struct sockaddr*) &MyAddress, sizeof(MyAddress));
61      if(res == SOCKET_ERROR)
62      {
63          printf("Bind error.  Error is 0x%x\n", WSAGetLastError());
64          return -1;
65      }
66      // listen/wait for server connections
67      res = listen(*listenSocket, 3);
68      if(res == SOCKET_ERROR)
69      {
70          printf("Listen error.  Error is 0x%x\n", WSAGetLastError());
71          return -1;
72      }
73      return 0;
74  }
```

D.4.3.2. PlatformServer()

This function processes incoming platform requests.

```
75  BOOL
76  PlatformServer(
77      SOCKET          s
78      )
79  {
80      BOOL                OK = TRUE;
81      uint32_t            Command;
82  //
83      for(;;)
84      {
85          OK = ReadBytes(s, (char*)&Command, 4);
86          // client disconnected (or other error).  We stop processing this client
87          // and return to our caller who can stop the server or listen for another
88          // connection.
89          if(!OK) return TRUE;
90          Command = ntohl(Command);
91          switch(Command)
92          {
93              case TPM_SIGNAL_POWER_ON:
94                  _rpc__Signal_PowerOn(FALSE);
95                  break;
96              case TPM_SIGNAL_POWER_OFF:
97                  _rpc__Signal_PowerOff();
98                  break;
99              case TPM_SIGNAL_RESET:
100                 _rpc__Signal_PowerOn(TRUE);
101                 break;
```

```
102                 case TPM_SIGNAL_RESTART:
103                     _rpc__Signal_Restart();
104                     break;
105                 case TPM_SIGNAL_PHYS_PRES_ON:
106                     _rpc__Signal_PhysicalPresenceOn();
107                     break;
108                 case TPM_SIGNAL_PHYS_PRES_OFF:
109                     _rpc__Signal_PhysicalPresenceOff();
110                     break;
111                 case TPM_SIGNAL_CANCEL_ON:
112                     _rpc__Signal_CancelOn();
113                     break;
114                 case TPM_SIGNAL_CANCEL_OFF:
115                     _rpc__Signal_CancelOff();
116                     break;
117                 case TPM_SIGNAL_NV_ON:
118                     _rpc__Signal_NvOn();
119                     break;
120                 case TPM_SIGNAL_NV_OFF:
121                     _rpc__Signal_NvOff();
122                     break;
123                 case TPM_SIGNAL_KEY_CACHE_ON:
124                     _rpc__RsaKeyCacheControl(TRUE);
125                     break;
126                 case TPM_SIGNAL_KEY_CACHE_OFF:
127                     _rpc__RsaKeyCacheControl(FALSE);
128                     break;
129                 case TPM_SESSION_END:
130                     // Client signaled end-of-session
131                     TpmEndSimulation();
132                     return TRUE;
133                 case TPM_STOP:
134                     // Client requested the simulator to exit
135                     return FALSE;
136                 case TPM_TEST_FAILURE_MODE:
137                     _rpc__ForceFailureMode();
138                     break;
139                 case TPM_GET_COMMAND_RESPONSE_SIZES:
140                     OK = WriteVarBytes(s, (char *)&CommandResponseSizes,
141                                        sizeof(CommandResponseSizes));
142                     memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
143                     if(!OK)
144                         return TRUE;
145                     break;
146                 default:
147                     printf("Unrecognized platform interface command %d\n",
148                            (int)Command);
149                     WriteUINT32(s, 1);
150                     return TRUE;
151             }
152             WriteUINT32(s, 0);
153         }
154         return FALSE;
155     }
```

D.4.3.3. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

```
156     DWORD WINAPI
157     PlatformSvcRoutine(
158         LPVOID          port
159         )
160     {
```

```
161     int                     PortNumber = (int)(INT_PTR)port;
162     SOCKET                  listenSocket, serverSocket;
163     struct                  sockaddr_in HerAddress;
164     int                     res;
165     int                     length;
166     BOOL                    continueServing;
167 //
168     res = CreateSocket(PortNumber, &listenSocket);
169     if(res != 0)
170     {
171         printf("Create platform service socket fail\n");
172         return res;
173     }
174     // Loop accepting connections one-by-one until we are killed or asked to stop
175     // Note the platform service is single-threaded so we don't listen for a new
176     // connection until the prior connection drops.
177     do
178     {
179         printf("Platform server listening on port %d\n", PortNumber);
180
181         // blocking accept
182         length = sizeof(HerAddress);
183         serverSocket = accept(listenSocket,
184                               (struct sockaddr*) &HerAddress,
185                               &length);
186         if(serverSocket == SOCKET_ERROR)
187         {
188             printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
189             return -1;
190         }
191         printf("Client accepted\n");
192
193         // normal behavior on client disconnection is to wait for a new client
194         // to connect
195         continueServing = PlatformServer(serverSocket);
196         closesocket(serverSocket);
197     } while(continueServing);
198
199     return 0;
200 }
```

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.

```
201 int
202 PlatformSignalService(
203     int             PortNumber
204     )
205 {
206     HANDLE                  hPlatformSvc;
207     int                     ThreadId;
208     int                     port = PortNumber;
209 //
210     // Create service thread for platform signals
211     hPlatformSvc = CreateThread(NULL, 0,
212                                 (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
213                                 (LPVOID)(INT_PTR)port, 0, (LPDWORD)&ThreadId);
214     if(hPlatformSvc == NULL)
215     {
216         printf("Thread Creation failed\n");
217         return -1;
218     }
```

```
219     return 0;
220 }
```

D.4.3.5. RegularCommandService()

This function services regular commands.

```
221 int
222 RegularCommandService(
223     int                 PortNumber
224     )
225 {
226     SOCKET                  listenSocket;
227     SOCKET                  serverSocket;
228     struct                  sockaddr_in HerAddress;
229     int                     res, length;
230     BOOL                    continueServing;
231 //
232     res = CreateSocket(PortNumber, &listenSocket);
233     if(res != 0)
234     {
235         printf("Create platform service socket fail\n");
236         return res;
237     }
238     // Loop accepting connections one-by-one until we are killed or asked to stop
239     // Note the TPM command service is single-threaded so we don't listen for
240     // a new connection until the prior connection drops.
241     do
242     {
243         printf("TPM command server listening on port %d\n", PortNumber);
244
245         // blocking accept
246         length = sizeof(HerAddress);
247         serverSocket = accept(listenSocket,
248                               (struct sockaddr*) &HerAddress,
249                               &length);
250         if(serverSocket == SOCKET_ERROR)
251         {
252            printf("Accept error.  Error is 0x%x\n", WSAGetLastError());
253            return -1;
254         }
255         printf("Client accepted\n");
256
257         // normal behavior on client disconnection is to wait for a new client
258         // to connect
259         continueServing = TpmServer(serverSocket);
260         closesocket(serverSocket);
261     } while(continueServing);
262     return 0;
263 }
```

D.4.3.6. StartTcpServer()

This is the 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.

```
264 int
265 StartTcpServer(
266     int                 PortNumber
267     )
268 {
269     int                     res;
```

```
270 //
271     // Start Platform Signal Processing Service
272     res = PlatformSignalService(PortNumber + 1);
273     if(res != 0)
274     {
275         printf("PlatformSignalService failed\n");
276         return res;
277     }
278     // Start Regular/DRTM TPM command service
279     res = RegularCommandService(PortNumber);
280     if(res != 0)
281     {
282         printf("RegularCommandService failed\n");
283         return res;
284     }
285     return 0;
286 }
```

D.4.3.7. ReadBytes()

This function reads the indicated number of bytes *(NumBytes)* into buffer from the indicated socket.

```
287 BOOL
288 ReadBytes(
289     SOCKET           s,
290     char            *buffer,
291     int              NumBytes
292     )
293 {
294     int                 res;
295     int                 numGot = 0;
296 //
297     while(numGot < NumBytes)
298     {
299         res = recv(s, buffer + numGot, NumBytes - numGot, 0);
300         if(res == -1)
301         {
302             printf("Receive error.  Error is 0x%x\n", WSAGetLastError());
303             return FALSE;
304         }
305         if(res == 0)
306         {
307             return FALSE;
308         }
309         numGot += res;
310     }
311     return TRUE;
312 }
```

D.4.3.8. WriteBytes()

This function will send the indicated number of bytes *(NumBytes)* to the indicated socket

```
313 BOOL
314 WriteBytes(
315     SOCKET           s,
316     char            *buffer,
317     int              NumBytes
318     )
319 {
320     int                 res;
321     int                 numSent = 0;
322 //
```



```
323         while(numSent < NumBytes)
324         {
325             res = send(s, buffer + numSent, NumBytes - numSent, 0);
326             if(res == -1)
327             {
328                 if(WSAGetLastError() == 0x2745)
329                 {
330                     printf("Client disconnected\n");
331                 }
332                 else
333                 {
334                     printf("Send error.  Error is 0x%x\n", WSAGetLastError());
335                 }
336                 return FALSE;
337             }
338             numSent += res;
339         }
340         return TRUE;
341     }
```

D.4.3.9. WriteUINT32()

Send 4 bytes containing hton(1)

```
342     BOOL
343     WriteUINT32(
344         SOCKET          s,
345         uint32_t        val
346         )
347     {
348         UINT32 netVal = htonl(val);
349     //
350         return WriteBytes(s, (char*)&netVal, 4);
351     }
```

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).

```
352     BOOL
353     ReadVarBytes(
354         SOCKET          s,
355         char           *buffer,
356         uint32_t       *BytesReceived,
357         int             MaxLen
358         )
359     {
360         int                 length;
361         BOOL                res;
362     //
363         res = ReadBytes(s, (char*)&length, 4);
364         if(!res) return res;
365         length = ntohl(length);
366         *BytesReceived = length;
367         if(length > MaxLen)
368         {
369             printf("Buffer too big.  Client says %d\n", length);
370             return FALSE;
371         }
372         if(length == 0) return TRUE;
373         res = ReadBytes(s, buffer, length);
374         if(!res) return res;
```

```
375     return TRUE;
376 }
```

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).

```
377 BOOL
378 WriteVarBytes(
379     SOCKET          s,
380     char           *buffer,
381     int             BytesToSend
382     )
383 {
384     uint32_t            netLength = htonl(BytesToSend);
385     BOOL res;
386 //
387     res = WriteBytes(s, (char*)&netLength, 4);
388     if(!res)
389         return res;
390     res = WriteBytes(s, buffer, BytesToSend);
391     if(!res)
392         return res;
393     return TRUE;
394 }
```

D.4.3.12. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```
395 BOOL
396 TpmServer(
397     SOCKET          s
398     )
399 {
400     uint32_t            length;
401     uint32_t            Command;
402     BYTE                locality;
403     BOOL                OK;
404     int                 result;
405     int                 clientVersion;
406     _IN_BUFFER          InBuffer;
407     _OUT_BUFFER         OutBuffer;
408 //
409     for(;;)
410     {
411         OK = ReadBytes(s, (char*)&Command, 4);
412         // client disconnected (or other error).  We stop processing this client
413         // and return to our caller who can stop the server or listen for another
414         // connection.
415         if(!OK)
416             return TRUE;
417         Command = ntohl(Command);
418         switch(Command)
419         {
420             case TPM_SIGNAL_HASH_START:
421                 _rpc__Signal_Hash_Start();
422                 break;
423             case TPM_SIGNAL_HASH_END:
424                 _rpc__Signal_HashEnd();
425                 break;
426             case TPM_SIGNAL_HASH_DATA:
```



```
427                 OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
428                 if(!OK) return TRUE;
429                 InBuffer.Buffer = (BYTE*)InputBuffer;
430                 InBuffer.BufferSize = length;
431                 _rpc__Signal_Hash_Data(InBuffer);
432                 break;
433             case TPM_SEND_COMMAND:
434                 OK = ReadBytes(s, (char*)&locality, 1);
435                 if(!OK)
436                     return TRUE;
437                 OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
438                 if(!OK)
439                     return TRUE;
440                 InBuffer.Buffer = (BYTE*)InputBuffer;
441                 InBuffer.BufferSize = length;
442                 OutBuffer.BufferSize = MAX_BUFFER;
443                 OutBuffer.Buffer = (_OUTPUT_BUFFER)OutputBuffer;
444                 // record the number of bytes in the command if it is the largest
445                 // we have seen so far.
446                 if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
447                 {
448                     CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
449                     memcpy(&CommandResponseSizes.largestCommand,
450                            &InputBuffer[6], sizeof(UINT32));
451                 }
452                 _rpc__Send_Command(locality, InBuffer, &OutBuffer);
453                 // record the number of bytes in the response if it is the largest
454                 // we have seen so far.
455                 if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
456                 {
457                     CommandResponseSizes.largestResponseSize
458                         = OutBuffer.BufferSize;
459                     memcpy(&CommandResponseSizes.largestResponse,
460                            &OutputBuffer[6], sizeof(UINT32));
461                 }
462                 OK = WriteVarBytes(s,
463                                    (char*)OutBuffer.Buffer,
464                                    OutBuffer.BufferSize);
465                 if(!OK)
466                     return TRUE;
467                 break;
468             case TPM_REMOTE_HANDSHAKE:
469                 OK = ReadBytes(s, (char*)&clientVersion, 4);
470                 if(!OK)
471                     return TRUE;
472                 if(clientVersion == 0)
473                 {
474                     printf("Unsupported client version (0).\n");
475                     return TRUE;
476                 }
477                 OK &= WriteUINT32(s, ServerVersion);
478                 OK &= WriteUINT32(s, tpmInRawMode
479                                  | tpmPlatformAvailable | tpmSupportsPP);
480                 break;
481             case TPM_SET_ALTERNATIVE_RESULT:
482                 OK = ReadBytes(s, (char*)&result, 4);
483                 if(!OK)
484                     return TRUE;
485                 // Alternative result is not applicable to the simulator.
486                 break;
487             case TPM_SESSION_END:
488                 // Client signaled end-of-session
489                 return TRUE;
490             case TPM_STOP:
491                 // Client requested the simulator to exit
492                 return FALSE;
```

```
493                 default:
494                     printf("Unrecognized TPM interface command %d\n", (int)Command);
495                     return TRUE;
496             }
497             OK = WriteUINT32(s, 0);
498             if(!OK)
499                 return TRUE;
500         }
501         return FALSE;
502     }
```


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

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <setjmp.h>
4   #include "TpmBuildSwitches.h"
5   #include <windows.h>
6   #include <winsock.h>
7   #include "Platform_fp.h"
8   #include "ExecCommand_fp.h"
9   #include "Manufacture_fp.h"
10  #include "_TPM_Init_fp.h"
11  #include "_TPM_Hash_Start_fp.h"
12  #include "_TPM_Hash_Data_fp.h"
13  #include "_TPM_Hash_End_fp.h"
14  #include "TpmFail_fp.h"
15  #include "TpmTcpProtocol.h"
16  #include "Simulator_fp.h"
17  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.

```
18  void
19  _rpc__Signal_PowerOn(
20      BOOL        isReset
21      )
22  {
23      // if power is on and this is not a call to do TPM reset then return
24      if(s_isPowerOn && !isReset)
25          return;
26
27      // If this is a reset but power is not on, then return
28      if(isReset && !s_isPowerOn)
29          return;
30
31      // Unless this is just a reset, pass power on signal to platform
32      if(!isReset)
33          _plat__Signal_PowerOn();
34
35      // Power on and reset both lead to _TPM_Init()
36      _plat__Signal_Reset();
37
38      // Set state as power on
39      s_isPowerOn = TRUE;
40  }
```


D.5.3.2. Signal_Restart()

This function processes the clock restart indication. All it does is call the platform function.

```
41  void
42  _rpc__Signal_Restart(
43      void
44      )
45  {
46      _plat__TimerRestart();
47  }
```

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.

```
48  void
49  _rpc__Signal_PowerOff(
50      void
51      )
52  {
53      if(!s_isPowerOn) return;
54
55      // Pass power off signal to platform
56      _plat__Signal_PowerOff();
57
58      s_isPowerOn = FALSE;
59
60      return;
61  }
```

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.

```
62  void
63  _rpc__ForceFailureMode(
64      void
65      )
66  {
67      SetForceFailureMode();
68  }
```

D.5.3.5. _rpc__Signal_PhysicalPresenceOn()

This function is called to simulate activation of the physical presence **pin**.

```
69  void
70  _rpc__Signal_PhysicalPresenceOn(
71      void
72      )
73  {
74      // If TPM is power off, reject this signal
75      if(!s_isPowerOn) return;
76
77      // Pass physical presence on to platform
78      _plat__Signal_PhysicalPresenceOn();
79
```



```
80      return;
81  }
```

D.5.3.6. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence **pin**.

```
82  void
83  _rpc__Signal_PhysicalPresenceOff(
84      void
85      )
86  {
87      // If TPM is power off, reject this signal
88      if(!s_isPowerOn) return;
89
90      // Pass physical presence off to platform
91      _plat__Signal_PhysicalPresenceOff();
92
93      return;
94  }
```

D.5.3.7. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start() event. It will call

```
95  void
96  _rpc__Signal_Hash_Start(
97      void
98      )
99  {
100     // If TPM is power off, reject this signal
101     if(!s_isPowerOn) return;
102
103     // Pass _TPM_Hash_Start signal to TPM
104     _TPM_Hash_Start();
105     return;
106 }
```

D.5.3.8. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data() event.

```
107 void
108 _rpc__Signal_Hash_Data(
109     _IN_BUFFER       input
110     )
111 {
112     // If TPM is power off, reject this signal
113     if(!s_isPowerOn) return;
114
115     // Pass _TPM_Hash_Data signal to TPM
116     _TPM_Hash_Data(input.BufferSize, input.Buffer);
117     return;
118 }
```

D.5.3.9. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End() event.

```
119 void
```

```
120 _rpc__Signal_HashEnd(
121     void
122     )
123 {
124     // If TPM is power off, reject this signal
125     if(!s_isPowerOn) return;
126
127     // Pass _TPM_HashEnd signal to TPM
128     _TPM_Hash_End();
129     return;
130 }
```

D.5.3.10. _rpc__Send_Command()

This is the interface to the TPM code.

```
131 void
132 _rpc__Send_Command(
133     unsigned char    locality,
134     _IN_BUFFER       request,
135     _OUT_BUFFER     *response
136     )
137 {
138     // If TPM is power off, reject any commands.
139     if(!s_isPowerOn)
140     {
141         response->BufferSize = 0;
142         return;
143     }
144     // Set the locality of the command so that it doesn't change during the command
145     _plat__LocalitySet(locality);
146     // Do implementation-specific command dispatch
147     _plat__RunCommand(request.BufferSize, request.Buffer,
148                       &response->BufferSize, &response->Buffer);
149     return;
150 }
```

D.5.3.11. _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.

```
151 void
152 _rpc__Signal_CancelOn(
153     void
154     )
155 {
156     // If TPM is power off, reject this signal
157     if(!s_isPowerOn) return;
158
159     // Set the platform canceling flag.
160     _plat__SetCancel();
161
162     return;
163 }
```

D.5.3.12. _rpc__Signal_CancelOff()

This function is used to turn off the indication to cancel a command in process.


```
164 void
165 _rpc__Signal_CancelOff(
166     void
167     )
168 {
169     // If TPM is power off, reject this signal
170     if(!s_isPowerOn) return;
171
172     // Set the platform canceling flag.
173     _plat__ClearCancel();
174
175     return;
176 }
```

D.5.3.13. _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.

```
177 void
178 _rpc__Signal_NvOn(
179     void
180     )
181 {
182     // If TPM is power off, reject this signal
183     if(!s_isPowerOn) return;
184
185     _plat__SetNvAvail();
186     return;
187 }
```

D.5.3.14. _rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

```
188 void
189 _rpc__Signal_NvOff(
190     void
191     )
192 {
193     // If TPM is power off, reject this signal
194     if(!s_isPowerOn) return;
195
196     _plat__ClearNvAvail();
197     return;
198 }
199 void RsaKeyCacheControl(int state);
```

D.5.3.15. _rpc__RsaKeyCacheControl()

This function is used to enable/disable the use of the RSA key cache during simulation.

```
200 void
201 _rpc__RsaKeyCacheControl(
202     int                 state
203     )
204 {
205 #if USE_RSA_KEY_CACHE
206     RsaKeyCacheControl(state);
207 #else
208     NOT_REFERENCED(state);
```

```
209 #endif
210 }
```

D.5.3.16. _rpc__Shutdown()

This function is used to stop the TPM simulator.

```
211 void
212 _rpc__Shutdown(
213     void
214     )
215 {
216     RPC_STATUS status;
217
218     // Stop TPM
219     TPM_TearDown();
220
221     status = RpcMgmtStopServerListening(NULL);
222     if(status != RPC_S_OK)
223     {
224         printf("RpcMgmtStopServerListening returned: 0x%x\n", status);
225         exit(status);
226     }
227
228     status = RpcServerUnregisterIf(NULL, NULL, FALSE);
229     if(status != RPC_S_OK)
230     {
231         printf("RpcServerUnregisterIf returned 0x%x\n", status);
232         exit(status);
233     }
234     return;
235 }
```

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

```
1   #include "TpmBuildSwitches.h"
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <stdint.h>
5   #include <ctype.h>
6   #include <string.h>
7   #include <windows.h>
8   #include <winsock.h>
9   #include "TpmTcpProtocol.h"
10  #include "Manufacture_fp.h"
11  #include "Platform_fp.h"
12  #include "Simulator_fp.h"
13  #define PURPOSE \
14  "TPM Reference Simulator.\nCopyright Microsoft Corp.\n"
15  #define DEFAULT_TPM_PORT 2321
16  void* MainPointer;
```

D.6.3. Functions

D.6.3.1. Usage()

This function prints the proper calling sequence for the simulator.

```
17  static void
18  Usage(
19      char                *pszProgramName
20      )
21  {
22      fprintf(stderr, "%s", PURPOSE);
23      fprintf(stderr, "Usage:\n");
24      fprintf(stderr, "%s        - Starts the TPM server listening on port %d\n",
25                pszProgramName, DEFAULT_TPM_PORT);
26      fprintf(stderr,
27                "%s PortNum - Starts the TPM server listening on port PortNum\n",
28                pszProgramName);
29      fprintf(stderr, "%s ?      - This message\n", pszProgramName);
30      exit(1);
31  }
```

D.6.3.2. main()

This is the main entry point for the simulator. It registers the interface and starts listening for clients

```
32  int
33  main(
34      int              argc,
35      char            *argv[]
36      )
37  {
38      int portNum = DEFAULT_TPM_PORT;
39      if(argc > 2)
```

```
40      {
41          Usage(argv[0]);
42      }
43
44      if(argc == 2)
45      {
46          if(strcmp(argv[1], "?") == 0)
47          {
48              Usage(argv[0]);
49          }
50          portNum = atoi(argv[1]);
51          if(portNum <= 0 || portNum > 65535)
52          {
53              Usage(argv[0]);
54          }
55      }
56      _plat__NVEnable(NULL);
57
58      if(TPM_Manufacture(1) != 0)
59      {
60          exit(1);
61      }
62      // Coverage test - repeated manufacturing attempt
63      if(TPM_Manufacture(0) != 1)
64      {
65          exit(2);
66      }
67      // Coverage test - re-manufacturing
68      TPM_TearDown();
69      if(TPM_Manufacture(1) != 0)
70      {
71          exit(3);
72      }
73      // Disable NV memory
74      _plat__NVDisable();
75
76      StartTcpServer(portNum);
77      return EXIT_SUCCESS;
78  }
```

D.6.3.3. nothing()

This function does nothing. It only exists so that there are two named functions in this section to avoid a 'dangling paragraph'.

```
79  void
80  nothing(
81      void
82  )
83  {
84  }
```

